The LSD^h-tree: An Access Structure for Feature Vectors

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

Abstract

Efficient access structures for similarity queries on feature vectors are an important research topic for application areas such as multimedia databases, molecular biology or time series analysis. Recently different access structures for high dimensional feature vectors have been proposed namely the SS-tree, the VAMSplit R-tree, the TV-tree, the SR-tree and the X-tree. All these access structures are derived from the R-tree. As a consequence, the fanout of the directory of these access structures decreases drastically for higher dimensions. Therefore we argue that the R-tree is not the best possible starting point for the derivation of an access structure for high-dimensional data. We show that k d-tree-based access structures are at least as well suited for this application area and we introduce the LSD^h-tree as an example for such a k d-tree-based access structure for high-dimensional feature vectors. We describe the algorithms for the LSD^h-tree and present experimental results comparing the LSD^h-tree and the X-tree.

Introduction

The efficient processing of similarity queries is an important requirement in various application areas where objects are represented by so-called feature vectors. Feature vectors are e.g. common in multimedia systems, where multimedia objects are mapped to feature vectors in a high-dimensional space Other areas where feature vectors are used to represent ob jects include CAD or molecular biology

Recently some specialized access structures have been proposed to support similarity queries on high-dimensional feature vectors namely the SS-tree the VAMSplit R-tree the TV-tree the SR-tree and the X-tree All these access structures can be envisaged as extensions of the spatial R-tree. In the present paper we will argue that the R-tree is not the best possible starting point for the derivation of an access structure for high-dimensional data for several reasons. We show that k d-tree-based access structures are at least as well suited for this application area and we introduce the LSD^h-tree as an example for such a k d-tree-based access structure for high-dimensional feature vectors. Experimental results presented in section demonstrate the potential of our approach. To this end we compare our access structure with the X-tree which has been shown to be superior to the TV-tree and the R*-tree in

The paper is organized as follows: Section gives a rough classification of spatial access methods. These access methods build a natural starting point for the derivation of an access structure for high-dimensional feature vectors. In section we discuss the ideas behind the SS-tree, the VAMSplit R-tree, the TV-tree, the SR-tree and the X-tree. In section we introduce the LSD^h-tree and describe the most important algorithms for this access structure. Thereafter in section 5w e present the results of experiments we performed to compare the X-tree and the LSD^h-tree. Finally section concludes the paper.

Spatial Access Methods

Most effort with respect to multi-dimensional access structures has been spent in the context of spatial applications in the past decades. Besides algorithms for exact-match queries and range queries algorithms to perform nearest neighbor queries have been proposed for various types of spatial access structures Given a two- or three-dimensional query point, the proposed algorithms yield the m closest objects in the access structure in sorted order. Hence, it seems to suggest itself to employ a spatial access structure and to adapt this structure for higher dimensions when dealing with similarity queries for high-dimensional feature vectors. However, various studies show that the performance of nearest neighbor queries becomes worse for higher dimensions
Hence, a straight forward application of spatial access structures for high-dimensional data seems to be inappropriate. On the other hand, the studies for high-dimensional data assume a uniform distribution and independence for the values in the single dimensions. Especially high-dimensional data usually does not ful-

fill these assumptions. If the access structure can exploit the distribution characteristics of the data, there remains some chance to achieve a satisfactory performance even for high-dimensional feature vectors.

As a consequence, we have to look for a spatial access structure which is flexible enough to exploit the distribution characteristics of the data and which does not suffer from high-dimensional data in other respects. To this end we consider the best understood spatial access structures in a little more depth.

Like the one-dimensional B-tree spatial access structures store the objects in buckets of fixed size. When a bucket B overflows the objects stored in B are distributed over two new buckets according to a given split strategy and one or more entries representing the split decision are inserted into the directory of the access structure. Roughly spoken, three types of spatial access structures can be distinguished:

- The rst group are *hash-based structures* These structures apply hash techniques to avoid a large directory Typical representatives of this group are the grid file]o r the buddy-tree Unfortunately these structures are relatively rigid in the way buckets have to be split in case of an overflow. Especially for skewly distributed objects and for higher dimensions this induces an unacceptably low bucket utilization. Structures like the grid file which try to overcome this problem by employing a small directory have only deferred the problem into the directory which grows superlinear with the dimensionality
- The second type of spatial access structures can be called the *R-tree family* These access structures provide complete freedom when splitting a bucket, because the split decision is maintained in the directory storing a bounding rectangle for each bucket (or directory page) together with the reference to the bucket (or directory page). Other advantages of the R-tree family are their robustness and the fact that due to the use of the bounding rectangles areas of the data space with no data points are not covered by the directory entries. However, for t-dimensional feature vectors the bounding rectangles in the directory become t-dimensional bounding intervals which require disproportional much storage space and reduce the fanout of the directory tree.
- Access structures which use a *generalized k d-tree* as directory avoid these problems with large directory entries. They employ($t-1$)-dimensional hyperplanes to represent split decisions. In the directory these hyperplanes are represented by the number of the split dimension and the split po sition. Obviously this restricts the freedom when splitting a bucket, because the partition of the objects has to be represented by a hyperplane. On the other hand, the size of the directory entries is independent of the number of dimensions of the data space. The main problem with k d tree-based approaches is the paging algorithm for the directory tree. However, various interesting approaches for this problem have been proposed Especially the latter approaches have shown a competitive performance in the spatial context.

When we compare R-tree-based access structures and k d-tree-based access structures with respect to their suitability for high dimensional data there are two main aspects: On the one hand, R-tree-based access structures provide more freedom when splitting a bucket. In fact, all split strategies applied for k d tree-based access structures can be applied to R-tree-based access structures as well. Therefore the question is if the split strategies which are applicable for k d tree-based access structures are sufficient to deal with skewly distributed feature vectors. On the other hand, the potential fanout of the directory pages of k d-tree-based access structures is higher for high-dimensional access structures.

From our experiences with spatial access structures we had the impression that the advantages of k d tree-based access structures should at least compensate the advantages of R-tree-based access structures. Therefore we have employed the LSD-tree as the starting point for our considerations. With some minor modifications reflecting the special requirements of high-dimensional data this led to the LSD$^{\mathrm{h}}$-tree which is presented in section in detail.

Related Approaches

Recently five access structures have been proposed for similarity queries on feature vectors.

The TV-tree is based on the idea to use only a few of the features for the directory entries, utilizing additional features whenever the additional discriminatory power is absolutely necessary The key point of the structure is that features are introduced on a 'when needed' basis. To this end, the TV-tree assumes that the dimensions of the feature vectors are ordered by 'importance'. The index entries of the TV-tree consist of minimum bounding regions (spheres) which address only the features actually considered. This way the TV-tree can soften the fanout problem of the

R-tree, but it cannot completely avoid this problem.

The SS-tree also uses spheres to represent the minimum bounding regions in the directory In contrast to the TV-tree the SS-tree always considers all dimensions of the feature space. In order to increase the bucket utilization the SS-tree uses a concept of forced reinserts.

The VAMSplit R-tree is an optimized R-tree constructed in a top-down manner. The applied tree construction algorithm is based on that of the k d-tree, i.e. the VAMSplit R-tree constructs the tree structure by partitioning points recursively with a coordinate plane which is orthogonal to the dimension with the highest variance. Similar to the LSD^h-tree the VAMSplit R-tree is inspired by the observation that a k d tree-based access structure performs well in the high-dimensional context. However, with the VAMSplit R-tree the k d-tree is roughly spoken employed only to calculate the partitioning of the objects during a bucket split. The directory remains similar to the R-tree, which means that a bounding rectangle is maintained with each pointer in the directory

Also the X-tree can be envisaged as an extension of the R-tree. The main idea of the extension is to avoid overlap of the minimum bounding regions (rectangles) in the directory by using a new organization of the directory which is optimized for high-dimensional space. Instead of allowing splits that introduce high overlaps, directory nodes are extended over the usual block size resulting in so-called supernodes. Besides describing the structure itself, Berchtold, Keim and Kriegel present detailed experimental results demonstrating the superiority of the X-tree with respect to the TV-tree.

Finally the SR-tree is an improvement of the SS-tree. The distinctive feature of the SR-tree is that it specifies the region associated with a pointer in the directory by the intersection of a bounding sphere and a bounding rectangle. The introduction of bounding rectangles permits neighborhoods to be partitioned into smaller regions than the SS-tree and improves the disjointness among the regions. On the other hand, the size of the directory entries is significantly increased by this approach.

Due to the use of minimum bounding regions (spheres or rectangles) in the directory all these access structures have problems with a limited fanout for high-dimensional data. Furthermore, except for the VAMSplit R-tree, the minimum bounding regions for the directory entries on one level will usually overlap to some extend. On the other hand, these problems do not arise with k d-tree-based access structures.

It remains to mention that we have proposed a k d tree-based access structure for document representations in vector space in Like the LSD^h-tree this access structure is an extension of the spatial LSD-tree. However, this extension has been especially designed for the vector space model for document retrieval. It comprises a cluster split technique which exploits the fact that over of the components of the maintained document representation vectors are Furthermore, special heuristics to speed up the similarity query processing for the vector space model are used. In contrast to this application specific access structure the present paper proposes a general access structure for feature vectors.

The LSD^h-tree

In the following we first sketch those ideas from the LSD-tree which can be directly employed for the high-dimensional variant. This comprises the basic concepts (section and the algorithm for nearest neighbor queries (section which can be used for similarity queries as well. Thereafter we present the special split strategy we use for high-dimensional data (section and the technique we employ to avoid the maintenance of empty space (section

Basic Idea of the Spatial LSD-tree

As usual for access structures which support spatial access to point objects the LSD-tree divides the data space into pairwise disjoint data cells. With every data cell a bucket of fixed size is associated, which stores all objects contained in the cell. In this context a data cell is often called *bucket region*

Figure illustrates the creation of an LSD-tree. In this example we assume that a bucket can hold two points. Initially the whole data space corresponds to one bucket. After two insertions the initial bucket has been filled, and an attempt to insert a third object causes the need for a bucket split. To this purpose, a *split line* is determined. In our example the split is performed in dimension at position The objects on the left side of the split line remain in the old bucket, while those on the right side are stored in a new bucket. The split is represented by a directory node containing the split dimension and the split position Thereafter the new object can be stored in bucket With the next insertion we again achieve an overflow in bucket Again the bucket is split into two, and the split decision is represented in the directory tree by a new node. This process is repeated each time the capacity of a bucket is exceeded.

Usually the directory grows up to a point where it cannot be kept in main memory any longer. In this case, subtrees of the directory are stored on secondary

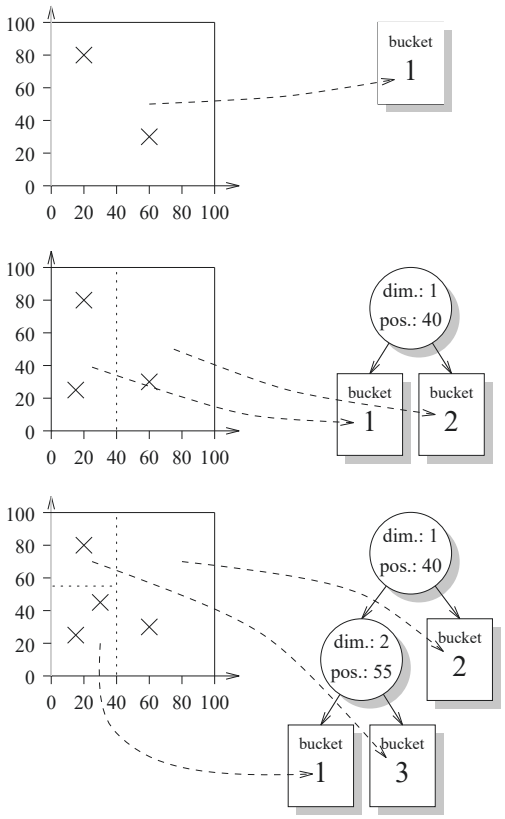

Figure The creation of an LSD-tree

memory whereas the part of the directory near the root remains in main memory For the details of the paging algorithm we refer to

Nearest Neighbor Queries

The following algorithm efficiently solves the m nearest neighbor problem where we look for the m points in the structure located closest to a given query point. Since the details of this algorithm can be found in]w e restrict ourselves to a terse description here.

To describe the processing of a m-nearest neighbor query,w e have to introduce the term of a *data region* The *data region* of a bucket is its bucket region, and the *data region* of a directory node is defined as the union of the two data regions of its sons.

In principle the m-nearest neighbor algorithm works as follows:

We start at the root of the directory and search for the bucket for which the bucket region contains the starting point p of the query Every time during this search when we follow the left son, we insert the right son into an auxiliary data structure NPQ node priority queue), where the element with the smallest distance between p and its data region has highest priority and every time we follow the right son, we insert the left son into NPQ

Then the objects in the found bucket are inserted into another auxiliary data structure OPQ object priority queue), where the object with the smallest distance to p has highest priority Thereafter the objects with a distance to p less than or equal to the minimal distance between p and the data region of the first element in NPQ are taken from OPQ These objects are already sorted correctly with respect to their distance to p They represent the closest neighbors.

Now the directory node or bucket with highest priority is taken from NPQ If it happens to be a directory node, a new search is started choosing always the son on the side of the split line facing p The other son is inserted into NPQ The bucket determined in this way is processed inserting the objects stored in this bucket into OPQ and removing the objects with a distance to p less than or equal to the minimal distance between p and the first element in NPQ from OPQ Thereafter the process is continued in the same way until either m objects are taken from OPQ or NPQ and OPQ are empty which means that m is larger than the number of objects in the access structure.

To avoid unnecessary directory pages accesses, the algorithm described so far can be slightly modified: Every time we access the root of a directory page d,a ll references to buckets or other directory pages stored in d are inserted into NPQ This assures that each directory page is accessed at most once.

Split Strategy

An important part of the insertion algorithm of the LSD-tree is the split strategy which determines the split position and the split dimension in case of a bucket split. In]w e distinguished 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 coordinates 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 is to split a cell 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. However, in accordance with the results presented in]w e observed that the risk for a degeneration of the directory tree caused by pre-sorted data decreases for higher dimensions. On the other hand, high dimensional data is usually more skewly distributed. Therefore we propose to apply a data dependent split strategy (e.g. the average over the coordinate values of the objects in the bucket to be split) for the determination of the split position when concerned with feature vectors. With respect to the split dimension we have to consider the fact that feature vectors usually contain features with only few potential feature values. Therefore we propose the following split strategy:

Assume we have to maintain t-dimensional feature vectors and the dimensions are numbered from to $t -$ Let d_{old} be the split dimension used in the node referencing the bucket to be split. Set i to If there are different feature values for dimension d_{old} $i)$m od t in the bucket to be split, use dimension d_{old} $i)$m od t as the split dimension. Otherwise increase i by1 until there are different feature values for dimension d_{old} $i)$m od t in the bucket to be split.

This strategy simply assures that a split dimension is used only if there are different feature values in this dimension in the bucket which has to be split.

For very high dimensional data the described split strategy may prefer lower dimensions. Therefore it can be extended in a way that the dimension with the highest variance for the object coordinates in the bucket to be split is used as the split dimension this directly corresponds to the strategy applied with the VAMSplit R-tree However, for our test data sets with at most dimensions (cf. section the simple algorithm for the selection of the split dimension described above performed pretty well.

In addition to the described split strategy we employed the redistribution scheme presented in to avoid unnecessary bucket splits. Roughly spoken this scheme works as follows: If the sibling of the filled bucket is a bucket as well (and not a directory node) which is not yet filled, we shift one object from the filled bucket into its sibling and readjust the split line in the directory This way we can increase the bucket utilization.

Actual Data Regions

In the introductory sections of this paper we mentioned that one advantage of k d-tree-based access structures compared to R-trees is the fact that the size of the directory entries is independent of the number of dimensions in the data space. As a consequence, the fanout of the directory pages in k d-tree-based access structures remains equally high for high dimensional data. The price we have to pay for the small directory entries in k d-tree-based access structures is that the data space partition covers the whole data space. This can lead to unnecessary high priorities for entries in the auxiliary data structure NPQ when processing a nearest neighbor query On the other hand, we also mentioned that an advantage of the R-tree compared to k d-tree-based access structures is that parts of the data space which do not contain any data are not represented in the directory

To combine these advantages we store so-called *actual data regions* with each reference to a bucket or a directory page in the directory tree of the LSD^h-tree.

To become more precise, we call the data region of a directory node or bucket which is determined by the split lines on the path from the root of the directory to the directory node or bucket its *potential data region* We distinguish this potential data region from the *actual data region* which is the minimal bounding interval containing all points actually stored in the bucket or in case of a directory node in the corresponding subtree of the access structure.

If we would store the accurate information about the actual data region with each reference to a bucket or directory page in the directory tree we would end up with the same fanout limitations as the X-tree. Fortunately we can adapt a technique introduced with the buddy-tree and exploit the potential data regions to reduce the storage space requirements for the actual data regions significantly To this end, we decide how many storage space we are willing to spend for the actual data regions say z bits per bound in each dimension. Then we lay a grid with $^{z\ t}$ cells over the potential data region. Now the actual data region is extended to complete cells and the numbers of the bordering grid lines are maintained. Figure illustrates this for a two-dimensional data region and z For this actual data region we obtain the *coded actual data region* × which is stored together with the reference to the corresponding bucket or directory page.

Experimental results

In a series of tests we compared the LSD^h-tree with and without coded actual data regions and the X-tree. We performed tests with three different data sets:

RAND: A synthetic data set containing uniformly distributed points with dimensions.

TIGER: Feature vectors with dimensions for line segments from the Census TIGER database `http://www.census.gov` As features we used

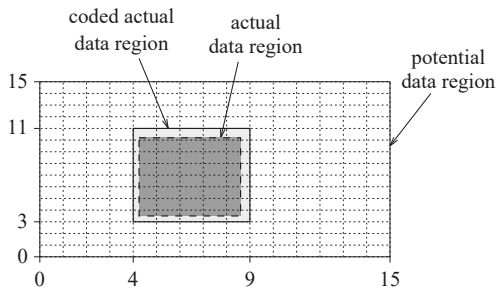

Figure Coding of an actual data region

e.g. the Line ID, the chain code, the start address and the block numbers. This way we achieved skewly distributed feature vectors.

CAR: A data set with feature vectors for car registrations. These vectors contained features such as a county shortcut, the category of the registered car, the length and the weight of the car, the year of the first registration of the car, the sex of the owner, etc.

For all structures we used a bucket capacity and a directory page size of bytes. For the LSD^h tree without coded actual data regions we allowed internal directory nodes and for the LSD^h-tree with coded actual data regions internal directory nodes to compensate the storage space requirements for the coded actual data regions in the internal directory

First we investigated the storage space requirements and the insertion times of the access structures. To this end, we used the dynamic variant of the X-tree as described in and inserted one feature vector after the other as we did with the LSD^h-tree. Table states the numbers of buckets and directory pages occupied by the access structures and the construction time (on a Sun UltraSPARC-1/170 with MB main memory). LSD^h means the tree with coded actual data regions while LSD^h means the tree without coded actual data regions. Two aspects might astonish at rst glance: The LSD^h-tree *with* coded actual data regions occupies more directory pages than the X-tree. This is a consequence of the relatively inflexible heap representation in the external directory pages, which results in a relatively low directory page utilization The construction times for the smaller

Obviously this low directory page utilization diminishes the advantage of the higher potential fanout of the directory pages of the LSD^h-tree *with* coded actual data regions compared to the X-tree. On the other hand, the fact that the potential fanout and the actual fanout decreases more slowly with higher dimensions for the LSD^h-tree *with* coded actual data regions compared to the SS-tree or the X-tree still remains true.

To overcome these problems with high insertion times bulk-loading techniques have been developed for the X-tree

data	tree	buckets	dir. pages	time (sec.)
RAND	LSD^h			
	LSD^h			
	X			
TIGER	LSD^h			
	LSD^h			
	X			
CAR	LSD^h			
	LSD^h			
	X			

Table Size and construction time of the structures

X-trees are only three times higher than the construction times of the LSD^h-tree with coded actual data regions, while the construction time for the X-tree with uniformly distributed feature vectors is seventy times higher. The reason for this phenomenon is the high effort for the maintenance of the supernodes, which become more frequent for large data sets

To assess the performance with similarity queries, we performed m-nearest neighbor queries with m m m m For the uniformly distributed objects we used randomly created query points. For the other data sets we used randomly chosen objects from the data set itself as query points. As a consequence for these data sets a -nearest neighbor query yields only the query point itself and a 2-nearest neighbor query yields the query point itself and its nearest neighbor We performed these queries for the LSD^h-tree with and without coded actual data regions, for an iteratively created X-tree (as described in and for an improved X-tree using a bulk-loading technique (as described in For these structures we have illustrated the numbers of page accesses (directory pages and buckets) in the figures and ("CADR" stands for "coded actual data regions"). For the uniformly distributed data set we included only one line for the X-tree, because both versions of the X-tree performed similar. To compensate the internal directory of the LSD^h-tree, we did not count page accesses in the two topmost levels of the X-tree.

For the uniformly distributed objects the performance of the LSD^h-tree with coded actual data regions is slightly better than the performance of the X-tree

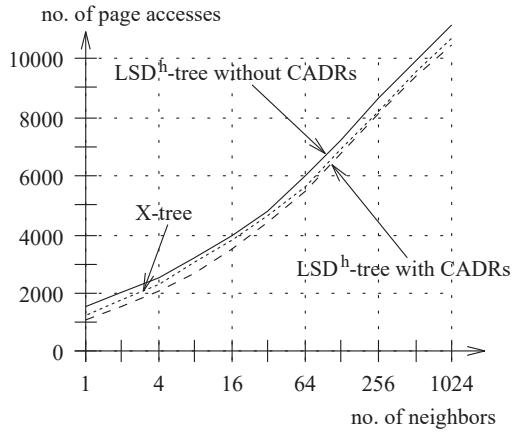

Figure Query performance with data set RAND

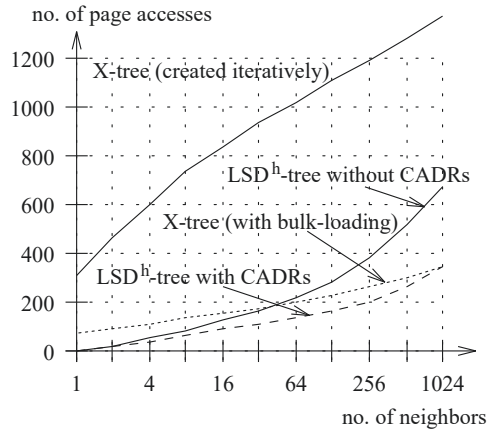

Figure Query performance with data set TIGER

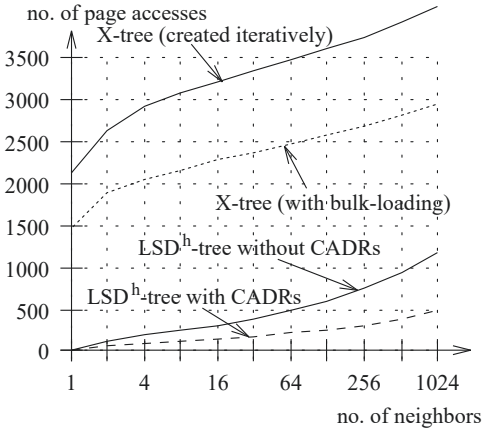

Figure Query performance with data set CAR

(between for m and for m and the performance of the LSD^h-tree without coded actual data regions is slightly worse than the performance of the X-tree. The situation becomes much more diverse for the two data sets with real data. For the data set CAR containing extremely skew distributed feature vectors both variants of the LSD^h tree outperform the X-tree by far. For the data set TIGER the LSD^h-tree still outperforms the X-tree for the first, most similar objects. However, when a large number of similar objects is needed m the LSD^h-tree without coded actual data regions requires more page accesses than the X-tree created by the bulk-loading technique.

To sum up, the experimental results demonstrate two points: A k d-tree-based access structure can achieve a query performance for high dimensional data which is at least competitive with R-tree-based access structures. Though the performance results for uniformly distributed 16-dimensional feature vectors have been rather bad (over page accesses to find the most similar object) the performance results for the two real data sets with 17-dimensions are rather encouraging.

Conclusion

We have presented the LSD^h-tree as an adaptation of the spatial LSD-tree for high dimensional feature vectors. The adaptations comprise a new split strategy and the use of coded actual data regions. The presented experimental results demonstrate the qualities of this access structure especially for real data sets. It turned out that even the variant without coded actual data regions has a rather competitive performance. This is noteworthy especially because the fanout of a directory page of this variant of the LSD^h-tree is independent of the number of dimensions in the feature space. Therefore the structure can be used for rather high dimensional data as long as the distribution characteristics of the data allow for an efficient query processing.

Our future research activities include tests with further real data sets, the consideration of other similarity measures, and other operations like similarity joins.

Acknowledgments. We are thankful to S. Berchtold, D.A. Keim, and H.-P Kriegel for making the implementation of the X-tree available to us.

References

N. Beckmann, H.-P Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Conf.* pages Atlantic City N.J., USA,

J.L. Bentley Multidimensional binary search trees used for associative searching. *Communications of the ACM*

S. Berchtold, C. Böhm, D.A. Keim, and H.-P Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. 16th ACM Symposium on Principles of Database Systems* pages Tucson, Arizona,

S. Berchtold, C. Böhm, and H.-P Kriegel. Improving the query performance of high-dimensional index structures by bulk-load operations. submitted for publication,

S. Berchtold, D.A. Keim, and H.-P Kriegel. The X tree: An index structure for high-dimensional data. In *Proc. 22th Intl. Conf. on VLDB* pages Mumbai Bombay), India,

M. Freeston. The BANG file: a new kind of grid file. In *Proc. ACM SIGMOD Conf.* pages San Francisco, Cal., USA,

J.H. Friedman, J.L. Bentley and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Software* September

A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf.* pages Boston, Mass., USA,

A. Henrich. A distance-scan algorithm for spatial access structures. In *Proc. 2nd ACM Workshop on Advances in Geographic Information Systems* pages Gaithersburg, Md., USA,

A. Henrich. Adapting a spatial access structure for document representations in vector space. In *Proc. 5th Intl. Conf. on Information and Knowledge Management* pages Rockville, Md., USA,

A. Henrich. Improving the performance of multi-dimensional access structures based on k-d-trees. In *Proc. IEEE Intl. Conf. on Data Eng.* pages New Orleans, La., USA,

A. Henrich and H.-W. Six. How to split buckets in spatial data structures. In *Proc. Intl. Conf. on Geographic Database Management Systems* Esprit Basic Research Series DG XIII pages Capri,

A. Henrich, H.-W. Six, and P Widmayer. The LSD-tree: spatial access to multidimensional point and non point objects. In *Proc. 15th Intl. Conf. on VLDB* pages Amsterdam,

G.R. Hjaltason and H. Samet. Ranking in spatial databases. In *Proc. 4th Intl. Symposium on Advances in Spatial Databases (SSD'95)*,v olume of *LNiCS* pages Portland, ME, USA,

N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD Conf.* pages Tucson, Arizona, USA,

M.J. van Kreveld and M.H. Overmars. Divided k-d trees. *Algorithmica*

K.-I. Lin, H.V. Jagadish, and C Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal* October

D.B. Lomet and B. Salzberg. A robust multi-attribute search structure. In *Proc. IEEE 5th Intl. Conf. on Data Engineering* pages Los Angeles,

J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid le: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*

B.C. Ooi, K.J. McDonell, and R Sacks-Davis. Spatial kd-tree: An indexing mechanism for spatial databases. *IEEE COMPSAC* pages

Mireille Regnier. Analysis of grid le algorithms. *BIT*

J.T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD Conf.* pages

N. Roussopoulos, S. Kelley and F. Vincent. Nearest neighbor queries. In *Proc. ACM SIGMOD Conf.* pages San Jose, Cal., USA,

G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. *Communications of the ACM* November

B. Seeger and H.-P Kriegel. The buddy-tree: an efficient and robust access method for spatial data base systems. In *Proc. 16th Intl. Conf. on VLDB* pages Brisbane, Australia,

T. Sellis, N Roussopoulos, and C. Faloutsos. The R+ tree: a dynamic index for multi-dimensional objects. In *Proc. 13th Intl. Conf. on VLDB* pages Brighton, England,

V.S. Subrahmanian and S. Jajodia, editors. *Multimedia Database Systems: Issues and Research Directions* Springer, Berlin, Heidelberg,

D.A. White and R. Jain. Similarity indexing: Algorithms and performance. In *Proc. Storage and Retrieval for Image and Video Databases IV (SPIE)*,v olume pages San Diego, CA, USA,

D.A. White and R Jain. Similarity indexing with the SS-tree. In *Proc. 12th Intl. Conf. on Data Engineering* pages New Orleans, La., USA,