Scientific and Statistical Database Management (SSDBM 2004), Santorini Island, Greece, 2004.

Spatial Join for High-Resolution Objects

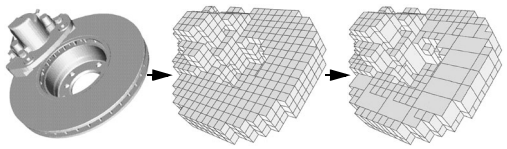
Hans-Peter Kriegel, Peter Kunath, Martin Pfeifle, Matthias Renz
University of Munich, Germany, {kriegel, kunath, pfeifle, renz}@dbs.informatik.uni-muenchen.de

Abstract

Modern database applications including computer-aided design (CAD), medical imaging, molecular biology, or Multimedia Information Systems impose new requirements on efficient spatial query processing. One of the most common query types in Spatial Database Management Systems is the spatial join. In this paper, we investigate spatial join processing for two sets of very complex spatial objects. We present an approach that is based on a fast filter step performing the spatial join on simple primitives which conservatively approximate the objects. Our main attention is focused on the problem how to generate approximations adequate for high-resolution objects. In this paper, we introduce gray approximations as a general concept which helps to range between replicating and non-replicating object approximations. The key idea of our approach is to build these replications based on statistical information taking the data distribution of the respective join-partner relation into account. Furthermore, our approach uses compression techniques for the effective storage and retrieval of the decomposed spatial objects. We demonstrate the benefits of our new method for the spatial intersection join on high resolution data. The experimental evaluation on real-world test data points out that our new concept accelerates the spatial intersection join considerably.

1. Introduction

The efficient management of complex objects has become an enabling technology for geographical information systems (GIS) as well as for many novel database applications, including computer aided design (CAD), medical imaging, molecular biology, and Multimedia Information Systems. One of the most common query types in Spatial Database Management Systems is the *spatial join*. In this paper, we concentrate on the intersection join, as the intersection is the most important join predicate for complex spatial objects [9]. The intersection join retrieves all object pairs from two given data sets that satisfy the spatial-intersection predicate, i.e. all pairs of overlapping objects are reported. A usual spatial join example of 2D geographical data is "find all cities which are crossed by a river". In the automobile industry, spatial join processing of complex 3D high-resolution objects is also required, e.g. to support efficient processing of queries like "find all engine parts which intersect the car body". Thereby an efficient processing of spatial joins is indispensable.



a) Spatial object **b)** Voxel set **c)** Interval sequence

Figure 1: Conversion pipeline from spatial objects to voxel interval sequences

An important new requirement for large objects, including cars, planes or space stations, is a high approximation quality. As a common and successful approach, spatial objects can be conservatively approximated by a set of voxels, i.e. cells of a grid covering the complete data space [17]. By means of space filling curves which achieve good spatial clustering properties, each cell of the grid can be encoded by a single z-value and, thus, an extended object is represented by a set of z-values.

High resolutions yield a high approximation quality but result in high efforts in terms of identifying the join candidate pairs. Thus, the join performance is primarily influenced by the size of the voxel sets, i.e. it depends on the resolution of the grid. We aim to manage very complex objects, e.g. the "777" from Boeing, which was completely digitally designed and assembled. It consists of about three million parts, some of which are composed of several millions of voxels. Following [11], adjacent cell values can be grouped together to *voxel interval sequences* (cf. Figure 1) which are basic datatypes for spatial applications. However, in the case of high-resolution data, the number of resulting intervals still remains very high.

In this paper, we introduce a method for spatial intersection join, especially designed to cope with high resolution objects. Our approach is not confined to the availability of a spatial index. It is based on grouping large sets of object voxels into few container objects in a preceding preprocessing step, and to perform the spatial join on these container objects.

1.1. Basic join algorithm

Figure 2 outlines the procedure for joining two sets of objects stored in two relations *R* and *S*. *id* denotes a unique object identifier and *link* refers to an external file containing the complete voxel set of an object. The overall join procedure is composed of two phases: in the first phase (*preprocessing phase*), we convert the voxels for each object of *S* into z-values, group the z-values into a set of container objects and store them in an auxiliary relation *S'* (cf. Figure 2a). The attribute *approx* is realized as *Nested Table* storing the set of container objects for each object in *S*. Thereby, each container object is composed of the respective z-value sequence and a minimal covering interval. In the second phase (*join phase*), we perform a *nested-loop join* between relation *R* and *S'*, where *R* is accessed within the outer join loop. Before joining each object of *R* with *S'*, we convert the voxels into z-values and group the z-values of each object into container objects on-the-fly (cf. Figure 2b), i.e. the grouping process of relation *R* is embedded within the outer loop of the *nested-loop join*. In a fast *filter step*, we use for each join pair the minimal covering intervals of the container objects to check them for intersection. Subsequently, in a potential expensive *refinement step* each *positive* candidate from the filter step has to be checked for intersection with respect to its exact geometry by considering the z-value sequences. In the rest of this paper, we refer to this basic join procedure performed on relation *R* and *S*, respectively *S'*.

The key idea that we use for grouping the z-values is a cost-based decomposition algorithm introduced in this paper which takes statistics about the data distribution into account. Furthermore, we deploy compression algorithms for the efficient storage and retrieval of the container objects of relation *S'*.

1.2. Outline of this paper

The remainder of the paper is organized as follows: Section 2 provides summaries of different aspects for efficient spatial join processing and decompositioning of complex spatial objects. Section 3 presents a cost-based decompositioning algorithm for generating container objects introduced as *gray intervals*. In Section 4, we show how the join procedure can be accelerated by using the generated *gray intervals*. In Section 5 we present a detailed experimental evaluation demonstrating the benefits of our approach. Finally, in Section 6, we summarize our work, and conclude the paper with a few remarks on future work.

2. Related work

In this section, we will shortly discuss different aspects of efficient spatial join processing of complex spatial objects.

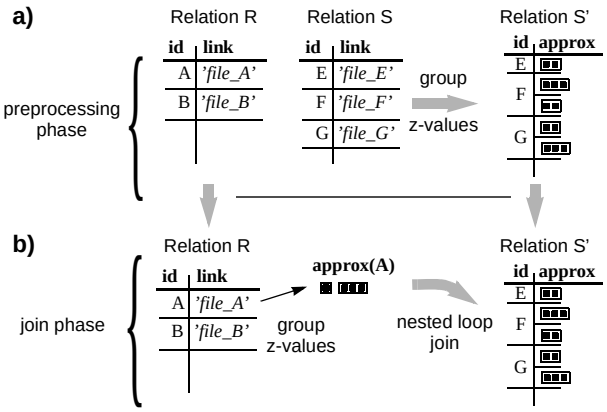


Figure 2: Spatial join procedure
a) Preprocessing phase b) Join phase

Spatial Join. Numerous spatial join algorithms have been proposed over the last decade. Most of them rely on the paradigm of multi-step query processing [3]. A fast *filter step* excludes all objects that cannot satisfy the join predicate. The subsequent *refinement step* is applied to the join candidate pairs which are returned from the *filter step*. Thereby, the main focus of research is on the *filter step* which is applied to geometric object approximations. On the basis of the availability of indices for processing the filter step, spatial join methods operating on two relations can be classified into three classes: index on both relations (Class 1), index on one relation (Class 2) and no indices (Class 3).

The common solutions for the spatial join methods of Class 1 are the algorithms based on matching two R-trees as presented in [2]. In the last few years, the international research community has focused on methods of Class 2 and Class 3. A simple Class 2 approach is the *index nested loop*, where each tupel of the non-indexed relation is used as query applied to the indexed relation. In [13], seeded trees were introduced in oder to process spatial joins efficiently when only one R-tree is available. The authors propose to create a second R-tree using the available tree as a skeleton and apply thereafter a Class 1 algorithm. For spatial join algorithms of Class 3, initially no indices are available which could be used to improve the query performance. Several techniques have been proposed which partition the tuples into buckets and then use hash based techniques, e.g. the *spatial-hash join* [14] or the *partition based spatial merge join* [18]. The *scalable sweeping-based spatial join* [1] is w.r.t. worst-case efficiency the most promising algorothm for processing spatial joins. The latter approaches work well for relative simply shaped 2D objects, which can be well approximated by their minimal bounding boxes. In contrast to these approaches, our approach deals with very complex 3D objects, where the minimal bounding box is a rather poor approximation.

Decomposition of Complex Spatial Objects. Approximations of extended objects generally consist of either one or several simple spatial primitives such as minimal bounding boxes which are often used for one-value approximations [3, 9]. Although providing the minimal storage complexity, one-value approximations of spatially extended objects often are far too coarse. In many applications, GIS or CAD objects feature a very complex and fine-grained geometry. The rectilinear bounding box of the brake line of a car, for example, would cover the whole bottom of the data space. A non-replicating storage of such data would cause too many false hits in the *filter step* of the join that have to be eliminated by the *refinement step*.

In contrast, approaches which use multi-value approximations, i.e. approximations which are composed of several spatial primitives, can achieve a better approximation than a single rectangle. In the case of a very accurate approximation, the number of primitives can become very high. For instance, Gaede [8] pointed out that the number of z-value intervals representing a spatially extended object exponentially depends on the granularity of the grid approximation. Furthermore, the extensive analyses given in [15] and [7] show that the asymptotic redundancy of an interval-based decomposition is proportional to the surface of the approximated object. Thus, in the case of high resolution huge parts (e.g. wings of an airplane), the number of intervals can become unreasonably high, which results in too many intersect verifications in the *filter step* of the join procedure.

A promising solution for a good trade-off between these conflicting objectives may be found somewhere in between one-value and multi-value object approximations. In [20], Kriegel and Schiwietz tackled the complex problem of "*complexity versus redundancy*" for 2D polygons. They investigated the natural trade-off between the complexity of the components and the redundancy, i.e. the number of components, with respect to its effect on efficient query processing. The presented empirically derived root-criterion suggests to decompose a polygon consisting of n vertices into $O(\sqrt{n})$ many simple approximations. 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 cost-based decomposition approach which can be used for 2D and 3D objects. It takes the cost of the filter step and refinement step of the join procedure into account.

3. Cost-based decomposition of complex spatial objects

In the following, the geometry of a spatial object is assumed to be described by a sequence of pixels/voxels, ordered by a space-filling curve, e.g. z-curve [17]. High resolution spatial objects may consist of several hundreds of thousands of voxels. For each object, there exist a lot of different possibilities to decompose it into approximations by grouping numerous voxels together. Following the approach that the voxels are linearized by a space-filling curve, the voxels are grouped into one-dimensional intervals. We call these intervals *gray intervals* throughout the rest of this paper. Informally spoken, *gray intervals* bridge the gap between *black intervals* obtained by simply connecting adjacent voxels together.

In the remainder of this section, we will introduce a cost-based grouping algorithm which finds an optimal trade-off between replicating and non-replicating approximations. In Section 3.1, we first introduce our gray intervals formally, and show how they can be integrated into a commercial ORDBMS. In Section 3.2, we discuss why it is beneficial to store the gray containers in a compressed way. In Section 3.3, we introduce our cost-based grouping algorithm for complex spatial objects.

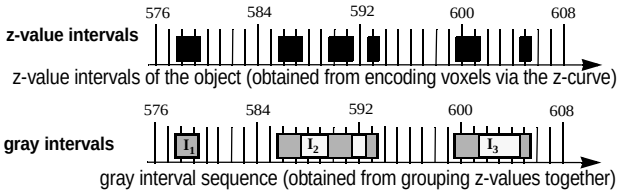
3.1. Gray intervals

Gray intervals are formed by grouping object voxels representing complex spatial objects. By means of space filling curves, $\rho: IN^d \rightarrow IN$, all multidimensional voxelized objects can be mapped to one-dimensional voxelized objects containing a set of integers. In our approach we employ the z-curve as space filling curve, thus an object is represented by a set of *z-values.*

Obviously, we can group adjacent z-values together to a *z-value interval*. The resulting sequence of intervals, representing a high resolution spatially extended object, often consists of very short intervals connected by short gaps. Experiments suggest that both gaps and intervals obey an exponential distribution (cf. Section 5). In order to overcome this obstacle, it seems promising to group adjacent voxel intervals together to longer intervals, which we call *gray intervals*.

Definition 1 (*gray interval, gray interval sequence*)
Let $W = \{(l, u) \in IN^2, l \le u\}$ be the domain of *z-value intervals*, where (l, u) contains all z-values z such that $l \le z \le u$. Furthermore, let $b_1 = (l_1, u_1), \ldots, b_n = (l_n, u_n) \in W$ be a sequence of *z-value intervals* of the object having *id* as its *object identifier*, with $u_i + 1 < l_{i+1}$ for all $i \in \{1, \ldots, n-1\}$. Moreover, let $m \le n$ and let $i_0, i_1, i_2, \ldots, i_m \in IN$ 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 interval sequence* of cardinality m. 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}.

Intuitively, a *gray interval* is a covering of one or more disjoint and nonadjacent *z-value intervals* where there is at least a gap of one z-value between adjacent intervals. In the next definition, we introduce a few useful operators on *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'

GrayIntervals			
id	*data*		
	$H(I_x)$	$D(I_x)$	$B(I_x)$
...	...	...	...
E	[578, 579]	1	'30'
	[586, 593]	5/8	'3340'
	[600, 605]	3/6	'C4'
...	...	...	...

Figure 3: Gray interval sequence

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 $0 \le s_i < 2^8$, $n = \lfloor u_s/8 \rfloor - \lfloor l_r/8 \rfloor$

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

Figure 3 demonstrates the values of some of these operators for a sample set of *gray intervals*.

In our approach, the *z-value intervals* $b_r, \ldots, b_s$ of each *gray interval* $I_{gray} = \langle b_r, \ldots, b_s \rangle$ are mapped to the complex attribute *data* of the relation *GrayIntervals* which is in Non-First-Normal-Form (NF^2). It consists of the hull $H(I_{gray})$, the density $D(I_{gray})$ and a *BLOB* containing the byte sequence $B(I_{gray})$ representing the exact geometry. Important advantages of this approach are as follows: First, the hulls $H(I_{gray})$ of the *gray intervals* can be used in a preceding fast primary filter step. Secondly, the density $D(I_{gray})$ can help to detect absolute intersections between two gray intervals without accessing the exact geometry $B(I_{gray})$. This second filter is specified in Section 4.3. Furthermore, we use the ability to store the content of a *BLOB* outside of the table. Therefore the column $B(I_{gray})$ contains a *BLOB locator*. This enables us to access the possibly huge *BLOB* content only if it is required and not automatically at the access time of the

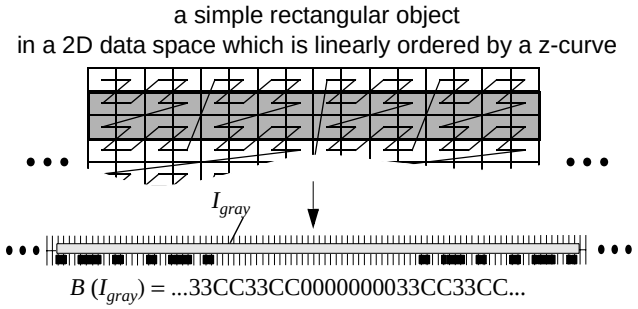


Figure 4: Pattern derivation by linearizing a voxelized object using a space-filling curve

rows. In the next section we discuss how the I/O cost of the BLOBs can be reduced by applying compression techniques.

Let us note, that each complex object is stored in one single row, where the corresponding gray intervals are managed in a nested table. This approach helps us to avoid costly duplicate elimination during the join processing.

3.2. Compression of gray intervals

In this section, we motivate the use of packers, by showing that $B(I_{gray})$ contains patterns. Therefore, $B(I_{gray})$ can efficiently be shrunken by using data compressors. Furthermore, we discuss the properties which a suitable compression algorithm should fulfill. In the following, we give a brief presentation of a new effective packer which seems promising for our approach. It exploits gaps and patterns included in the byte sequence $B(I_{gray})$ of our *gray interval* I_{gray}.

3.2.1. Patterns. To describe a rectangle in a 2D vector space we only need 4 numerical values, e.g. we need two 2-dimensional points. In contrast to the vector representation, an enormous redundancy is contained in the corresponding voxel sequence of an object, an example is shown in Figure 4. As space filling curves enumerate the data space in a structured way, we can find such "structures" in the resulting voxel sequence representing simply shaped objects. We can pinpoint the same phenomenon not only for simply shaped parts but also for more complex real-world spatial parts. Assuming we cover the whole voxel sequence of an object *id* by one interval, i.e. $O_{gray} = (id, \langle I_{gray} \rangle)$, and survey its byte representation $B(I_{gray})$ in a hex-editor, we can notice that some byte sequences occur repeatedly. For more details about the existence of patterns in $B(I_{gray})$ we refer the reader to [12]. We will now discuss how these patterns can be used for the efficient storage of *gray intervals* in an ORDBMS.

3.2.2. Compression rules. A voxel set belonging to a *gray interval* I_{gray} can be materialized and stored in a *BLOB* in

many different ways. A good materialization should consider two "*compression rules*":

Rule 1:	As little as possible secondary storage should be occupied.
Rule 2:	As little as possible time should be needed for the (de)compression of the *BLOB*.

A good join response behavior is based on the fulfillment of both aspects. The first rule guarantees that the I/O cost $c_{\mathrm{I/O}}^{BLOB}$ are relatively small whereas the second rule is responsible for low CPU cost c_{CPU}^{BLOB}. The overall time $c^{BLOB} = c_{\mathrm{I/O}}^{BLOB} + c_{\mathrm{CPU}}^{BLOB}$ for the evaluation of a BLOB is composed of both parts. A good behavior related to an efficient retrieval and evaluation of $B(I_{gray})$ depends on the fulfillment of both rules.

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

3.2.3. Spatial compression techniques. In our approach we deploy the *Quick Spatial Data Compressor* (*QSDC*) algorithm, which is especially designed for high resolution spatial data and includes specific features for the efficient handling of patterns and gaps. It is optimized for speed and does not perform time intensive computations as for instance Huffman compression. *QSDC* is a derivation of the *LZ77* technique [6]. However, it compresses data in only one pass and much faster than other Lempel-Ziv based compression schemes as for example *XRAY* [5]. For more details we refer the reader to [13].

3.3. Cost-based grouping

For our grouping algorithm we take the estimated join cost between a *gray interval* I_{gray} and a join-partner relation T into account. The overall join cost $cost_{join}$ is composed of two parts, the filter cost $cost_{filter}$ and the refinement cost $cost_{refine}$:

$$cost_{join}(I_{gray}, T) = cost_{filter}(I_{gray}, T) + cost_{refine}(I_{gray}, T).$$

The question at issue is, which grouping is most suitable for an efficient join process. A good grouping should take the following "*grouping rules*" into consideration:

***Rule 1*:**	The number of gray intervals should be small.
***Rule 2*:**	The approximation error of all gray intervals should be small.
***Rule 3*:**	The gray intervals should allow an efficient evaluation of the contained voxels.

The first rule guarantees that $cost_{filter}$ is small, as each *gray interval* $I_{gray}(T)$ of the join-partner relation T is a potential filter candidate, which has to be loaded from disk (*BLOB* content excluded) and evaluated for intersection with respect to their hulls.

In contrast, the second rule guarantees that many unnecessary candidate tests of the refinement step can be omitted, as the number and size of gaps included in the *gray intervals*, i.e. the approximation error, is small. Finally, the third rule guarantees that a candidate test can be carried out efficiently. Thus, Rule 2 and Rule 3 are responsible for low $cost_{refine}$. A good join response behavior results from an optimum trade-off between these grouping rules.

Filter cost. The $cost_{filter}(I_{gray}, T)$ can be computed by the expected number of intersection tests required to perform the join between I_{gray} and the join partners, which is equal to the overall number $N_{gray}(T)$ of *gray intervals* $I_{gray}(T)$. Therefore, we penalize each intersection test by the cost c_f which are required to access the *gray intervals* $I_{gray}(T)$ and evaluate the join predicate for each pair $(H(I_{gray}), H(I_{gray}(T)))$:

$$cost_{filter}(I_{gray}, T) = N_{gray}(T) \cdot c_f,$$

where $N_{voxel}(T)$ (number of voxels) $\geq N_{gray}(T) \geq N_{object}(T)$ (number of objects) holds for the join-partner relation. The value of parameter c_f depends on the used system.

Refinement cost. The cost of the refinement step $cost_{refine}$ is determined by the selectivity of the filter step. For each candidate pair resulting from the filter step, we have to retrieve the exact geometry $B(I_{gray})$ in order to verify the intersection predicate. Consequently, our cost-based grouping algorithm is based on the following two parameters:

- Selectivity σ_{filter} of the filter step.
- Evaluation cost $cost_{eval}$ of the exact geometries.

The *refinement cost* of a join related to a *gray interval* I_{gray} can be computed as follows:

$$cost_{refine}(I_{gray}, T) = N_{gray}(T) \cdot \sigma_{filter}(I_{gray}, T) \cdot cost_{eval}(I_{gray}).$$

In the following subsections, we show how we can estimate the selectivity of the filter step σ_{filter} and the evaluation cost $cost_{eval}$.

3.3.1. Selectivity estimation. We use simple statistics of the join-partner relation T to estimate the selectivity $\sigma_{filter}(I_{gray}, T)$. In [10], it was shown that using quantiles ('equi-count histograms') is more suitable for estimating the selectivity than using histograms ('equi-width histograms'). The runtime required for the histogram computation is increased by the cost of barrier-crossings between the declarative environment of the SQL layer and our stored procedure. Fortunately, most ORDBMSs comprise efficient built-in functions to compute single-column statistics, particularly for cost-based query optimization. Available optimizer statistics are accessible to the user by the relational data dictionary. The basic idea of our quantile-based selectivity estimation is to exploit these built-in index statistics rather than to add and maintain user-defined histograms.

We start with the definition of a quantile vector, the typical statistic type supported by relational database kernels.

Definition 3 *(Quantile Vector)*.
Let $(M, \leq)$ be a totally ordered multi-set. Without loss of generality, let $M = \{m_1, m_2, \ldots, m_N\}$ with $m_j \leq m_{j+1}$, $1 \leq j < N$. Then, $QV(M, \nu) = (q_0, \ldots, q_\nu) \in M^\nu$ is called a *quantile vector* for M, given a resolution $\nu \in IN$, iff the following conditions hold:

(*i*) $q_0 = m_1$

(*ii*) $\forall i \in 1, \ldots, \nu: \exists j \in 1, ..., N: \; q_i = m_j \wedge \frac{j-1}{N} < \frac{i}{\nu} \leq \frac{j}{N}$

The multi-set M of our quantile vector $(q_0, \ldots, q_\nu)$ is formed by the z-value attribute of the domain values of the join-partner relation.

The selectivity $\sigma_{filter}(I_{gray},T)$ related to a gray interval I_{gray} can be determined by applying the quantile vector $QV(T,v)$ of the z-values of the join-partner relation T. In the following formula, v denotes the resolution of the quantile vector and *overlap*() returns the intersection length of two intersecting intervals:.

$$\sigma_{filter}(I_{gray}, T) \approx \sum_{i=1}^{v} \left(\frac{overlap(H(I_{gray}), (q_{i-1}, q_i))}{q_i - q_{i-1}} \right) / v$$

3.3.2. *BLOB*-Evaluation cost. For the computation of the *evaluation cost* we have to consider the I/O cost required to retrieve the *BLOB* from the secondary storage and the CPU cost related to the evaluation of the *BLOB*. These cost heavily depend on how we organize $B(I_{gray})$ within our *BLOB*, i.e. they depend on the used compression algorithm. For each compression algorithm we provide statistics, i.e. an empirically derived look-up table *LUT* (cf. Figure 5), by means of which we can estimate the I/O cost and CPU cost. Roughly speaking, the evaluation cost $cost_{eval}(I_{gray}, LUT)$ depends on the length of our *gray interval* $L(I_{gray})$ and on the used packer.

3.3.3. Join cost. To sum up the join cost $cost_{join}(I_{gray})$ related to a *gray interval* I_{gray} and a join-partner relation T can be expressed as follows:

$$cost_{join}(I_{gray},T) =$$
$$N_{gray}(T) \cdot (c_f + \sigma_{filter}(I_{gray},T) \cdot cost_{eval}(I_{gray},LUT))$$

The *filter selectivity* and *BLOB-evaluation cost* are computed as described in Section 3.3.1 and 3.3.2. For the computation of the filter cost, we propose to empirically derive the value of c_f. Let us note that the inequality '$cost_{gray} > cost_{dec}$' in Figure 5 is independent of $N_{gray}(T)$, and thus $N_{gray}(T)$ is not required during the grouping algorithm.

3.3.4. Grouping algorithm. Orenstein [16] 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 approximation error of all gray intervals should be small". For fulfilling both rules, we introduce the following top-down grouping algorithm for *gray intervals*, called *GroupCon* (cf. Figure 5). *GroupCon* is a recursive algorithm which starts with a *gray interval* I_{gray} initially covering the complete object. For reasons of efficient computation we use the following simple heuristics: In each step of our algorithm, we look for the maximum gap g within the actual *gray interval*. We carry out the split at this gap, if the estimated join cost caused by the decomposed intervals is smaller than the estimated cost caused by our input interval I_{gray}. The expected join cost $cost_{join}(I_{gray},T)$ can be computed as described above. Data compressors which have a shallow *LUT* curve, e.g. PACKER 2 in Figure 5, result in an early stop of the *GroupCon* algorithm generating a small number of *gray intervals*.

QV: quantile vector
LUT:look-up table with packer specific cost
GroupCon (I_{gray}, *LUT*, *QV*(*T*), *T*) **{**
  left gray_interval;
  right gray_interval;
  interval_pair tupel(left gray_interval, right gray_interval)
  interval_pair **:=** split_at_maximum_gap(I_{gray});
  left **:=** *interval_pair.left*;
  right **:=** *interval_pair.right*;
  $cost_{gray}$ **:=** $cost_{join}(I_{gray},T)$;
  $cost_{dec}$ **:=** $cost_{join}(left,T) + cost_{join}(right,T)$;
  if $cost_{gray} > cost_{dec}$ **then**
    GroupCon (*left*,*LUT*,*QV*(*T*),*T*);
    GroupCon (*right*,*LUT*,*QV*(*T*),*T*);
  else report (I_{gray});
}

look-up table *LUT*

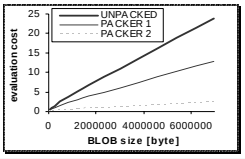


Figure 5: Grouping algorithm *GroupCon* based on a look-up table *LUT* and a quantil vector *QV*

4. Accelerated relational join processing

In contrast to the last section, where we focused on building the object approximations and organizing them within the database, in this section we turn our attention to processing the join. We first present our new join algorithm in Section 4.1, using the techniques presented in Section 3. In Section 4.2, we show how we can express the join procedure on top of the SQL engine. Furthermore, we introduce useful optimizations for the evaluation of the intersect predicate in Section 4.3.

4.1. Join algorithm.

Our two phase join algorithm is shown in Figure 6. Thereby we build the *gray intervals* (function *decompose*())

```
R   table(id, z-val);      //objects of relation R
S   table(id, z-val);      //objects of relation S
S'  table(Gray_Intervals);
join(R,S){
  for each object obj_S in S do {                    } preprocessing phase
    obj_S' := decompose(obj_S);                      }
    store (obj_S') in S'; }                          }
  result_set := ∅;
  for each object obj_R in R do {                    } join phase
    obj_R' := decompose(obj_R);                      }
    for each object obj_S' in S' do                  }
     if intersect(obj_R', obj_S') then               }
       result_set := result_set ∪ (obj_R'.id, obj_S'.id);}}
```

Figure 6: *Nested-Loop* join algorithm

by means of the cost-based grouping algorithm presented in Section 3.3.3. In the following, we assume that we have to join relation *R* with relation *S* containing complex spatial objects.

Preprocessing phase. For each object obj_S in relation *S* we apply the function *decompose*(obj_S), which builds the *gray intervals* according to the grouping algorithm *GroupCon* (cf. Figure 5). This grouping algorithm takes the statistics of the data distribution with respect to relation *R* into account. Finally, the *gray intervals* of each object are materialized in relation *S'* following the NF^2 schema *GrayIntervals* (cf. Figure 3).

In the following nested-loop join, we assume that the objects obj_R of relation *R* will be accessed only once. Thus, there is no need to materialize the gray intervals of obj_R in the database as done for the objects in relation *S*, or *S'*. Assuming that one object completely fits in memory, its *gray intervals* can be built on-the-fly during the join phase.

Join phase. The *join phase* is performed in a *nested-loop* fashion. For each object, we perform the function *decompose*(obj_R) in the outer loop in order to build the *gray intervals* of object obj_R. This time, we apply the data distribution statistics of relation *S*. In the inner loop, we test each object obj_S for intersection with object obj_R calling the boolean function *intersect*().

The function *intersect*($obj_{R'}$, $obj_{S'}$) checks whether two objects $obj_{R'}$ and $obj_{S'}$ intersect. They intersect, iff there is at least one *gray-interval* pair ($obj_{R'}.I_{gray}$, $obj_{S'}.I_{gray}$) which intersects. We assume that the rows of both nested tables $obj_{R'}.data$ and $obj_{S'}.data$ are sequentially accessed and that the gray intervals are ordered with respect to their hulls. Both nested tables are processed in parallel, thus we need to access each row only once. As soon as an intersection is detected, the remaining tests can be skipped and the value "*true*" is issued. The intersection test of a *gray-interval* pair is performed in two steps: In the first step (*filter step*) the pair is tested with respect to their hulls. If the result of the filter step is positive, i.e. the hulls intersect, a subsequent

```
SELECT R'.id, S'.id FROM
(   GrayIntervals S',
    ( SELECT R.id AS id, decompose(R.z_val) AS data
      FROM R
      GROUP BY R.id ) R'
WHERE intersect(R'.data, S'.data) = true // filter/refinement
```

Figure 7: SQL statement for spatial-intersection join

refinement step verifies the intersection with respect to the exact geometric object representations. Before testing the two byte sequences for intersection, we have to load $B(obj_{S'}.I_{gray})$ from disk and decompress it.

As already mentioned in Section 3.2.2, 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 only the objects of the inner relation *S'* are in a compressed form, whereas the byte sequence $B(obj_{R'}.I_{gray})$ does not affect the I/O cost. Furthermore, a fast decompression algorithm is required to evaluate the *BLOB* quickly.

In the next section, we show how we can easily express the intersection join query on top of the SQL engine.

4.2. The *spatial-intersection join* SQL statement

Most ORDBMSs, including Oracle, IBM DB2 or Informix IDS/UDO, provide extensibility interfaces in order to enable database developers to seamlessly integrate custom object types and predicates within the declarative DDL and DML. These interfaces form a necessary prerequisite for the seamless embedding of user-defined spatial objects, functions and aggregates into off-the-shelf ORDBMSs. On this basis, we define the intersection join query which is expressed on top of the SQL engine as shown in Figure 7.

The input of this SQL query is the relation *R* and the auxiliary relation *S'* derived from the preprocessing step. In the subquery, which results in the new relation *R'*, we use the function *decompose*() which is a user-defined aggregate function as provided in the SQL:1999 standard. This function decomposes each object of *R* into a set of *gray intervals*. The function *intersect*() is implemented as stored procedure and behaves as described in Section 4.1.

4.3. Optimizations

For the *intersect* predicate, it suffices to find a single intersecting interval pair in order to report the join-pair. Obviously, it is desirable to detect such intersecting pairs as early as possible in order to avoid unnecessary *refinement tests*. In this section, we present an optimization aiming at this goal. We introduce a *fast second filter step* which tries to determine intersecting pairs without examining the *BLOB*s. This test is entirely based on aggregated informa-

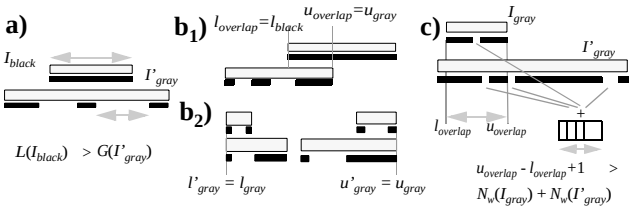


Figure 8: Fast intersection tests on gray intervals

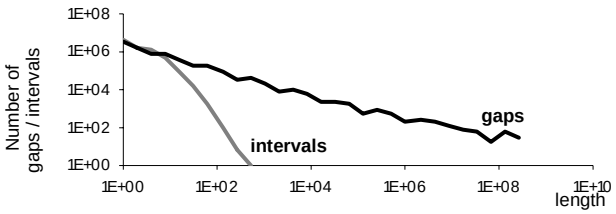


Figure 9: Interval and gap histograms (*CAR*)

tion of the *gray intervals*. The following optimization can easily be integrated into the function *intersection*(). If the *fast second filter step* determines an intersecting *gray interval* pair, all remaining candidate tests can obviously be skipped. Thus this filter step acts as an additional filter between the *first filter step* and the *refinement step*.

4.3.1. Fast intersection tests. Let us first mention that a *gray interval* with maximum density is called a *black interval*. Furthermore, we speak of "overlapping" intervals, if the hulls of the intervals intersect. We will now discuss what *gray intervals* have to look like so that we can decide whether two overlapping intervals actually intersect each other or not without accessing their *BLOB*s. If any of the following five conditions holds, then two *gray intervals* intersect:

- If two *black intervals* overlap, they necessarily intersect as well.
- If a *black interval* is longer than the maximum gap between two *black intervals* contained in I_{gray}, then the two intervals intersect (cf. Figure 8a).
- If a *black interval* overlaps the start or end of a *gray interval*, then the intervals intersect. This is due to the fact that any *gray interval* ends and starts with a *black interval* (cf. Figure $8b_1$).
- If *gray intervals* start or end at the same point, then the intervals intersect. This is due to the fact that any *gray interval* ends and starts with a *black interval* (cf. Figure $8b_2$).
- If the sum of the number of the white voxels of two overlapping *gray intervals* is smaller than the length of the overlapping area, then the two intervals necessarily intersect. (cf. Figure 8c). This test is similar to the false area test in [3].

Let us note that we carry out this *fast-intersection* test for all overlapping *gray intervals* before testing the exact geometry for any *gray interval*. If one of these *fast-intersection-tests* yields *true*, the intersection routine returns *true*, without testing any data stored in the *BLOB*s. If none of these *fast-intersection-tests* yields *true*, it is beneficial to order the gray intervals of the objects by descending density values $D(I_{gray})$ before carrying out the expensive *BLOB*-intersection test. Thus, the intervals having the highest density are tested first, which increases the probability for an early intersection detection.

5. Experimental evaluation

In this section, we evaluate the performance of our approach with a special emphasis on different grouping algorithms *GRP* in combination with various data compression techniques *DC*. We used the following data compressors: no compression (*NOOPT*), *BZIP2* approach [4] and the *QSDC* approach. Furthermore, we grouped object voxels into *gray intervals* following two grouping algorithms, called *MAXGAP* and *GroupCon*.

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 3, namely a small number of *gray intervals* and a small approximation error of each of these intervals. A *one-value* approximation is achieved by setting the MaxGap parameter to infinite.

GroupCon. We grouped the voxels according to our cost-based grouping algorithm *GroupCon* (cf. Section 3.3.3), where we used the statistics of Section 3.3.1 and a look-up table for each packer. We set the resolution of the quantile vector to 100 quantiles. The look-up table was created by experimentally determining the average cost for evaluating a *gray interval* I_{gray}, dependent on the length of its byte sequence. Let us note, that the grouping based on *MaxGap*(*DC*) does not depend on *DC*, whereas *GroupCon*(*DC*) takes the actual data compressor *DC* into account for performing the grouping.

The *refinement-step* evaluation of the *intersect*() routine 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 sets. The tests are based on two test data sets *CAR* (3D CAD data) and *SEQUOIA* (subset of 2D GIS data representing woodlands derived from the *SEQUOIA 2000* benchmark [19]). The first test data set was provided by our industrial partner, a German car manufacturer, in form of high resolution voxelized three-dimensional CAD parts. The properties of both data sets are depicted in Table 1.

Table 1: Test Data Sets

Dataset	# voxels	# objects	size of Data Space
CAR	14 million	200	2^{33} cells
SEQUOIA	32 million	1100	2^{34} cells

In both cases, the z-curve was used as a space filling curve to enumerate the pixels/voxels. Figure 9 depicts the interval and gap histograms for the *CAR* test data set. This characteristic can also be observed for *SEQUOIA*. Both test data sets consist of many short black intervals and short gaps and only a few longer ones.

5.1. Effectivity of the Compression Techniques

First we present the compression effectivity of our preferred packer by looking at the storage requirements of the materialized *gray intervals* of the *CAR* dataset. Figure 10 shows the different storage requirements of the *BLOB*s with respect to the different data compression techniques. For high *MAXGAP* values the *BZIP2* approach yields very high compression rates, for the one-value approximation even more than 1:1000. On the other hand, due to an enormous overhead, the *BZIP2* approach occupies even more secondary storage space than *NOOPT* for small *MAXGAP* values. Contrary, the *QSDC* approach yields good results over the full range of the *MAXGAP* parameter. Using the QSDC compression technique, we achieve low I/O cost for the *BLOB*s which drastically enhance the efficiency of the *refinement step* of the join process.

5.2. Efficiency of the Join Process

In this section, we want to turn our attention to the efficiency of the join process. The figures presented in this paragraph depict the performance of the spatial join queries. We have performed intersection joins over two relations, each containing approximately a half of the parts from the *CAR* dataset. We took care that the data of both relations have similar characterizations with respect to the object size and distribution. Similarly, the intersection join is performed on parts of the *SEQUOIA* data set which is divided into two relations, consisting of deciduous-forest and mixed-forest areas.

In Figure 11 it is shown in which way the response time for the intersection join query, including the preprocessing

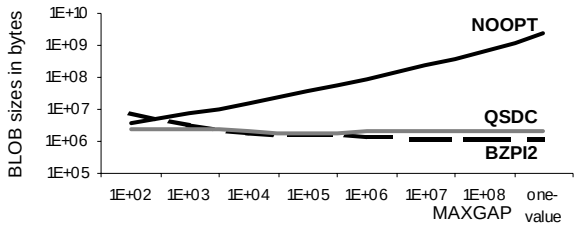


Figure 10: Storage requirements for the *BLOB* (*CAR*)

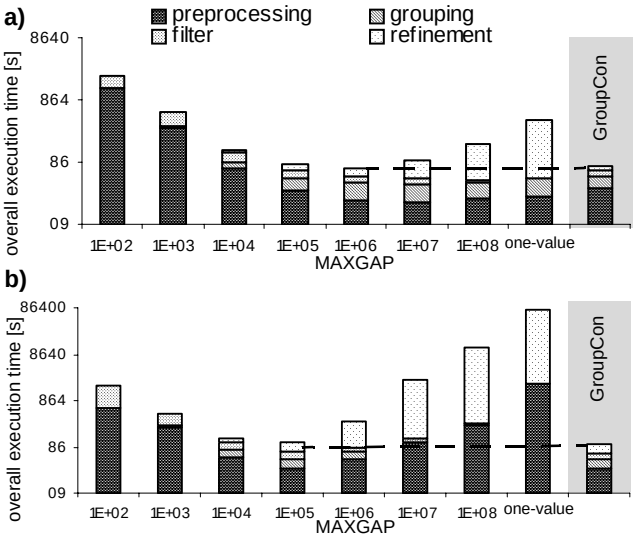


Figure 11: *GRP*(*DC*) evaluated for intersection joins on the *CAR* datasets
a) QSDC compression b) NOOPT (no compression)

step, depends on the *MAXGAP* parameter using the *QSDC* compression (cf. Figure 11a) and no compression (cf. Figure 11b). The figures depict the overall contributions of the *preprocessing phase* (cf. Figure 2a), of the *on-the-fly grouping* (cf. Figure 2b) and of the *filter* and *refinement step*. If we use small *MAXGAP* parameters, we need a lot of time for the filter step whereas the refinement step, which is influenced by the BLOB sizes, is relatively cheap. On the other hand, for high *MAXGAP* values we can see that the *refinement step* is very expensive in contrast to the *filter step* which shows very little contribution. Due to the fact, that the performance mainly depends on the I/O cost, the *preprocessing step* shows a similar performance behavior as the pure join. We can observe that for both compression cases the *GroupCon* approach exceeds the best MAXGAP approach with respect to both compression variants. The marginally higher preprocessing cost of the GroupCon algorithm result from the computation of the cost-estimations required for the decomposition.

Figure 12 illustrates how the overall join run-time depends on the different grouping techniques for both data-sets, *CAR* (cf. Figure 12a) and *SEQUOIA* (cf. Figure 12b). For packed data the optimum *MAXGAP* value is higher than for unpacked data, i.e. MAXGAP = 10^5 for *NOOPT* and MAXGAP = 10^6 for *BZIP2* and *QSDC*. The *GroupCon* algorithm produces for both data sets object decompositions which yield almost optimum join response time for varying compression techniques. It adapts to the optimum *MAXGAP* parameter for varying compression techniques, by allowing greater gaps for packed data, i.e the number of generated *gray intervals* is smaller in the case of packed data.

To sum up, the *GroupCon* algorithm produces object decompositions which yield the optimal trade-off between

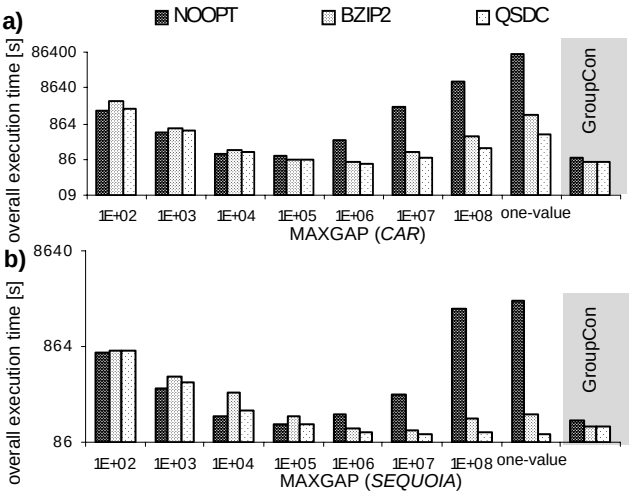


Figure 12: Overall join performance for different packers (a) *CAR* dataset (b)*SEQUOIA* dataset

the *filter* and *refinement cost* for both high-resolution data sets.

6. Conclusion

In this paper, we introduced a new approach for efficient processing of spatial intersection joins over high-resolution objects. In our approach, it is assumed that there is no spatial index available and that the join is performed in nested loop fashion. The join procedure is based on fast filter steps performed on object approximations and an expensive refinement step. We introduced *gray intervals* as an object approximation concept which is based on grouping large voxel sets, which are high-resolution object representations, into few container objects in a preceding preprocessing step. It is shown how they can efficiently be stored by means of data compression techniques within ORDBMSs. In particular, we deployed a quick spatial data compressor *QSDC*, in order to emphasize those packer characteristics which are important for efficient join processing, namely *good compression ratio* for low I/O cost and *high unpack speed* for low evaluation cost. The core of our approach is a cost-based decompositioning algorithm for complex spatial objects, called *GroupCon*. It takes cost of the filter and refinement step into account. The refinement cost are based on selectivity estimations of the *filter step* which are derived from statistics and estimation of the decompression cost of the *gray intervals*. The decomposition algorithm demonstrates good performance for different compression techniques. We showed that our new approach, i.e. the combination of *GroupCon* and *QSDC*, accelerates the spatial join by more than one order of magnitude compared to the uncompressed one-value approximation.

In our future work, we want to extend the application ranges of our new approach from the area of digital mock-up to real-time virtual reality applications. Furthermore, we plan to deploy spatial access methods in order to further improve the join efficiency.

7. References

[1] Arge L., Procopiuc O., Ramaswamy S., Suel T., Vitter J.S.: *Scalable Sweeping-Based Spatial Join*, In Proc. of the VLDB Conference, 1998, 570-581.

[2] Brinkhoff T., Kriegel H.P., Seeger B.: *Efficient Processing of Spatial Joins Using R-trees*, In Proc. of the ACM SIGMOD Conference, 1993, 237-246.

[3] Brinkhoff T., Kriegel H.-P., Schneider R., Seeger B.: *Multi-Step Processing of Spatial Joins*, In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1994, 197-208.

[4] Burrows M., Wheeler D. J.: *A Block-sorting Lossless Data Compression Algorithm*, Digital Systems Research Center Research Report 124, 1994.

[5] Cannane A., Williams H.: *A Compression Scheme for Large Databases*, Australasian Database Conference (ADC), 6-11, 2000.

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

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

[8] Gaede V.: *Optimal Redundancy in Spatial Database Systems*, In Proc. 4th Int. Symp. on Large Spatial Databases, 1995, 96-116.

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

[10] 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, 131-141.

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

[12] Kunath P.: *Compression of CAD-data*, Diploma thesis, University of Munich.

[13] Lo M-L., Ravishankar C.V.: *Spatial Joins Using Seeded Trees*, In Proc. of the ACM SIGMOD Conference, 1994, 209-220.

[14] Lo M-L., Ravishankar C.V.: *Spatial Hash-Joins*, In Proc. of the ACM SIGMOD Conference, 1996, 247-258.

[15] 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.

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

[17] Orenstein J. A.: *Spatial Query Processing in an Object-Oriented Database System*, In Proc. of the ACM SIGMOD Conference, 1986, 326-336.

[18] Patel J.M., DeWitt D.J.: *Partition Based Spatial-Merge Join*, In Proc. of the ACM SIGMOD Conference, 1996, 259-270.

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

[20] 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, 377-396.