The LSD tree: spatial access to multidimensional point and non-point objects *

Andreas Henrich
FernUniversität Hagen
5800 Hagen
West Germany

Hans-Werner Six
FernUniversität Hagen
5800 Hagen
West Germany

Peter Widmayer
Universität Freiburg
7800 Freiburg
West Germany

Abstract

We propose the Local Split Decision tree (LSD tree, for short), a data structure supporting efficient spatial access to geometric objects. Its main advantages over other structures are that it performs well for all reasonable data distributions, cover quotients (which measure the overlapping of the data objects), and bucket capacities, and that it maintains multidimensional points as well as arbitrary geometric objects. These properties demonstrated by an extensive performance evaluation make the LSD tree extremely suitable for the implementation of spatial access paths in geometric databases. The paging algorithm for the binary tree directory is interesting in its own right because a practical solution for the problem of how to page a (multidimensional) binary tree without access path degeneration is presented.

1. Introduction

In non-standard applications such as cartography, CAD and robotics, Database Management Systems have to organize large sets of multidimensional geometric objects on secondary storage such that these objects can quickly be retrieved according to their spatial locations. Typical spatial queries are the retrieval of an object by its coordinates (exact match) and range queries where all objects geometrically intersecting the query region are selected for further processing or presentation on a screen. Since the set of objects varies over time, insertions and deletions have to be performed as well.

Data structures, which efficiently support spatial access to geometric objects, usually divide the data space into cells and store all objects located in a cell in an associated data bucket. As far as multidimensional points are concerned, various efficient data structures have been proposed (see e.g. [Free87], [HSW88a], [HSW88b], [KS86], [KS88], [KW85], [NHS84], [Otoo86], [Rob81]). In typical applications, however, most objects are arbitrary geometric, i.e. non-point, objects. In many situations, it has proven to be useful to represent non-point objects by their (minimal) bounding boxes, serving as simple geometric keys. We therefore concentrate on multidimensional intervals, as far as non-point objects are concerned.

Proceedings of the Fifteenth International
Conference on Very Large Data Bases

An obvious approach for storing intervals uses data structures for points. Here, intervals need not be entirely contained in a cell, but may instead intersect several cells. Hence, in order to perform range queries efficiently, information about an interval must be stored in each bucket, whose corresponding cell intersects the interval. Using this so-called *clipping technique*, the space requirements increase substantially due to the redundant information.

The clipping approach is based on data structures which partition the data space into pairwise disjoint cells. To avoid clipping problems, in R-Trees ([Gut84], [FSR87]), respectively multilayer grid files [SW88], the data space is divided into *overlapping cells*, such that all, respectively most, intervals are entirely contained in a cell, i.e. need not be clipped. Unfortunately, the R-Tree suffers from a poor exact match performance and often from inefficient range queries because cells may overlap considerably in a dynamic setting. In the multilayer grid file with each layer a grid file is associated inducing a directory overhead which deteriorates the efficiency of operations concerning few objects.*

In the so-called *transformation technique* ([Hin85], [SK88]), k-dimensional intervals are interpreted as points in a 2k-dimensional space, in order to use point data structures in a standard way. For instance, a 1-dimensional interval [a, b] may be interpreted as point (a, b). Since $a \leq b$ holds, the image space is a triangle. The main drawbacks of this approach are that the point distribution in the triangular image space is extremely skew and that a (bounded) range query on intervals becomes a partly unbounded range query on image points.

From a wide spectrum of performance tests we have got the experience that the efficiency of spatial data structures depends at least on the object distribution, the cover quotient defined as the sum of all object areas divided by the area of the data space, and the bucket capacity, i.e. the maximal number of objects in a bucket. For an increasing cover quotient as well as for small bucket capacities, clipping and overlapping cell techniques deteriorate substantially. On the other hand, all non-tree structures (see e.g. [HSW88a], [KS86], [KS88], [NHS84]) degenerate for skew object distributions.

In this paper, we propose a data structure supporting spatial access to k-dimensional points as well as k-dimensional intervals. The access to intervals is based on the transformation technique. A sophisticated directory tree together with a refined splitting technique eliminates the pre-

* This work has been supported by DFG grants Si 374/1 and Wi 810/2.

Amsterdam, 1989

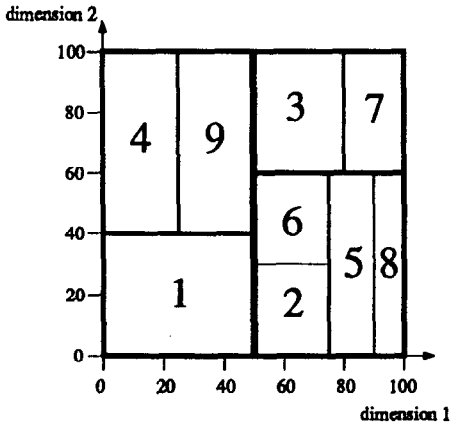

Figure 2.1: Possible partition of the data space for an LSD tree

vious drawbacks of this technique. The performance evaluation convincingly demonstrates that the new structure is well qualified for maintaining large sets of geometric objects. However, the main advantage of the new structure is not only the efficient spatial access but also its robustness. By robustness we mean that the new structure behaves well for all reasonable data distributions, cover quotients and bucket capacities.

Section 2 explains the new data structure for point objects while in section 3 the generalization to non-point objects is provided. In section 4 a performance evaluation of the new structure and a comparison with the multilayer grid file is presented. Section 5 concludes the paper.

2. The LSD tree for points

2.1 Basic ideas and properties

Like most structures, the new structure partitions the data space into pairwise disjoint cells with associated buckets of fixed size. In contrast to the grid file [NHS84], however, it is not grid oriented, i.e. all cell boundaries may occur at arbitrary positions. The free choice of split positions is the basis of the graceful adaptation to arbitrary skew object distributions. Since a new split position can be chosen locally optimal, i.e. optimal with respect only to the cell to be split and independent from other existing cell boundaries, we call the new structure **Local Split Decision tree** (LSD tree, for short). Figure 2.1 shows a possible partition of a 2-dimensional data space for an LSD tree.

The LSD directory maintaining the flexible data space partition is a binary tree similar to a k-d tree [Ben75]. Each node of this tree represents one split decision by storing the split dimension and the split position. Figure 2.2 illustrates the LSD tree associated with the data space partition of Figure 2.1.

It should be obvious that the directory provides the freedom for using the split strategy best suitable for the actual application. This is an important advantage over other structures (see e.g. [Free87], [HSW88a], [KS86], [KS88], [NHS84]) where split decisions are more or less influenced by previous split decisions. Furthermore, the size of the

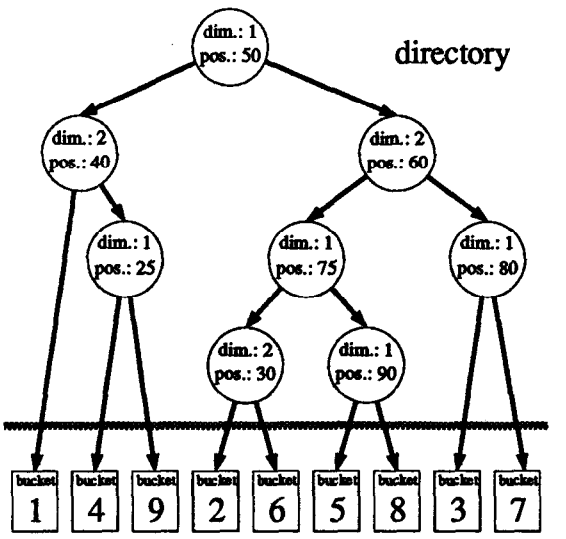

Figure 2.2: LSD tree associated with the data space partition of Figure 2.1

directory is directly related to the number of buckets, i.e. for n buckets the directory contains n-1 nodes. This is in contrast to the grid file where several entries in the directory may point to the same bucket.

Besides the advantages of the LSD tree directory there are some drawbacks which are typical for multidimensional binary tree structures:

1. A multidimensional binary tree may become unbalanced, i.e. may contain long paths with almost no branches, and
2. no suitable method for paging a multidimensional binary tree is known. (The interesting paging technique presented in [LZL88] is suitable only for the one-dimensional case.)

We overcome these problems by introducing a paging algorithm which preserves the following **external balancing property**:

The number of external directory pages which are traversed on any two paths from the directory root to a bucket differs by at most 1.

When geometric objects are inserted into an initially empty LSD tree the directory grows up to a size when it cannot be kept in the dedicated part of the main memory any longer. Then the paging algorithm determines a subtree to be paged on secondary storage such that the external balancing property is preserved. If the subtree consists of n_s nodes, the main memory is then able to receive additional n_s nodes until a further invocation of the paging algorithm must take place.

Figure 2.3 shows the overall structure of the LSD tree.

2.2 A closer look

In this section, we discuss the LSD tree in more detail by explaining the insertion of a new geometric object into the structure.

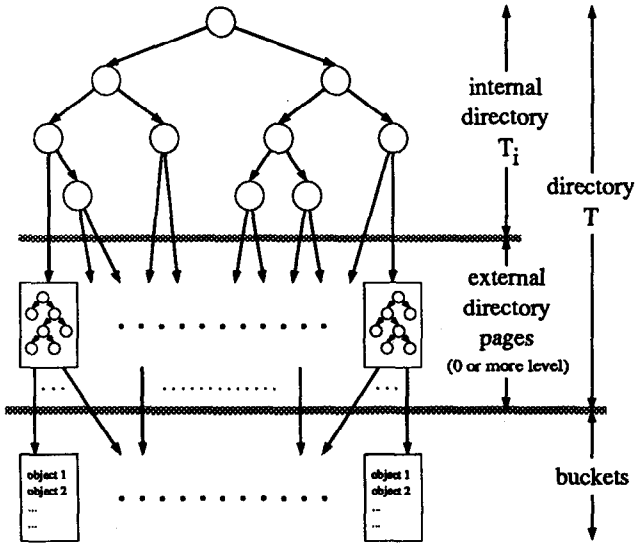

Figure 2.3: Overall structure of the LSD tree

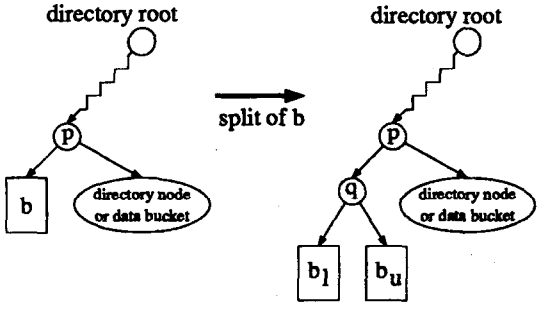

Figure 2.4: Effect of a bucket split

The search for the bucket b which will receive the new object is guided by the directory as in k-d trees. If b does not overflow, the insertion is finished, otherwise the **bucket split algorithm** creates two new buckets b_l and b_u from b according to a *split strategy* we describe afterwards. In case of a bucket split the pointer in the directory referencing b is changed to a pointer referencing a new directory node q representing the split decision concerning b, i.e. the new node q is inserted into the directory by calling the *directory insertion algorithm* explained later. The new node q points to the new buckets b_l and b_u. Figure 2.4 depicts the effect of a bucket split.

We distinguish between two inherently distinct types of **split strategies**:

1. *Data dependent split strategies*
   These strategies depend only on the objects stored in the bucket to be split. A typical example for such a strategy is to choose for the split position the average of all object coordinates with respect to a certain dimension.
2. *Distribution dependent split strategies*
   These strategies choose the split dimension and the split position independently of the actual objects stored in the bucket to be split. A typical example for such a strategy is to split a cell into two cells of equal areas. Note that this "halving split strategy" relies on the assumption of a uniform distribution of the objects.

Since the LSD directory is a binary tree, any type of split strategy can be implemented in an easy and efficient manner. Note that data dependent split strategies cannot be realized by data structures based on hashing (see e.g. [HSW88a], [KS86], [KS88], [NHS84]).

We now turn our attention to the directory of the LSD tree. As already mentioned in the previous section, if the number of nodes in the directory T exceeds the maximal possible number of internal nodes, a subtree of T is written onto secondary storage, i.e. stored in a directory page. In such a directory page a subtree is organized as a sequential heap of fixed height h_p. Hence, whenever the height of the associated subtree exceeds h_p after an additional insertion, a directory page split has to be performed. The **directory page split algorithm** is simple: the left and right subtree of the root are stored in two distinct directory pages and the root is inserted into the directory T by calling the *directory insertion algorithm.*

We are now in a position to describe how a new node q, resulting from a bucket or a directory page split, is inserted into the directory T by the **directory insertion algorithm.** The heart of this algorithm is the paging algorithm we explain afterwards. In the following, we assume that the main memory capacity reserved for the directory T is n_i. The directory insertion algorithm assures that the internal prefix tree T_i of the directory T contains at most n_i-1 nodes. Two cases may occur:

case 1: The father p of the new node q is an internal node.

- **case 1.1:** The number of internal nodes is less than n_i-1. Then insert q into T_i and finish.
- **case 1.2:** The number of internal nodes is equal to n_i-1. Then insert q into T_i, call the *paging algorithm* for T, and finish. Note that after the execution of the paging algorithm the number of internal nodes is at most n_i-1.

case 2: The father p of the new node q is a node in a subtree T_p of T stored in an external directory page.

- **case 2.1:** After the insertion of q the height of T_p is at most h_p; then finish.
- **case 2.2:** After the insertion of q the height of T_p is greater than h_p. Then call the *directory page split algorithm* for T_p and finish.

The **paging algorithm** is called when after an insertion of an additional node the size of the internal prefix tree T_i reaches the maximal possible number n_i of internal nodes. The algorithm searches for a subtree T_s in T_i such that paging T_s preserves the external balancing property defined in section 2.1. This property is preserved if T_s is a *paging candidate*, i.e. T_s fulfills the following properties:

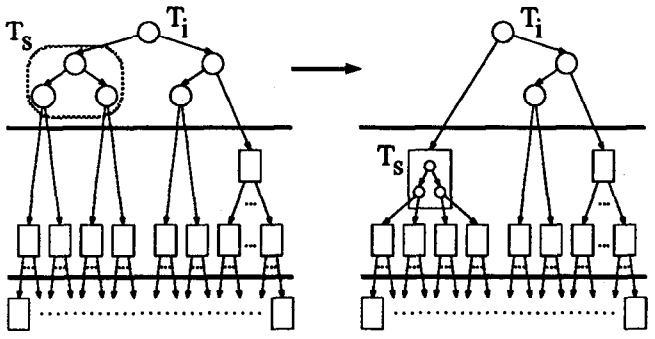

Figure 2.5: Directory before and after paging subtree T_s

1. Any path from the root of T_s down to a bucket contains the minimal number of external directory pages (of all paths in the directory T).
2. The height of T_s is at most h_p.

Figure 2.5 shows a directory before and after paging the subtree T_s.

If more than one paging candidate occurs in T_i, the paging algorithm chooses a candidate with the maximal possible number of nodes.

In order to direct the search for a paging candidate in T_i the following numbers are attached to each node v in T_i:

$nep_{min}(v)$, resp. $nep_{max}(v)$,: the minimal, resp. maximal, number of external directory pages occurring on any path in T containing v.

s(v): the number of nodes of the biggest paging candidate which can be reached from v.

h(v): The height of the subtree with root v in T_i.

The paging algorithm moves down the internal directory T_i branching at each node w on the search path according to the following criteria:

1. If nep_{min}(left son of w) $\neq$ nep_{min}(right son of w), continue with the son with lower nep_{min}.
2. If nep_{min}(left son of w) = nep_{min}(right son of w), continue with the son with greater s.

The root r of a paging candidate T_s is determined if

1. $h(r) \leq h_p$ and
2. $nep_{min}(r) = nep_{max}(r)$.

The second condition assures that after paging T_s the external balancing property is preserved for T.

It should be clear that after the insertion of a new node q into the internal directory T_i, resp. the paging of a subtree T_s of T_i, the numbers nep_{min}, nep_{max}, s and h must be updated for each node w on the path P from the root of T_i to the father of q, resp. to the father of the root of T_s (now stored in an external directory page). These numbers can easily be recomputed from the existing numbers of the nodes (and their direct sons) on the path P.

2.3 The operations

2.3.1 Exact match

In an exact match operation the directory is traversed until the corresponding bucket is determined. The bucket is scanned until the object searched for is located or the search ends unsuccessfully.

2.3.2 Insertion

The insertion algorithm has been explained in detail in the previous section.

2.3.3 Deletion

The deletion of a node in the directory T basically works inversely to the insertion of a node. Due to space limitations we cannot discuss this topic in more detail.

2.3.4 Range query

In a range query all points located in the query region are reported. According to Fredman [Fred80], a query region may be a rectangle (*orthogonal range query*), a circle (*circular range query*) or a polygon (*polygonal range query*). In the following, we restrict the discussion to orthogonal range queries, because the algorithm is the same for all query types, except for the procedures evaluating whether a data region is enclosed by, intersected by, or disjoint from the query region. But these are details left to the implementation level.

In order to report all points located in the query region Q we have to traverse the LSD tree to determine all buckets whose associated cells intersect Q. The query algorithm moves down the LSD tree branching at each directory node w according to the following criteria: (Here D(w) denotes the data region which is the union of all data cells whose corresponding buckets can be reached from w.)

1. If $Q \cap D$(right son of w) $= \emptyset$, continue with the left son of w.
2. If $Q \cap D$(left son of w) $= \emptyset$, continue with the right son of w.
3. Otherwise continue with both sons of w.

Note that $Q \cap D(w) \neq \emptyset$ is the invariant condition of the loop of the query algorithm. Hence, in 1., resp. 2., $Q \cap D$(left son of w) $\neq \emptyset$, resp. $Q \cap D$(right son of w) $\neq \emptyset$, holds.

3. The LSD tree for non-point objects

We explain the non-point situation for k-dimensional intervals which serve as bounding boxes for arbitrary geometric objects in many applications. We restrict the discussion to the 2-dimensional situation, i.e. to rectangles in the plane, because a generalization to higher dimensions is straightforward.

To store a set of rectangles in the LSD tree we use the *transformation technique* ([Hin85], [SK88]), i.e. 2-dimensional rectangles are stored as 4-dimensional points. We choose the simple corner representation [SK88] which considers for each of the two dimensions the lower and upper bounds of the rectangles to be distinct dimensions.

The idea is simple but several severe problems arise from this approach. First, there is a strong correlation between upper and lower bounds, because for each dimension the upper bound of a rectangle is always greater than (or

equal to) the lower bound. Because of the correlation all points are located in a triangular shaped subspace of the image space. Furthermore, since in almost all applications all rectangles are small compared to the data space, the points are located in a small strip above the diagonal.

Data structures which rely on a rectangular shaped data space and partition the data space into rectangular cells tend to degenerate for such applications, especially if they are based on hashing techniques. However, the LSD tree overcomes the drawbacks of the transformation technique if a refined bucket split strategy is used. Since the split strategy is crucial to the efficiency of the LSD tree for non-point objects, we devote the next section to this topic.

3.1 The split strategies

In this section, we discuss split strategies suitable for the skew data distributions induced by the transformation technique. First of all, a suitable split strategy must take into account the correlation between lower and upper bounds of rectangles in the original dimension 1, resp. 2, stored in dimensions 1 and 2, resp. 3 and 4.

We will explain two different split strategies, a data dependent and a distribution dependent one. The *data dependent split strategy* is simple: For the split dimension under concern the average over all coordinates of objects stored in the bucket to be split including the object to be inserted is chosen as the split position.

The *distribution dependent split strategy* is a combination of two basic (distribution dependent) split strategies each of them designed for an extreme situation. The first split strategy relies on the (fictitious) assumption that all rectangles are degenerated to points, i.e. the upper and lower bounds coincide for each dimension. Here, all image points are located on the diagonal w.r.t the dimensions 1 and 2, resp. 3 and 4. A suitable split strategy for this case is to split the data cell into two cells containing equally long parts of the diagonal. The split position achieved by this split strategy is denoted bp SP_1 in Figure 3.1.

The second basic split strategy relies on the assumption that all image points are uniformly distributed over the triangular subspace of the image space built from dimensions 1 and 2, resp. 3 and 4. Here, a suitable split strategy halves the data cell into two cells of equal areas. The split position achieved by this split strategy is denoted bp SP_2 in Figure 3.1.

The split position SP calculated by the combined split strategy is the weighted sum of SP_1 and SP_2:

$$SP = \alpha SP_1 + (1-\alpha)\,SP_2\,, \qquad \text{where}$$

$$\alpha = \sqrt[10]{\frac{\textit{data cell area above diagonal}}{(U_d - L_d)^2}}\,.$$

(Here L_d denotes the lower and U_d the upper bound of the data space w.r.t the split dimension d.)

The effect of the choice of α is that for large data cells SP approaches SP_1 while for small cells SP approaches

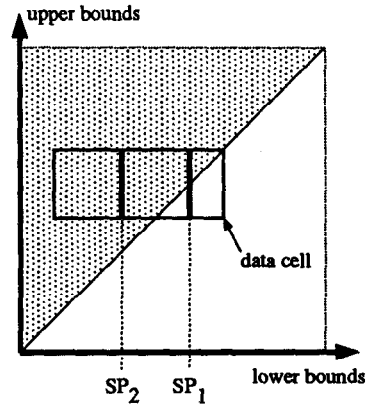

Figure 3.1: Split positions achieved by two basic split strategies

SP_2. This effect is desirable for the usual situation where rectangles tend to be small compared to the data space and hence the image points tend to be located in a strip above the diagonal. We performed simulations with other roots but the 10^{th} root behaved well in all cases.

3.2 The operations

In this section, we discuss the LSD tree operations for non-point objects. The operations *exact match*, *insertion* and *deletion* are identical to the corresponding operations for 4-dimensional points. In the case of a *range query* the situation is different, because the original 2-dimensional query region and the 4-dimensional image query region differ substantially because of the different representations of the objects.

In a range query, the query region can either be an orthogonal rectangle, a circle or a polygon. Independent of the three kinds of query regions we distinguish between two query types for a set of rectangles $\Re$ (see [SK88]):

1. *Rectangle intersection*:
   Given a query region Q find all $R \in \Re$ s.t. $Q \cap R \neq \emptyset$.
2. *Rectangle enclosure*:
   Given a query region Q find all $R \in \Re$ s.t. $R \subseteq Q$.

In contrast to the point situation, the algorithm for circular and polygonal range queries is different from the algorithm for orthogonal range queries. However, due to space limitations of the paper we discuss only orthogonal range queries.

We begin explaining the *rectangle intersection problem for orthogonal query regions*. In this case, the original 2-dimensional query region for rectangles $[l_1, u_1] \times [l_2, u_2]$ is transformed into a 4-dimensional query region for points.

For each original dimension $d \in \{1, 2\}$ we define

$$\varphi([l_d, u_d]) \stackrel{def}{=} [L_d, u_d] \times [l_d, U_d]\,.$$

Then for the original query region $[l_1, u_1] \times [l_2, u_2]$ the image region is given by $\varphi([l_1, u_1]) \times \varphi([l_2, u_2])$. Figure 3.2 illustrates the transformation of a query interval $[l_d, u_d]$.

The area of the image region can be reduced by using the transformation φ' instead of φ. Let E_d denote the

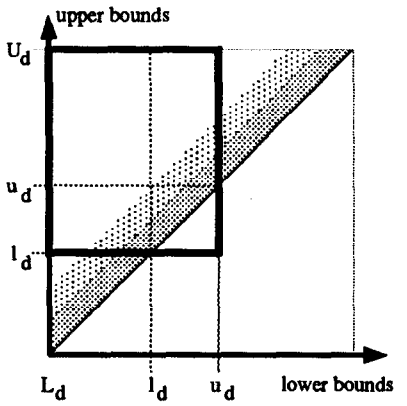

Figure 3.2: Transformation of query interval $[l_d, u_d]$

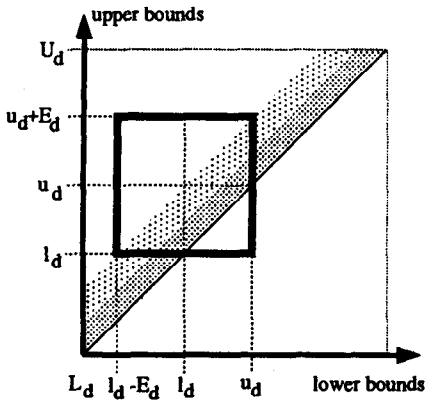

Figure 3.3: Improved transformation of query interval $[l_d, u_d]$

greatest extension of an inserted rectangle for dimension d, then

$$\varphi'([l_d, u_d]) \stackrel{def}{=} [l_d - E_d, u_d] \times [l_d, u_d + E_d] .$$

Figure 3.3 illustrates the improved transformation of a query interval $[l_d, u_d]$.

For the image region the range query algorithm for points can directly be used.

We continue the discussion with the *rectangle enclosure problem for orthogonal query regions*. Since in this case for each dimension d both, the lower and the upper bound of a rectangle, must be enclosed in the query interval $[l_d, u_d]$, we use the simple transformation

$$\vartheta([l_d, u_d]) \stackrel{def}{=} [l_d, u_d] \times [l_d, u_d] .$$

The image query region for the rectangle enclosure problem is smaller than for the rectangle intersection problem. Hence, an enclosure query can be performed more efficiently than an intersection query. Note that this holds only for transformation techniques and not for clipping or overlapping cell techniques.

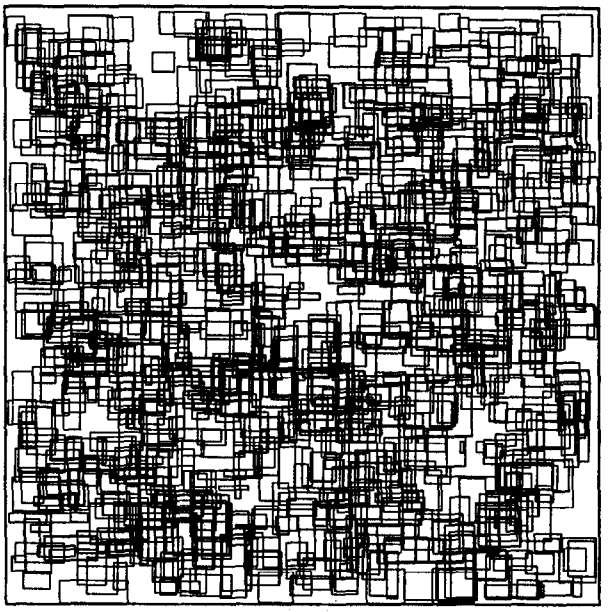

Figure 4.1: "Uniformly distributed" rectangles

4. Performance evaluation

To assess the merits of the LSD tree we have evaluated the performance for rectangles in the plane. We do not discuss the efficiency of the LSD tree for points, because the performance for rectangles is an upper bound of the performance for points. We have implemented an LSD tree on a SUN workstation in Modula-2.

The *internal directory* is stored in an array storing 1000 nodes. An *external directory page* of size 512 bytes contains subtrees up to a height of 6 organized as sequential heaps. We choose bucket capacities of 5 and 50 rectangles.

The simulations are based on a sophisticated random rectangle generator creating sets of 10,000 and 100,000 rectangles according to two different distributions. These distributions are illustrated for 1,000 rectangles in Figures 4.1 and 4.2. Since the cover quotient remains constant at 2.5, in the case of 10,000, resp. 100,000, rectangles the average area of a rectangle is 10, resp. 100, times smaller than for 1,000 rectangles.

Bucket splits are performed according to the split strategies described in section 3.1. In the case of the data dependent split strategy we used a refined insertion procedure: If a new rectangle causes the split of a bucket b_1 which has a brother bucket b_2, i.e. both buckets stem from the same bucket split with split line S, and the capacity of b_2 is not exhausted, the object which is closest to S in b_1 is moved to b_2 and S is updated in the directory. Then the new rectangle can be inserted without a bucket split. Otherwise, b_1 is split.

First, we focus on the *directory evaluation*. We have randomly inserted 10,000, resp. 100,000, uniformly distributed rectangles and 10,000, resp. 100,000, skew distributed rectangles into an initially empty LSD tree. To simulate a "worst case" situation, 100,000 uniformly distributed rectangles have been inserted in "sorted" order. The "sorting" has been carried out by random insertions into an LSD

test data				size of the directory						storage utilization		
number of rectangles	rectangle distribution	split strategy	bucket capacity	nodes (internal and external)	internal nodes	height of the internal directory	number of external directory levels	number of directory pages	average utilization of directory pages	buckets	bucket utilization	overall storage utilization (100 byte per objekt)
100,000	uniform	data	50	2,864	976	13	1	41	73.5 %	2,865	69.8 %	68.2 %
10,000	"	"	"	281	281	9	0	0	-	282	70.9 %	68.1 %
100,000	skew	"	"	2,837	962	13	1	40	74.8 %	2,838	70.5 %	68.8 %
10,000	"	"	"	289	289	9	0	0	-	290	69.0 %	66.3 %
100,000	uniform	distrib.	"	3,577	986	15	1	150	28.6 %	3,302	60.6 %	59.0 %
100,000	skew	"	"	3,526	985	16	1	156	27.0 %	3,266	61.2 %	59.6 %
100,000	skew	data	5	25,882	980	14	2	1,172	34.7 %	25,883	77.3 %	71.8 %
10,000	"	"	"	2,592	979	17	1	67	39.2 %	2,593	77.2 %	71.7 %
100,000	"	distrib.	"	34,688	997	17	2	4,031	14.6 %	30,678	65.2 %	56.0 %
10,000	"	"	"	3,419	994	21	1	221	18.7 %	3,058	65.4 %	58.4 %
100,000	sorted	data	5	23,085	997	204	2	4,836	8.7 %	23,086	86.6 %	69.6 %
100,000	"	distrib.	"	33,787	985	16	2	3,501	16.2 %	29,967	66.7 %	58.1 %

Table 4.1: Size of the directory and storage utilization (directory page size = 512 bytes; max. number of internal nodes = 1000)

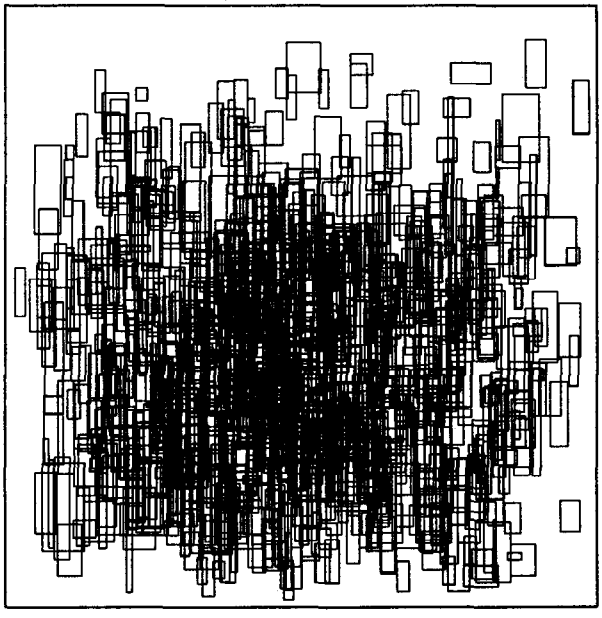

Figure 4.2: "Skew distributed" rectangles

tree with bucket capacity 5 followed by a left to right scan through the LSD leaves. The results are shown in Table 4.1.

It comes out very clearly that the size of the directory does not depend on the data distribution but on the split strategy (and of course on the size of the data set and the bucket capacity). For unsorted situations, the data dependent split strategy performs significantly better than the distribution dependent variant, while, as expected, in the sorted case the distribution dependent split strategy is the winner. For the sorted case and the data dependent split strategy the unbalance of the directory is reflected mainly by the height of the internal directory. Because of the external balancing property the number of external directory levels is 2 as for the distribution dependent split strategy. The utilization of the directory pages is mainly influenced by the split strategy and the bucket capacity (and, for the data dependent split strategy, of course by the order of insertion).

For the same test set we have also measured the *bucket utilization* which is defined as

$$\frac{\text{number of stored objects}}{\text{number of buckets} \times \text{bucket capacity}}$$

and the *overall storage utilization* defined as

$$\frac{\text{number of stored objects} \times 100\,\text{bytes}}{\text{bytes needed for the LSD tree}}$$

which includes the storage space needed for the directory and some administrative informations. Empty buckets are not allocated but represented by nil-pointers in the directory. The results are shown in Table 4.1.

For the data dependent split strategy the bucket utilization is independent of the object distribution and slightly above the theoretical value of 69.3% (ln 2). Due to the refinement of the insertion procedure which prefers small bucket capacities the utilization is even higher for bucket capacity 5. The bucket utilization of 86.6% for the sorted situation is a consequence of the same effect. For the distribution dependent split strategy the bucket utilization is

number of rectangles	rectangle distribution	split strategy	bucket capacity	range query type 1 (0.5% of the data space)			range query type 2 (5% of the data space)		
				objects found	bucket accesses	directory page accesses	objects found	bucket accesses	directory page accesses
100,000	uniform	data	50	579.7	35.6	1.9	5272.8	199.3	6.3
10,000	"	"	"	76.4	9.7	-	585.1	30.2	-
100,000	skew	"	"	818.9	49.9	1.8	6625.9	244.6	6.3
10,000	"	"	"	110.7	13.5	-	744.4	38.2	-
100,000	uniform	distrib.	"	579.7	39.5	3.8	5272.8	229.4	14.2
100,000	skew	"	"	818.9	52.0	4.2	6625.9	276.1	17.7
100,000	skew	data	5	818.9	332.3	26.0	6625.9	1995.3	109.9
10,000	"	"	"	110.7	70.3	4.1	744.4	277.3	11.0
100,000	"	distrib.	"	818.9	347.8	66.9	6625.9	2282.3	350.0
10,000	"	"	"	110.7	73.5	8.7	744.4	304.9	27.5
100,000	sorted	data	5	579.7	537.6	139.6	5272.8	2059.5	633.9
100,000	"	distrib.	"	579.7	251.8	44.5	5272.8	1790.1	249.9

Table 4.2: Range query performance (directory page size = 512 bytes; max. number of internal nodes = 1000)

below 69% but still above 60% and hence not bad at all. A comparison of the overall storage utilization and the bucket utilization convincingly demonstrates that the storage space needed to accommodate the directory is rather small compared to the data storage space.

We now turn our attention to the performance of the LSD tree operations. Clearly, in an *exact match* at most i directory pages and one bucket must be read if the directory contains i external levels. The performance of the *insertion procedure* is easy to estimate, too. For bucket capacity 5, resp. 50, we have between 4 and 5, resp. less than 3, external accesses (directory page and bucket I/O) per inserted object, if 100,000 objects are inserted into an initially empty LSD tree irrespective of the split strategy used.

Hence, we focus on *range queries*. We concentrate on the intersection query because its performance is an upper bound of the enclosure query performance. (Experiments show that the enclosure queries can be carried out 10% faster than intersection queries on the average.) Table 4.2 shows the average number of external accesses for two types of range queries. For square regions of sizes 0.5% and 5% of the size of the data space, we have performed 20 range queries each, at random positions.

As with all other data structures larger query regions lead to fewer disk accesses per found object, because the number of buckets completely contained in the query region grows faster than the number of buckets intersected by the region boundary.

Another important characteristic number is the *hit ratio*, defined as

$$\frac{\text{number of objects found}}{\text{bucket capacity} \times \text{disk accesses}}.$$

The hit ratio is higher for smaller bucket capacities, because of the higher selectivity. For the data dependent split strategy, bucket capacity 5, skew distribution, and query type 2, the hit ratio is 66.4% if only bucket accesses are counted. This is nearly optimal with respect to a normal bucket utilization of 69.3%. For smaller query regions and larger bucket capacities the hit ratio deteriorates: Changing the bucket capacity to 50 yields 54.2%.

The performance results in the sorted situation can be explained by the fact that the data cells tend to be long and small for the data dependent split strategy in this case.

For the remainder of this section we compare the *performance of the LSD tree* and the *multilayer grid file* [SW88] using 5 layers (5L-GF for short). According to the multilayer philosophy layer 5 is implemented as a clipping grid file. We have inserted 50,000 uniform distributed rectangles in random order into both initially empty structures. The cover quotient is 2.5 and the bucket capacity varies from 5 to 30 in steps of 5. It turns out that the 5L-GF is not able to work with bucket capacity 5. After the insertion of 20,974 rectangles a bucket of the clipping layer could not be split because each rectangle stored in this bucket covered the whole corresponding data cell. (For the skew data distribution the 5L-GF runs into a similar error situation even for bucket capacity 10.)

Figure 4.3 shows the bucket utilization for the LSD tree with the data dependent split strategy (LSD_{data}), the LSD tree with the distribution dependent split strategy (LSD_{dis}), and the 5L-GF.

The range query performance is illustrated in Figures 4.4 and 4.5. We have used the same query types as before. For the first, resp. second type, 308, resp. 2712, objects are selected on the average.

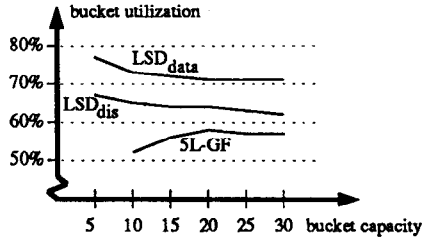

Figure 4.3: Bucket utilization

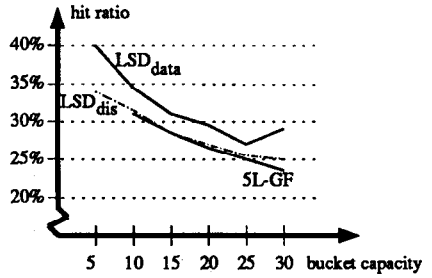

Figure 4.4: Range query performance (0.5% of the data space)

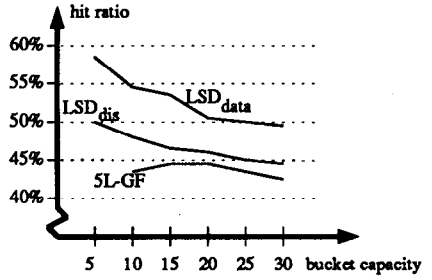

Figure 4.5: Range query performance (5% of the data space)

It turns out that the LSD tree with the data dependent split strategy clearly outperforms the 5L-GF while the LSD tree with the distribution dependent split strategy is at least as efficient as the 5L-GF. We have not compared the exact match and insertion performance because it is obvious that the 5 layers of the 5L-GF do not allow a competitive performance.

It should be noted that besides its better overall performance the LSD tree is much easier to implement than the 5L-GF and does not need an additional (completely different) "overflow" data structure for storing objects which do not fit into the main structure.

5. Conclusion

We have proposed the LSD tree, a data structure supporting efficient spatial access to geometric objects. Its main advantages over other structures are that it performs well for all reasonable data distributions, cover quotients, and bucket capacities, and that it maintains multidimensional points as well as arbitrary geometric objects. These properties make the LSD tree extremely suitable for the implementation of spatial access paths in geometric databases.

In addition to the performance evaluation an analysis of the expected storage utilization and the expected external height proves the efficiency of the LSD tree [HSW89]. At the moment, we are implementing more general (spatial) operations, like non-orthogonal range queries, point queries [SK88] and queries where geometric as well as standard attributes are qualified. Furthermore, we are embedding the LSD tree as spatial access path into the geometric database system Gral [Güt89]. Hence, an empirical study about the benefits of the LSD tree in such an environment can be carried out in the near future.

References

[Ben75] Bentley, J.L.: 'Multidimensional Binary Search Trees Used in Database Applications', Communications of the ACM, Vol. 18, 9, 509-517, 1975

[FSR87] Faloutsos, C., Sellis, T., Roussopoulos, N.: 'Analysis of Object Oriented Spatial Access Methods', Proc. ACM SIGMOD Int. Conf. on Management of Data, 426-439, 1987

[Fred80] Fredman, M.L.: 'The Inherent Complexity of Dynamic Data Structures Which Accommodate Range Queries', IEEE, CH1498–5/80, 1980

[Free87] Freeston, M.: 'The BANG file: a new kind of grid file', Proc. ACM SIGMOD Int. Conf. on Management of Data, 260-269, 1987

[Gut84] Guttman, A.: 'R-Trees: A Dynamic Index Structure for Spatial Searching', Proc. ACM SIGMOD Int. Conf. on Management of Data, 47-57, 1984

[Güt89] Güting, R.H.: 'Gral An Extensible Relational Database System for Geometric Applications', Proc. 15th Int. Conf. on VLDB (1989), to appear

[Hin85] Hinrichs, K.: 'The Grid File System: Implementation and Case Studies of Applications', Doctoral Thesis No. 7734, ETH Zürich, 1985

[HSW88a] Hutflesz, A., Six, H.-W., Widmayer, P.: 'Globally Order Preserving Multidimensional Linear Hashing', Proc. IEEE 4th Int. Conf. on Data Engineering, 572-579, 1988

[HSW88b] Hutflesz, A., Six, H.-W., Widmayer, P.: 'Twin Grid Files: Space Optimizing Access Schemes', Proc. ACM SIGMOD Int. Conf. on Management of Data, 183-190, 1988

[HSW89] Henrich, A., Six, H.-W., Widmayer, P.: 'Paging binary trees with external balancing', Proc. Int. Workshop on Graphtheoretic Concepts in Computer Science (WG '89), Springer Lecture Notes in Comp. Science, to appear

[KS86] Kriegel, H.-P., Seeger, B.: 'Multidimensional Order Preserving Linear Hashing with Partial Expansions', Proc. Int. Conf. on Database Theory, 203-220, 1986

[KS88] Kriegel, H.-P., Seeger, B.: 'PLOP-Hashing: A Grid File without Directory', Proc. IEEE 4th Int. Conf. on Data Engineering, 369-376, 1988

[KW85] Krishnamurthy, R., Whang, K.-Y.: 'Multilevel Grid Files', IBM Research Report, Yorktown Heights, 1985

[LZL88] Litwin, W., Zegour, D., Levy, G.: 'Multilevel Trie Hashing', Proc. Int. Conference Extending Database Technology (EDBT '88), Springer Lecture Notes in Comp. Science, 309–335, 1988

[NHS84] Nievergelt, J., Hinterberger, H., Sevcik, K.C.: 'The Grid File: An Adaptable Symmetric Multikey File Structure', ACM Transactions on Database Systems, Vol. 9, 1, 38-71, 1984

[Otoo86] Otoo, E.J.: 'Balanced Multidimensional Extendible Hash Tree', Proc. 5th ACM SIGACT / SIGMOD Symposium on Principles of Database Systems, 100-113, 1986

[Rob81] Robinson, J.T.: 'The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes', Proc. ACM SIGMOD Int. Conf. on Management of Data, 10-18, 1981

[SK88] Seeger, B., Kriegel, H.-P.: 'Techniques for Design and Implementation of Efficient Spatial Access Methods', Proc. 14th Int. Conf. on VLDB, 360–371, 1988

[SW88] Six, H.-W., Widmayer, P.: 'Spatial Searching in Geometric Databases', Proc. IEEE 4th Int. Conf. on Data Engineering, 496–503, 1988