# Improving the Performance of Multi-Dimensional Access Structures Based on k-d-Trees

Andreas Henrich
Universität Siegen, Fachbereich Elektrotechnik und Informatik, Praktische Informatik
D-57068 Siegen, Germany, Email: henrich@informatik.uni-siegen.de

## Abstract

*In recent years, various k-d-tree based multi-dimensional access structures have been proposed. All these structures share an average bucket utilization of at most* $\ln 2 \approx 69.3\%$. *In this paper we present two algorithms which perform local redistributions of objects to improve the storage utilization of these access structures. We show that under fair conditions a good improvement algorithm can save up to 20% of space and up to 15% of query processing time. On the other hand we also show that a local redistribution scheme designed without care, can improve the storage utilization and at the same time worsen the performance of range queries drastically.*

*Furthermore we show the dependencies between split strategies and local redistribution schemes and the general limitations which can be derived from these dependencies.*

## 1 Introduction

Various efficient multi-dimensional access structures have been proposed. The focus of most of these structures has been on the maintenance of spatial objects, but nearly all can be used as multi-attribute access structures as well [4]. A lot of these access structures use a variant of the *k-d*-tree [2] as directory (see e.g. [11, 10, 9, 5, 8]).

As usual for secondary storage access structures these structures store the objects in buckets of fixed size. Each bucket is associated with a dynamically defined part of the data space called its bucket region.

If a new object has to be inserted into a bucket which is already filled, the bucket has to be split. To this end a split line is determined, which is used as the criterion to decide for each object in the overflowing bucket whether it should remain in the bucket or moved into its newly created sibling. Thereafter, the new split line has to be inserted into the directory.

If we assume uniformly distributed objects which are inserted in random order, this base algorithm of splitting a bucket into two in case of a bucket overflow leads to an average bucket utilization of $\ln 2 \approx 69.3\%$ for all these structures.

Over 30% of wasted storage are unpleasant in two respects:

1. Although secondary storage has become cheap, it should not be a matter of squander.
2. Especially with range queries, searching all objects in a given query region, the bucket utilization limits the query performance. This can be seen easily if we consider the *hit ratio* which is a good measure for the range query performance:

$$\text{hit ratio} \stackrel{\text{def}}{=} \frac{\text{number of objects found}}{\text{bucket accesses} \cdot \text{bucket capacity}}$$

Obviously, the *hit ratio* can not be higher than the average bucket utilization.

Therefore it is worth considering techniques to improve the bucket utilization. A typical approach in this respect for a one-dimensional access structure is the B*-tree. In a B*-tree an insertion employs a local redistribution scheme to delay splitting until two sibling nodes are full. Then the two nodes are divided into three, each $\frac{2}{3}$ full.

Even for multi-dimensional access structures approaches in this direction exist. The *twin grid file* presented in [7] simply uses two grid files in parallel. If the insertion of a new object would enforce a bucket split in one grid file, before actually splitting the bucket, an attempt is made to store the object in the second grid file. Whereas the advantage of this approach is its general applicability, it causes a bad performance for exact match queries because two grid files have to be considered. In [1] a local redistribution scheme is presented for the R*-tree. This scheme exploits the fact, that an R-tree is nondeterministic in allocating entries onto the buckets and redistributes selected objects of an overflowing bucket in order to avoid the split. Since *k-d*-tree based access structures are deterministic in allocating entries onto the buckets there is no obvious extension of this scheme for *k-d*-tree based access structures.

In this paper we present an approach to improve the storage utilization which can be directly applied to multi-dimensional access structures using a variant of the *k-d*-tree [2] as directory [11, 10, 9, 5, 8]. In order to explain the algorithm we use the LSD-tree[1] as an example.

The paper is organized as follows: Section 2 describes those aspects of the LSD-tree which are essential for this paper. In section 3 we present a first

[1] LSD stands for *Local Split Decision*.

approach to avoid bucket splits. The evaluation of this approach will gain deeper insights into the complex dependencies influencing the performance of *k-d*-tree based multi-dimensional access structures. Based on these insights section 4 presents a refined redistribution scheme and experimental results depicting the advantages and limitations of this approach. Section 5 concludes the paper.

## 2 Basic Concepts of the LSD-Tree

For multi-dimensional access structures it is important to distinguish points and extended, i.e. non-point, objects. Conceptually the LSD-tree is an access structure for multi-dimensional points, but it was designed to maintain extended objects as well using the transformation technique [6, 12, 3]. Using the transformation technique does not restrict the applicability of the algorithms presented in this paper, but due to space limitations we can only deal with points here.

Initially, the whole data space corresponds to one bucket. After a certain number of insertions the initial bucket has been filled, and an attempt to insert an additional object causes the need for a bucket split. To this purpose, a *split line* is determined and the objects on one side of the split line are stored in one bucket, while those on the other side are stored in another bucket. After some further insertions, the capacity of another bucket will be exceeded. In this case, a split line in the corresponding bucket region is determined, thus splitting this region into two subregions. This process is repeated each time the capacity of a bucket is exceeded. For example, consider the two-dimensional data space in figure 1. The first split was made according to co-ordinate 50 in the first dimension, giving buckets 1 and 2. Then bucket 2 was split according to position 60 of the second dimension, giving buckets 2 and 3, and so on.

The split lines of an LSD-tree are maintained in a *k-d*-tree based directory. For each split, a node containing the position and the dimension of the split line is inserted into the directory tree (see figure 1).

The directory of an LSD-tree is subdivided into an internal and an external part. The size of the internal part can be specified by the maximum number of internal nodes. The external part of the directory is maintained in directory pages. For the whole directory a so-called *external balancing property* is preserved by a sophisticated paging algorithm described in [5].

We mentioned in passing, that whenever an attempt to insert an additional object into a bucket causes the need for a bucket split, a *split line* has to be determined. In fact, the choice of an appropriate strategy to compute the split lines is crucial for the performance of the access structure. We distinguish two types of split strategies: *Data dependent split strategies* depend only on the objects stored in the bucket to be split. An example is to use the average of all object co-ordinates with respect to the split dimension as split position. *Distribution dependent split strategies* choose the split dimension and the split position independently of the actual objects stored in the bucket to be split. An example based on the assumption of a uniform distribution of the objects is to split a cell

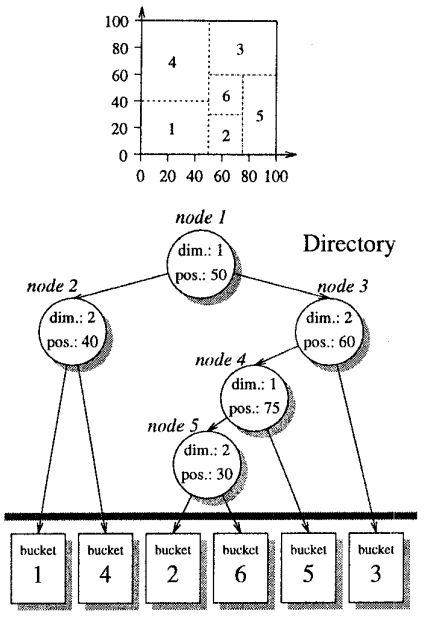


Figure 1: Possible data space partition and associated LSD-tree

into two cells of equal areas.

Since data dependent split strategies calculate the split line based on the objects actually stored in the access structure, on the one hand, they yield a good adaptation of the data space partition. On the other hand, the data space partition depends on the order of insertion and degenerates especially if the objects are inserted in sorted order.

In contrast, the data space partition induced by a distribution dependent split strategy is independent of the order of insertion. The drawback of distribution dependent split strategies is, that a hypothesis about the object distribution must be found in advance.

In section 4 we will see that the use of the local redistribution schemes presented in this paper automatically brings up data dependent split lines, even if the split lines are initially chosen by a distribution dependent split strategy, because the redistribution schemes have to adjust the split lines.

## 3 A Brute Force Approach

In order to improve the bucket utilization, bucket splits have to be avoided whenever possible. A first approach in this direction is to consider the sibling of an overflowing bucket. If this sibling can take an additional element without a bucket split, a local redistribution of objects is performed shifting objects from the overflowing bucket into its sibling and adjusting the split line in the corresponding directory node.

Two cases have to be distinguished: (1) the sibling of the overflowing bucket is a bucket, and (2) the sibling of the overflowing bucket is a subtree with more

then one bucket.

In the first case, shifting an object from the overflowing bucket into its sibling may be either possible or not. In the second case three subcases can occur: (a) An object from the overflowing bucket can be inserted into the sibling without problems. (b) An object from the overflowing bucket can be inserted into the sibling only if recursive redistributions are performed in the sibling. (c) Even recursive redistributions in the sibling do not allow to shift an object from the overflowing bucket into the sibling.

To assess the performance improvement achieved by this algorithm, we inserted 100,000 uniformly distributed points into a 2-dimensional LSD-tree with bucket capacity 5. We compared 4 variants: (1) A structure built without the improvement. (2) A structure where a redistribution is done only if the brother of the overflowing bucket $b$ is a bucket. (3) A structure where redistributions are performed even if the brother of $b$ is a subtree containing multiple buckets, but without recursive redistributions. (4) A structure where recursive redistributions have been allowed.

The results achieved for these structures have been disillusioning. Whereas the bucket utilization can be improved from 69.3 % to 77.5 % (only buckets), 81.9 % (no recursion) or even 85.6 % (full recursion) the number of directory pages in the structure is increased from 1342 to 1895 (no recursion) or even 3938 (full recursion). Only if a redistribution is only performed if the sibling of the overflowing bucket is a bucket, the number of directory pages is decreased a bit to 1171.

Furthermore, the performance of range queries has been improved only for large query regions. For small query regions, the hit ratio is worse than for the normal structure.

The only variant of the algorithm which gains satisfactory results, is the one which moves objects only if the sibling of the overflowing bucket is itself a bucket.

To show the reasons for the problems with the other variants, we have inserted 500 objects in trees with the same parameter settings and reported the resulting data space partition and the shape of the directory tree in figure 2. Two effects can be observed: (1) In contrast to the directory trees for the first two variants, the directory trees for the other variants are much more unbalanced. (2) Especially if recursive redistributions are allowed, the bucket regions tend to be elongated. On the other hand, it is useful for a high range query performance to have similar shapes for bucket regions and query regions.

The main reason for the stated effects is that every time, we shift an object from an overflowing bucket $b$ into its sibling which is not a bucket, but a subtree containing multiple buckets, we make the tree a little bit more unbalanced, because the sibling is already heavier than $b$ and the redistribution aggravates this unbalance. The only variant of the presented approach which does not suffer from these problems is the one which only shifts objects if the sibling of the overflowing bucket is itself a bucket. In contrast to the other variants, this variant really does something like rebalancing.

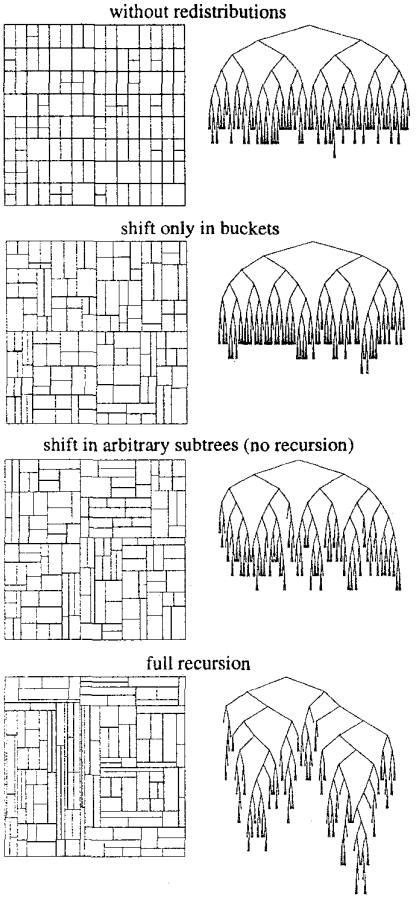


Figure 2: Resulting data space partition and LSD-tree for the brute force approach

## 4 A Refined Approach

The experiences reported in section 3 show, that we should shift an object only if the subtree giving away the object is heavier than the receiving subtree. The simplest approach in this direction is to shift objects only between direct sibling buckets. A natural extension of this approach is as follows:

We check, whether shifting objects into the sibling bucket is possible. If it is not possible, we move up one level in the directory and try to redistribute objects between the subtree containing the overflowing bucket and its sibling subtree. If this again fails, we can continue to move up in the directory or we can stop our attempt to avoid a bucket split.

### 4.1 An Example

In the upper left corner of figure 3 a set of objects is given for the data space partition of figure 1. Assume that only two objects fit into one bucket. The object $o_1$ has to be inserted into the structure. Because of its location in the data space, $o_1$ must be stored in bucket

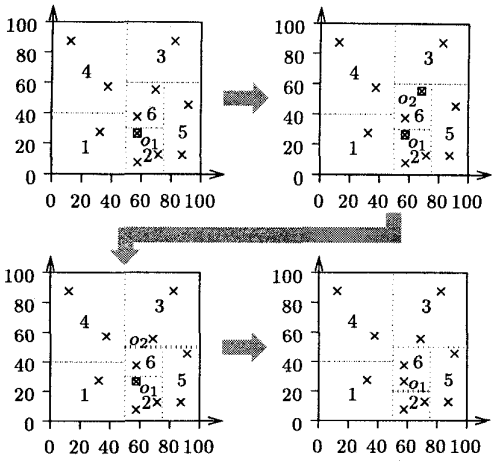


Figure 3: Example for the refined algorithm

2, but bucket 2 is already filled.

In order to avoid a bucket split, we consider the children of the father of bucket 2 and check whether the direct sibling of bucket 2, i.e. bucket 6, can take additional objects. Since bucket 6 is already filled, this attempt fails.

Now we move up one level in the directory, i.e. we consider the children of *node 4* with respect to figure 1. Since $o_1$ has to be inserted into the subtree with root *node 5*, we must try to shift an object from this subtree into the other child of *node 4*, i.e. into bucket 5. Since bucket 5 is already filled this attempt fails.

Again we move up one level in the directory and consider the children of *node 3*. Since $o_1$ has to be inserted into the subtree with root *node 4*, we must try to shift an object from this subtree into the other child of *node 3*, i.e. into bucket 3. To this end, we delete the object $o_2$ from the subtree with root *node 4*. This situation is sketched in the upper right corner of figure 3. Note, that objects represented by a ⊠ are not stored in the structure in the actual situation, but are to be inserted in the following steps.

Now object $o_2$ has to be inserted into the other child of *node 3*, i.e. into bucket 3, and the split line has to be adjusted. The resulting data space partition is shown in the lower left corner of figure 3.

Now that we have relieved the subtree with root *node 4* we can start a new attempt to insert object $o_1$ into this subtree. As before, $o_1$ has to be inserted into bucket 2, but bucket 2 is already filled.

To avoid a bucket split, we again check whether the direct sibling of bucket 2, i.e. bucket 6, can take additional objects. Since bucket 6 contains only one object, this seems to be possible and we can try to relieve bucket 2. To this end, we search the object located closest to the split line separating buckets 2 and 6 among the objects stored in bucket 2 and $o_1$. $o_1$ itself happens to be this object. Hence, $o_1$ is inserted into bucket 6 and the split line in *node 5* is adjusted. The resulting data space partition is given in the lower right corner of figure 3.

```
FUNCTION InsertObject (root, o_new, l);
{ inserts object o_new into the LSD-tree with root root and
  performs local redistributions in order to avoid bucket
  splits in subtrees up to a height of l }
BEGIN
  h1 := root;  stack := EmptyStack();
  WHILE h1 is a directory node DO
    { search the bucket which has to take o_new }
    Push(stack, h1);
    IF o_new[s_dim(h1)] < s_pos(h1) THEN
      h1 := left son of h1;
    ELSE
      h1 := right son of h1;
    END IF;
  END WHILE;
  Push(stack, h1);
  i := 0;
  LOOP
    h2 := Pop(stack);
    IF RecInsertObject(h2, o_new) THEN
      { insertion performed without a bucket split }
      RETURN;
    ELSE IF StackIsEmpty(stack) OR i ≥ l THEN
      { stop trying local redistributions }
      EXIT;
    ELSE  { try next level }
      i := i + 1;
    END IF;
  END LOOP;
  { redistribution has not been successful }
  Split(h1);  InsertObject(root, o_new, l);
END InsertObject;
```

Figure 4: *InsertObject* procedure

This example shows the basic principle which means to step up in the directory and try at each level to redistribute objects between the children until either a redistribution has been successful or a predefined maximum number $l$ of levels has been reached. Furthermore it shows, that recursive applications of this redistribution scheme may be necessary, because shifting objects between subtrees might not relieve the overflowing bucket but another bucket in the corresponding subtree.

## 4.2 The Algorithm

The algorithm implicitly given in the example above, is stated in more detail in the procedures shown in figure 4 and figure 5. The procedure *InsertObject* describes the modified insertion algorithm for an LSD-tree. First of all, a stack is created to maintain the directory nodes traversed while searching the bucket which must take the new object. This search is performed in the following while-loop. When the bucket is reached, the bucket itself is also pushed on top of the stack. Now the elements are taken from the stack one after the other, and calls of the procedure *RecInsertObject* are performed until either *RecInsertObject* reports success or the stack is empty or the given maximum number of levels $l$ has been reached. In the two

```
FUNCTION RecInsertObject (t, o_new) : BOOL;
{ tries to insert o_new into the subtree t of an LSD-tree and
  performs local redistributions if appropriate }
BEGIN
  h_1 := t;
  WHILE h_1 is a directory node DO
    IF o_new[s_dim(h_1)] < s_pos(h_1) THEN
      h_1 := left son of h_1;
    ELSE
      h_1 := right son of h_1;
    END IF;
  END WHILE;
  IF h_1 can take o_new THEN
    insert o_new; RETURN true;
  ELSE IF t is a bucket THEN
    RETURN false;
  ELSE
    s_1 := the son of t for which the data region contains o_new;
    s_2 := the other son of t;
    IF Height(s_2) > Height(s_1) THEN
      RETURN false;
    ELSE
      LOOP
        IF DeleteClosestObject(s_pos(t), s_dim(t), s_1, o_new,
                               {out} o_del, s'_pos) THEN
          s_pos(t) := s'_pos;
          IF RecInsertObject(s_2, o_del) THEN
            IF o_del = o_new OR
               RecInsertObject(s_1, o_new) THEN
              RETURN true;
            END IF;
          ELSE
            undo all changes; RETURN false;
          END IF;
        ELSE
          undo all changes; RETURN false;
        END IF;
      END LOOP;
    END IF;
  END IF;
END RecInsertObject;
```

Figure 5: *RecInsertObject* procedure

latter cases, no redistribution has been possible and a bucket split has to be performed[2].

The procedure *RecInsertObject* tries to insert a new object into a subtree of an LSD-tree and performs local redistributions if appropriate. First of all, the bucket of the subtree, which must take the new object is determined. If this bucket can take the object, it is inserted and success is reported. Otherwise, we have to check, whether the whole subtree consists only of this bucket. In this case, we have to report failure. If the subtree consists of multiple buckets, and the

[2]Obviously the stated procedure does not reflect any possible optimization. For example, a fast stack implementation can be used because only the $l+1$ elements on top of the stack have to be maintained and of course a real implementation will not use a recursive call of *InsertObject* when a bucket has been split, but use a loop.

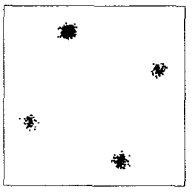

Figure 6: 500 objects generated according to the multi-heap-distribution

height of the child which has to take the new object, is not smaller then the height of its sibling, we can try to redistribute objects between the children. To this end, we perform three steps: (1) We remove the object located closest to the split line from the subtree which has to take the new object. (2) We insert the removed object into the sibling. Since this insertion is done by a recursive call of *RecInsertObject*, recursive redistributions are performed where appropriate. (3) If the removed object has not been the new one, we can now try to insert the new object into the relieved subtree. It has to be noted, that if this insertion fails, the stated procedure tries to remove another object from the subtree, because the object(s) removed up to now and the recursive redistributions might not have freed space in the overflowing bucket.

### 4.3 Experimental Results

To cover a wide range of application profiles, we used three data sets for the experiments with the refined algorithm: (1) 100,000 uniformly distributed 2-dimensional points inserted in random order. (2) 100,000 uniformly distributed 2-dimensional points sorted with respect to their distance to the point (0 , 0) to examine the effect of the data dependency of the split lines induced by the redistribution scheme which adjusts split lines in order to avoid bucket splits. (3) 100,000 2-dimensional points generated according to a *multi-heap distribution* which is illustrated in figure 6. For each heap the points have been inserted in random order, but the heaps have been inserted in sequence.

The bucket capacity has been 5 and the maximum number of internal directory nodes 500. We used 512 bytes per directory page such that external directory pages could take subtrees up to a height of 6.

Table 1 states the bucket utilization, the directory size and the construction time. The first two columns have been achieved without local redistributions, using a distribution and a data dependent split strategy. The remaining columns state the values if redistributions are allowed for subtrees up to a height of $l$. Here the initial split line is computed by the distribution dependent split strategy.

It becomes obvious that for the uniform distribution – in contrast to the brute force approach – not only the bucket utilization but also the directory page utilization can be improved.

Because redistributions are only performed, if this does not increase the unbalance of the structure, the

| | $l = 0$ (distrib.) | $l = 0$ (data) | $l = 1$ | $l = 2$ | $l = 3$ | $l = 4$ | $l = 5$ |
|---|---|---|---|---|---|---|---|
| *uniform distribution* | | | | | | | |
| bucket utilization | 69.3 % | 73.3 % | 77.5 % | 79.7 % | 81.7 % | 82.3 % | 82.6 % |
| height of directory | 18 | 20 | 20 | 21 | 20 | 20 | 19 |
| external directory pages | 1342 | 1370 | 1171 | 1169 | 1051 | 953 | 922 |
| external height of directory | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| construction time in sec. | 211 | 287 | 255 | 320 | 455 | 794 | 1207 |
| *insertion in sorted order* | | | | | | | |
| bucket utilization | 69.3 % | 67.0 % | 86.9 % | 88.0 % | 88.2 % | 88.5 % | 88.6 % |
| height of directory | 18 | 526 | 457 | 359 | 353 | 348 | 351 |
| external directory pages | 1342 | 7476 | 4930 | 4220 | 4163 | 4116 | 4136 |
| external height of directory | 2 | 5 | 3 | 3 | 3 | 3 | 3 |
| construction time in sec. | 158 | 806 | 812 | 760 | 809 | 866 | 1069 |
| *multi-heap distribution* | | | | | | | |
| bucket utilization | 68.9 % | 73.2 % | 77.4 % | 79.9 % | 81.1 % | 82.0 % | 82.5 % |
| height of directory | 25 | 41 | 44 | 41 | 31 | 41 | 40 |
| external directory pages | 1562 | 1453 | 1196 | 1187 | 1059 | 979 | 943 |
| external height of directory | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| construction time in sec. | 283 | 307 | 360 | 494 | 586 | 964 | 1460 |

Table 1: Storage utilization and directory size for the refined algorithm

data space partitions and directory trees depicted for 500 inserted objects in figure 7 do not show any degeneration. Consequently, the query performance stated in figure 8 is superior to the variant without local redistributions especially for large query regions.

The values for sorted insertions given in table 1 show that only the pure use of a distribution dependent split strategy brings up a structure which is completely insensitive with respect to the order of insertion[3]. On the other hand, using the redistribution scheme yields slightly better results than simply using a data dependent split strategy, because the local redistributions smooth the degeneration of the structure. This also becomes obvious from figure 9 depicting the resulting data space partition and LSD-tree after 500 insertions. Finally table 2 depicting the performance for window queries confirms these relations.

The results achieved with the multi-heap distribution show that the degeneration of the structure observed for completely pre-sorted objects will hardly occur in practice. Although the objects are inserted heap by heap, only the shape of the corresponding LSD-trees given in figure 10 (for 500 objects) reflects the sorted insertions. The bucket utilization and the directory size stated in table 1 and the query performance depicted in figure 8 are nearly the same as for the uniform distribution.

## 5 Conclusion

We have presented two algorithms to improve the storage utilization of $k$-$d$-tree based access structures. The brute force approach showed, that just avoiding bucket splits does not yield satisfactory results.

[3]In our experiments insertions using a distribution dependent split strategy have even been faster for pre-sorted objects, because buffering can be much more effective in this case.

| found objects | $l = 0$ (distr.) | $l = 0$ (data) | $l = 1$ | $l = 2$ | $l = 5$ |
|---|---|---|---|---|---|
| bucket accesses | | | | | |
| 2.5 | 3.3 | 32.3 | 22.6 | 22.3 | 22.3 |
| 40 | 19 | 131 | 91 | 90 | 89 |
| 642 | 211 | 656 | 468 | 463 | 462 |
| 10266 | 3049 | 4700 | 3318 | 3285 | 3273 |
| directory page accesses | | | | | |
| 2.5 | 2.1 | 16.9 | 10.5 | 8.7 | 8.6 |
| 40 | 3.8 | 57.3 | 35.5 | 29.2 | 28.9 |
| 642 | 17 | 242 | 151 | 124 | 122 |
| 10266 | 163 | 1512 | 966 | 819 | 794 |

Table 2: Necessary page accesses for structures with sorted insertions

Rather a bucket split should only be avoided if the necessary local redistribution does not increase the imbalance of the structure. The experimental results achieved with the refined algorithm show that it is possible to improve the storage utilization by 10 % to 15 % with reasonable additional effort for insertions. As a consequence the performance with large range queries is improved in the same degree. Nevertheless, an important drawback of all improvement algorithms based on local redistributions is, that they introduce data dependent split lines and hence the access structure becomes sensitive for pre-sorted insertions.

Another main conclusion of this paper is that it is not useful to apply a data dependent split strategy. Rather one should use a distribution dependent split strategy and our redistribution scheme with $l = 1$.

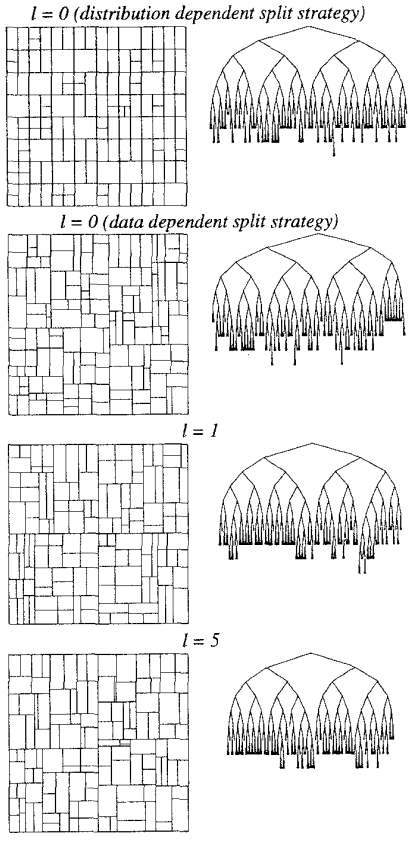


Figure 7: Resulting data space partition and LSD-tree for the refined approach (uniform distribution)

## References


[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, 1990.

[2] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[3] A. Henrich. Adapting the Transformation Technique to Maintain Multi-Dimensional Non-Point Objects in *k-d*-Tree Based Access Structures. In *Proceedings of the 3rd ACM Workshop on Advances in Geographic Information Systems*, Baltimore, Maryland, December 1995. ACM Press.

[4] A. Henrich and J. Möller. Extending a spatial access structure to support additional standard attributes. In M. J. Egenhofer and J. R. Herring, editors, *Proc. 4th Int. Symposium on Advances in Spatial Databases (SSD'95)*, volume 951 of *LNiCS*, pages 132–151. Springer, 1995.


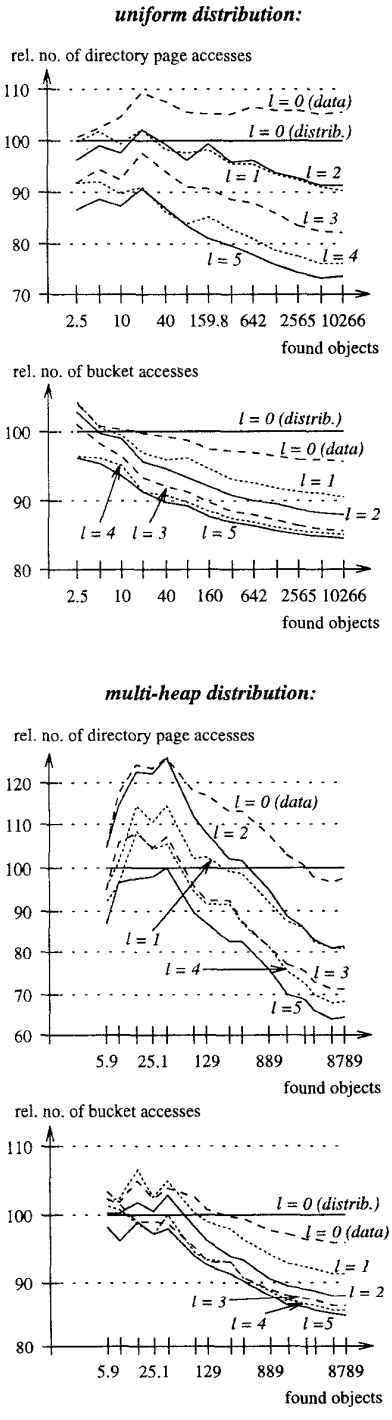


Figure 8: Performance for window queries using the refined approach

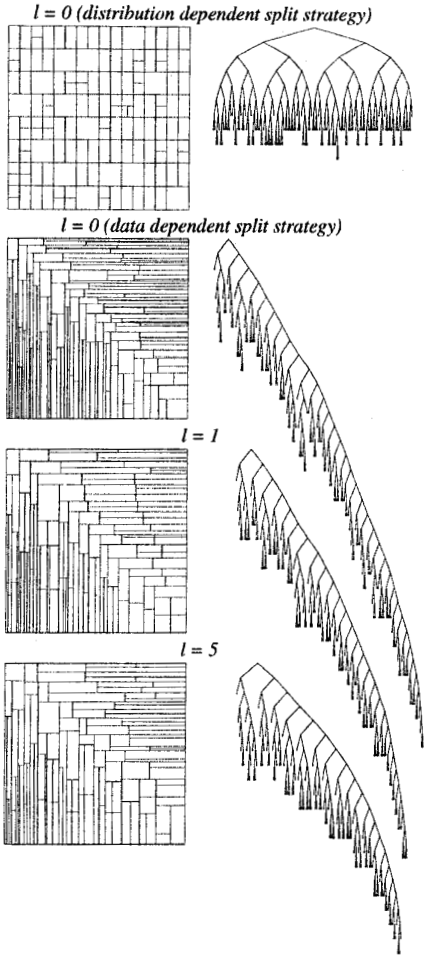


Figure 9: Resulting data space partition and LSD-tree for the refined approach (sorted insertions)

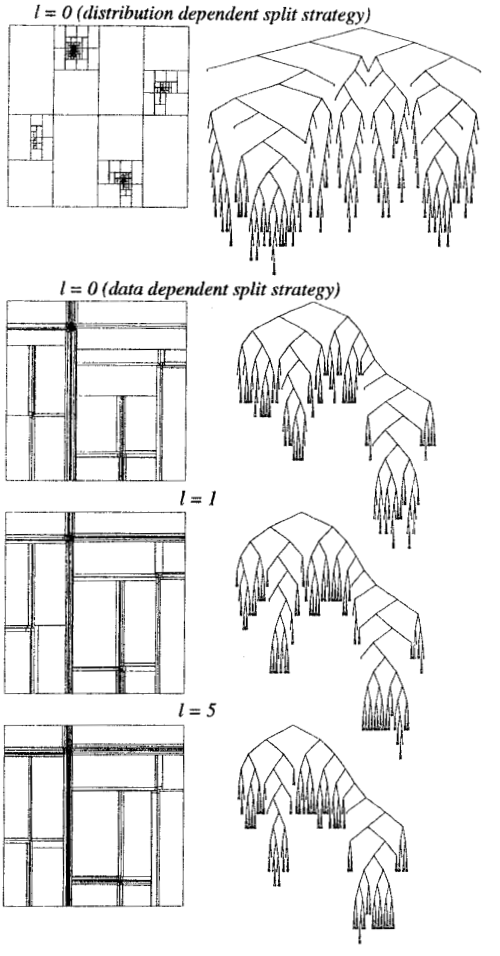


Figure 10: Resulting data space partition and LSD-tree for the refined approach (multi-heap distribution)

[5] A. Henrich, H.-W. Six, and P. Widmayer. The LSD-tree: spatial access to multidimensional point and non point objects. In *Proc. 15th Int. Conf. on VLDB*, pages 45–53, 1989.

[6] K. Hinrichs. *The grid file system: implementation and case studies of applications.* Dissertation Nr. 7734, ETH Zürich, 1985.

[7] A. Hutflesz, H.-W. Six, and P. Widmayer. Twin Grid Files: Space Optimizing Access Schemes. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 183–190, 1988.

[8] M.J.van Kreveld and M.H. Overmars. Divided k-d Trees. *Algorithmica*, 6:840–858, 1991.

[9] D.B. Lomet and B. Salzberg. A Robust Multi-Attribute Search Structure. In *Proc. IEEE 5th Int. Conf. on Data Engineering*, pages 296–304, 1989.

[10] B.C. Ooi, K.J. McDonell, and R. Sacks-Davis. Spatial kd-Tree: An Indexing Mechanism for Spatial Databases. In *IEEE COMPSAC*, pages 433–438, 1987.

[11] J.T. Robinson. The K-D-B-Tree: A Search Structure for Large Multdimensional Dynamic Indexes. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 10–18, 1981.

[12] B. Seeger and H.-P. Kriegel. Techniques for design and implementation of efficient spatial access methods. In *Proc. 14th Int. Conf. on VLDB*, pages 360–371, 1988.