

Generalized Density-Based Clustering for Spatial Data Mining

Dissertation im Fach Informatik
an der Fakultät für Mathematik und Informatik
der Ludwig-Maximilians-Universität München

von

Jörg Sander

Tag der Einreichung: 21.09.1998

Tag der mündlichen Prüfung: 10.12.1998

Berichterstatter:

Prof. Dr. Hans-Peter Kriegel, Ludwig-Maximilians-Universität München
Prof. Raimond Ng Ph.D., University of British Columbia, Canada

Acknowledgments

I would like to thank all the people who supported me in the development of this thesis.

First of all, I would like to thank Prof. Dr. Hans-Peter Kriegel, my supervisor and first referee of this thesis. He made this work possible by providing his excellent technical expertise and the organizational background which resulted in an inspiring and motivating working atmosphere. I would also like to thank Prof. Dr. Raymond Ng for his interest in my work and his willingness to act as the second referee.

This work could not have grown without the cooperation of my colleagues in the KDD group at Prof. Kriegels chair, Martin Ester and Xiaowei Xu. They deserve my very special thanks for the inspiring and productive teamwork that I could take part in.

Many fruitful discussions which brought this work forward took place also with other colleagues - in alphabetical order: Mihael Ankerst, Christian Boehm, Bernhard Braunmüller, Markus Breunig and Thomas Seidl and I thank them all.

I also appreciate the substantial help of the students whose study thesis and diploma thesis I supervised, especially Florian Krebs and Michael Wimmer.

Last but not least, I would like to thank Susanne Grienberger for carefully reading the manuscript and eliminating some major roughness in my English.

Munich, September 1998

Table of Contents

1 Introduction	1
1.1 Knowledge Discovery in Databases, Data Mining and Clustering	2
1.2 Outline of the thesis	6
2 Related Work	11
2.1 Efficient Query Processing in Spatial Databases	12
2.2 Clustering and Related Algorithms	20
2.2.1 Hierarchical Clustering Algorithms	20
2.2.2 Partitioning Clustering Algorithms	22
2.2.3 Region Growing	31
2.3 Exploiting the Clustering Properties of Index Structures	32
2.3.1 Query Support for Clustering Algorithms	32
2.3.2 Index-Based Sampling	34
2.3.3 Grid clustering	35
2.3.4 CF-Tree	37
2.3.5 STING	38
2.4 Summary	40
3 Density-Based Decompositions	41
3.1 Density-Connected Sets	42
3.1.1 Motivation	42
3.1.2 Definitions and Properties	44
3.2 Generalized Clustering and Some Specializations	54
3.2.1 Density-Based Decompositions	54
3.2.2 Specializations	57
3.3 Determining Parameters	67
3.4 Summary	73

4 GDBSCAN: An Algorithm for Generalized Clustering	75
4.1 Algorithmic Schema GDBSCAN	76
4.2 Implementation	80
4.3 Performance	86
4.3.1 Analytical Evaluation	86
4.3.2 Experimental Evaluation	88
4.4 Database Support for GDBSCAN	92
4.4.1 Neighborhood Indices	94
4.4.2 Multiple Neighborhood Queries	102
4.5 Summary	112
5 Applications	113
5.1 Earth Science (5d points)	114
5.2 Molecular Biology (3d points)	117
5.3 Astronomy (2d points)	119
5.4 Geography (2d polygons)	122
5.5 Summary	128
6 Incremental GDBSCAN	129
6.1 Motivation	130
6.2 Affected Objects	134
6.3 Insertions	139
6.4 Deletions	142
6.5 Implementation	145
6.6 Performance Evaluation	149
6.7 Summary	156
7 Hierarchical GDBSCAN	157
7.1 Nested Density-Based Decompositions	158
7.1.1 Motivation	158
7.1.2 Definitions and Properties	160

7.2	Algorithm H-GDBSCAN	167
7.2.1	Multiple Clustering Levels	168
7.2.2	Ordering the Database with respect to Cluster Structure	183
7.3	Summary	198
8	Conclusions	199
	References	205
	Index	217
	List of Definitions	221
	List of Figures	223

Chapter 1

Introduction

This chapter shortly introduces the context of this thesis which contributes to the field of spatial data mining, especially to the task of automatically grouping objects of a spatial database into meaningful subclasses. In section 1.1, the connection between the notions *Knowledge Discovery in Databases*, *(Spatial) Data Mining* and *Clustering* is elaborated. Section 1.2 describes the goal and gives an outline of this thesis.

1.1 Knowledge Discovery in Databases, Data Mining and Clustering

Both, the number of databases and the amount of data stored in a single database are growing rapidly. This is true for almost any type of database such as traditional (relational) databases, multimedia or spatial databases. Spatial databases are, e.g., databases for geo-marketing, traffic control, environmental studies or sky and earth observation databases. The accelerated growth of such databases by far exceeds the human capacity to analyze the data. For instance, databases on sky objects consist of billions of entries extracted from images generated by large telescopes. The NASA Earth Observing System, for example, is projected to generate some 50 GB of remotely sensed data per hour.

Classical analysis methods are in general not well suited for finding and presenting implicit regularities, patterns, dependencies or clusters in today's databases. Important reasons for the limited ability of many statistical methods to support analysis and decision making are the following:

- They do not scale to large data volumes (large number of rows/entries, large number of columns/dimensions) in terms of computational efficiency.
- They assume stationary data which is not very common for real-life databases. Data may change and derived pattern may become invalid. Then all patterns derived from the data have to be calculated from scratch.
- Modeling in the large requires new types of models that describe pattern in the data at different scales (e.g. hierarchical).

For these reasons, in the last few years new computational techniques have been developed in the emerging research field of *Knowledge Discovery in Databases (KDD)*. [FPS 96] propose the following definition of KDD:

Knowledge Discovery in Databases is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data.

The KDD process is an interactive and iterative process, involving numerous steps including preprocessing of the data, applying a data mining algorithm to enumerate patterns from it, and the evaluation of the results ([BA 96]), e.g.:

- *Creating a target data set*: selecting a subset of the data or focusing on a subset of attributes or data samples on which discovery is to be performed.
- *Data reduction*: finding useful features to represent the data, e.g., using dimensionality reduction or transformation methods to reduce the number of variables under consideration or to find invariant representations for the data.
- *Data mining*: searching for patterns of interest in the particular representation of the data: classification rules or trees, association rules, regression, clustering, etc.
- *Interpretation of results*: this step can involve visualization of the extracted patterns or visualization of the data given the extracted models. Possibly the user has to return to previous steps in the KDD process if the results are unsatisfactory.

For a survey of industrial applications of KDD see [PBK+ 96], and for applications in science data analysis see [FHS 96].

The core step of the KDD process is the application of a data mining algorithm. Hence, the notions “KDD” and “data mining” are often used in the same way. Actually, most of the research conducted on knowledge discovery in *relational* as well as *spatial* databases is about data mining algorithms (*c.f.* [CHY 96] for a survey on algorithms for knowledge discovery in relational databases, and [KAH 96] for an overview on knowledge discovery in spatial databases). The following broad definition of data mining can be found in [FPS 96a]:

Data mining is a step in the KDD process consisting of applying data analysis algorithms that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the data.

The different data mining algorithms that have been proposed in the literature can be classified according to the following primary data mining methods ([MCP 93], [FPS 96a]):

- *Clustering*: identifying a set of categories or clusters to describe the data.
- *Classification*: learning a function that maps (classifies) a data item into one of several predefined classes.
- *Regression*: learning a function that maps a data item to a real-valued prediction variable and the discovery of functional relationships between variables.
- *Summarization*: finding a compact description for a subset of data.
- *Dependency Modeling*: finding a model which describes significant dependencies between variables (e.g., learning of belief networks).
- *Change and Deviation Detection*: discovering the most significant changes in the data from previously measured or normative values.

Spatial data mining is data mining applied to spatial data, i.e. data for which at least a distance function between objects is defined. Typically, some attributes specify a location and possibly an extension in some d -dimensional space, for objects such as points, lines or polygons. The objects may additionally have other non-spatial attributes.

Spatial data is typically stored and managed in spatial database systems (SDBS) (see [Gue 94] for an overview). Applications of data mining algorithms to spatial databases are important, e.g., for geo-marketing, traffic control or environmental studies.

In [LHO 93], attribute-oriented induction is performed by using (spatial) concept hierarchies to discover relationships between spatial and non-spatial attributes. A spatial concept hierarchy represents a successive merging of neighboring regions into larger regions. In [NH 94], the clustering algorithm CLARANS, which groups neighboring objects automatically without a spatial concept hierarchy, is combined with attribute-oriented induction on non-spatial attributes. [KH 95] introduces spatial association rules which describe associations between objects based on different spatial neighborhood relations. [Ng 96] and [KN 96] present algorithms to detect properties of clusters using reference maps and thematic maps. For instance, a cluster may be explained by the existence of certain neighboring objects which may “cause” the existence of the cluster. New algorithms for spatial characterization and spatial trend analysis are sketched in [EKS 97] and elaborated in [EFKS 98]. For spatial characterization, it is important that class membership of a database object is not only determined by its non-spatial attributes but also by the attributes of objects in its neighborhood. In spatial trend analysis, patterns of change of some non-spatial attribute(s) in the neighborhood of a database object are determined. A more comprehensive overview of spatial data mining can be found in [KHA 96].

We observe that a lot of work on *spatial* data mining deals with clustering.

Clustering is the task of grouping the objects of a database into meaningful subclasses - either as a stand alone task or in combination with some other data mining algorithms which operate on detected clusters.

Applications of clustering in spatial databases are, e.g., the detection of seismic faults by grouping the entries of an earthquake catalog [AS 91], the creation of thematic maps in geographic information systems by clustering feature spaces [Ric 83] and detection of clusters of objects in geographic information systems and to explain them by other objects in their neighborhood ([NH 94] and [KN 96]). An application to a more abstract “spatial” database is the clustering of a WWW-log

database to discover groups of similar access patterns for a Web server which may correspond to different user profiles.

Clustering has been studied in statistics (e.g. [And 73], [Eve 81], [Har 75], [JD 88], [KR 90]), machine learning (e.g. [CKS 88], [Fis 87], [Fis 95], [FPL 91]), and recently in the context of KDD (e.g. [EKX 95a], [NH 94], [SEKX 98], [WYM 97], [XEKS 98] and [ZRL 96]). The reasons for the new database-oriented clustering methods have already been indicated: The well-known clustering algorithms from statistics such as k -means [Mac 67], k -medoids [KR 90] or Single Link Clustering [Sib 73] are too inefficient on large databases and they also assume that all objects to be clustered can reside in main memory at the same time. Despite growing main memories, this assumption is not always true for large databases. Additionally, data mining in real-world database creates new challenges for clustering algorithms. These kinds of databases may be highly dynamic and/or the objects may be defined by data types other than numeric - properties which are usually neglected by traditional clustering approaches.

1.2 Outline of the thesis

This thesis contributes to the field of spatial data mining, especially to the task of clustering, i.e. automatically grouping the objects of a spatial database into meaningful subclasses.

Starting from a density-based clustering approach for point objects (presented in [EKSX 96]), we develop a general method to decompose a database into a set of cluster-like components. This method is applicable to objects of arbitrary data type provided only that there is (1) a binary (neighborhood) predicate for objects which is symmetric and reflexive and there is (2) a predicate that allows the user to determine whether or not a set of objects has a “minimum weight”.

Our method for density-based decompositions relies on a formal set-theoretic framework of density-connected sets which generalizes the results of different clustering methods and similar techniques like region growing algorithms in the following sense: The results of these techniques for grouping objects of a database can be described as special cases of density-based decompositions. Thus, they do have the same underlying formal structure and can be produced by the same algorithmic schema. Furthermore, density-based decompositions have the following nice properties which are important for spatial data mining in real-world databases:

- It is possible to perform incremental updates on density-based decompositions very efficiently in a dynamic environment of insertions and deletions.
- Hierarchical descriptions of the underlying grouping of objects in a database are possible by extending the basic algorithmic schema without a significant loss in performance.
- The algorithmic schema can be supported very efficiently by the query processing facilities of a spatial database system.

The theoretical foundations and algorithms concerning these tasks are elaborated in this thesis which is organized as follows:

After this introduction, related work on database oriented clustering techniques is reviewed in chapter 2. For that purpose, methods to support efficient query processing in spatial database systems are sketched. We also show how to integrate clustering algorithms with spatial database management systems and present the most recent clustering techniques from the KDD literature which essentially exploit clustering properties of spatial index structures.

In chapter 3, a motivation for the generalization of density-based clustering is presented. After that, the notions “density-connected set” and “density-based decomposition”, i.e. a generalized density-based clustering, are defined and important specializations of these notions are discussed. These specializations include

density-based clustering, clustering levels produced by the well-known single link clustering method, results of simple forms of region growing algorithms as well as new applications which may be appropriate for grouping spatially extended objects such as polygons in geographic information systems. The task of determining the parameters for certain specializations of the algorithm is also discussed in this chapter.

In chapter 4, our algorithmic schema GDBSCAN to compute density-connected sets is introduced and some implementation issues are discussed. The performance is evaluated analytically for the algorithmic schema and experimentally for the most important specialization of GDBSCAN, i.e. DBSCAN [EK SX 96]. In the experimental evaluation, the performance of DBSCAN is compared with the performance of some newer clustering algorithms proposed in the KDD literature. The implementation of DBSCAN used for this comparison is based on a particular spatial index structure, the R*-tree. A discussion of different methods to support the construction of a density-based decomposition concludes this chapter. The most important technique is a new query type called “multiple neighborhood query”. We will show that multiple neighborhood queries are applicable to a broad class of spatial data mining algorithms, including GDBSCAN, to speed up the performance of these algorithms significantly.

In chapter 5, four typical applications of our algorithm are presented in more detail. First, we present a “standard” clustering application for the creation of a land-use map by clustering 5-dimensional feature vectors extracted from several satellite images. Second, 3-dimensional points on a protein surface are clustered, using also non-spatial attributes, to extract regions with special properties on the surface of the protein. Third, a special instance of our algorithm is applied to 2-dimensional astronomical image data, performing ‘region growing’ to detect celestial sources from these images. In the last application, GDBSCAN is used to detect influence regions for the analysis of spatial trends in a geographic information system on Bavaria. This application demonstrates how sophisticated neighborhood predicates

utilizing spatial and non-spatial attributes of the data can be used to detect interesting groups of spatially extended objects such as polygons representing communities.

In chapter 6, we show that updates on a database affect a density-based decomposition only in a small neighborhood of inserted or deleted objects. We present incremental versions of our algorithm for updating a density-based decomposition on insertions and deletions. A cost model for the performance of *Incremental GDBSCAN* is presented and validated by using synthetic data as well as real data from a WWW-log database showing that clustering in a dynamic environment can be handled very efficiently.

The basic algorithm GDBSCAN determines only a single level clustering in a single run of the algorithm. In chapter 7 this basic algorithm is extended such that hierarchical layers of clusterings can be computed very efficiently. Hierarchical clusterings can be described easily by “nested” density-based decompositions. The efficiency of *Hierarchical GDBSCAN* is due to the fact that the costs for computing nested or flat density-connected sets are nearly the same. Starting from the hierarchical version of GDBSCAN a second algorithm is developed for distance based neighborhoods. In this algorithm a maximal distance d_{max} is used to produce an ordering of the database with respect to its clustering structure. Storing few additional information for each object in this ordering allows a fast computation of every clustering level with respect to a smaller distance than d_{max} . However, a “cluster-ordering” of the database can as well be used as a stand-alone tool for cluster analysis. A visualization of the cluster-ordering reveals the cluster structure of a data set of arbitrary dimension in a very comprehensible way. Furthermore, the method is rather insensible to input parameters.

Chapter 8 concludes the thesis with a short summary and a discussion of future work.

Chapter 2

Related Work

The well-known clustering algorithms have some drawbacks when applied to large databases. First, they assume that all objects to be clustered reside in main memory. Second, these methods are too inefficient when applied to large databases. To overcome these limitations, new algorithms have been developed which are surveyed in this chapter. Most of these algorithms (as well as our own approach) utilize spatial index structures. Therefore, we first give a short introduction to efficient query processing in spatial databases (section 2.1). Then, we survey clustering algorithms and show how to integrate some of them into a database management system for the purpose of data mining in large databases (section 2.2). Furthermore, we discuss recently introduced methods to exploit the (pre-)clustering properties of spatial index structures (section 2.3) to derive clustering information about large databases. This chapter is a major extension of a similar overview given in [EK SX 98].

2.1 Efficient Query Processing in Spatial Databases

Numerous applications, e.g., geographic information systems and CAD systems, require the management of *spatial* data. We will use the notion *spatial data* in a very broad sense. The space of interest may be an abstraction of a real two- or three-dimensional space such as a part of the surface of the earth or the geometric description of a protein as well as a so called high-dimensional “feature space” where characteristic properties of the objects of an application are represented by the different values of a high-dimensional feature vector. Basic two-dimensional data-types, e.g., are points, lines and regions. These notions are easily extended to the general d -dimensional case. Although most research on spatial databases is considering d -dimensional vector spaces, we will not restrict the notion *spatial* to this case. We say that a database is a spatial database if at least a distance metric is defined for the objects of the database, i.e if the space of interest is at least a metric space.

A *spatial database system (SDBS)* is a database system offering spatial data-types in its data model and query language and offering an efficient implementation of these data-types with their operations and queries [Gue 94]. Typical operations on these data-types are the calculation of the distance or the intersection. Important query types are similarity queries, e.g.:

- *region queries*, obtaining all objects within a specified query region and
- *k-nearest neighbor (kNN)* queries, obtaining the k objects closest to a specified query object.

Similarity queries are important building blocks for many spatial data mining algorithms - especially for our approach to ‘generalized clustering’. Therefore, the underlying *SDBS* technology, i.e. spatial index structures, to support similarity queries efficiently, is sketched briefly in the following.

A trivial implementation of the spatial queries would scan the whole database and check the query condition on each object. In order to speed up query processing, many spatial index structures have been developed to restrict the search to the relevant part of the space (for a survey see, e.g., [Gue 94] or [Sam 90]). All index structures are based on the concept of a *page*, which is the unit of transfer between main and secondary memory. Typically, the number of page accesses is used as a cost measure for database algorithms because the run-time for a page access exceeds the run-time of a CPU operation by several orders of magnitude.

Spatial index structures can be roughly classified as organizing the data space (hashing) or organizing the data itself (search trees). In the following, we will introduce well-known representatives which are typical for a certain class of index structures and which are used in the following sections and chapters.

The *grid file* [NHS 84] has been designed to manage points in some d -dimensional data space, generalizing the idea of one-dimensional hashing. It partitions the data space into cells using an irregular grid. The split lines extend through the whole space and their positions are kept in a separate scale for each dimension. The d scales define a d -dimensional array (the *directory*) containing a pointer to a page in each cell. All d -dimensional points contained in a cell are stored in the respective page (*c.f.* figure 1, left). In order to achieve a sufficient storage utilization of the

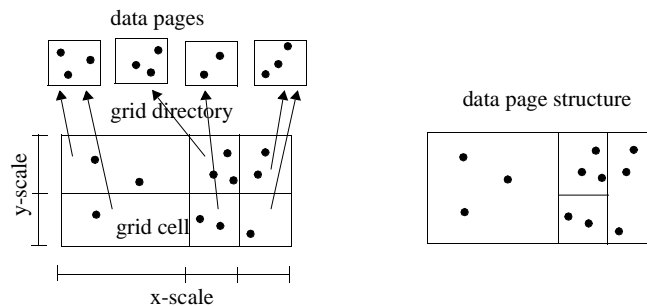


Figure 1: Illustration of the grid file

secondary memory, several cells of the directory may be mapped to the same data page. Thus, the data space is actually divided according to the data page structure of the grid file (*c.f.* figure 1, right). Region queries can be answered by determining from the directory the set of grid cells intersecting the query region. Following the pointers yields a set of corresponding data pages, and the points in these pages are then examined. A drawback of this method is that the number of grid cells may grow super-linear in the number of objects N , depending on the distribution of the objects.

The *R-tree* [Gut 84] generalizes the one-dimensional B-tree to d -dimensional data spaces, specifically an R-tree manages d -dimensional hyper-rectangles instead of one-dimensional numeric keys. An R-tree may organize extended objects such as polygons using *minimum bounding rectangles (MBR)* as approximations as well as point objects as a special case of rectangles. The leaves store the MBR of data objects and a pointer to the exact geometry if needed, e.g. for polygons. Internal nodes store a sequence of pairs consisting of a rectangle and a pointer to a child node. These rectangles are the MBRs of all data or directory rectangles stored in the subtree having the referenced child node as its root (*c.f.* figure 2).

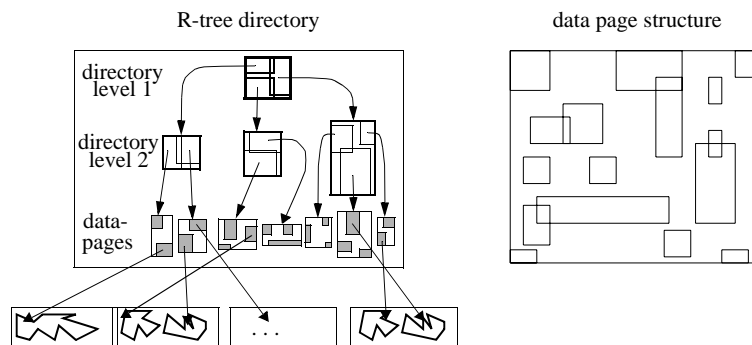


Figure 2: Illustration of the R-tree

To answer a region query, the set of rectangles intersecting the query region is determined recursively starting from the root. In a directory node, the entries intersecting the query region are determined and then their referenced child nodes are searched until the data pages are reached.

The MBRs in a directory node may overlap. They may also cover large fractions of the data space where no objects are located. Both properties do have a negative impact on the performance of query processing since additional paths in the tree must be searched. Especially the split strategy (choice of split axis and choice of split value) in case of an overflow of a page has the most significant effect on the performance of the R-tree. Therefore, different split strategies have been proposed, for instance the *R*-tree* [BKSS 90], for minimizing the overlap and coverage of empty space in the directory of the tree.

The grid file as well as the R-tree and their variants are efficient only for relatively small numbers of dimensions d . The average upper bound for d using these index structures is about 8, but the actual value also depends on the distribution of the data. The better the data is clustered the more dimensions can be managed efficiently.

It is a result of recent research activities ([BBKK 97], [BKK 96]) that basically none of the known querying and indexing techniques perform well on high-dimensional data for larger queries - under the assumption of uniformly distributed data. This due to some unexpected effects in high-dimensional space. For instance, the side length of a query grows dramatically with increasing dimension for hypercube range queries which have a constant selectivity (i.e. relative volume). Thus, the probability of an intersection of the query cube with a directory or data rectangle in the known index structures approaches 1 with increasing dimension d .

Following the ideas of search trees, index structures have been designed recently which are also efficient for some larger values of d ([BKK 96], [LJF 95]). For instance, the X-tree ([BKK 96]) is similar to an R*-tree but introduces the concept of

supernodes, i.e. nodes of variable size in the directory of the tree (see figure 3). The basic idea is to avoid computational overhead in the directory while performing similarity queries. Directory nodes are “merged” into one supernode (actually, directory nodes are *not* split) if there is a high probability that all parts of the node have to be searched anyway for most queries.

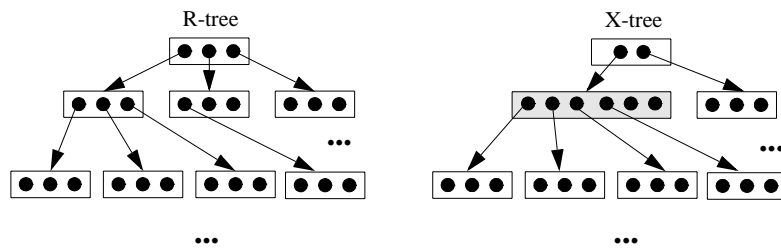


Figure 3: Comparison of R-tree and X-tree structure

However, this approach will perform better than a linear scan over all data objects only for values of $d \leq 16$ on the average - again, depending on the distribution of the data. These values of d are still moderate with regard to many applications.

Up to now, the indexing method which performs best for high-dimensional data, also for values of d significantly larger than 16, seems to be the pyramid-technique proposed recently in [BBK 98]. However, this method is highly specialized to rectangular shaped region queries.

The pyramid method consists of two major steps: First, an unconventional pyramidal partitioning strategy is applied in order to increase the chance that pages are not intersected by a query hypercube in a high-dimensional data space (see figure 4, left). Second, the d -dimensional points p are transformed to one-dimensional *pyramid values* pv_p (see figure 4, right) which are then managed by a traditional one-dimensional index-structure, the B^+ -tree. A d -dimensional rectangular shaped range query is first transformed into one-dimensional queries selecting pyr-

amid values. This yields a set of candidates, and a “point in rectangle test” is performed for each candidate to determine the answers to the original query.

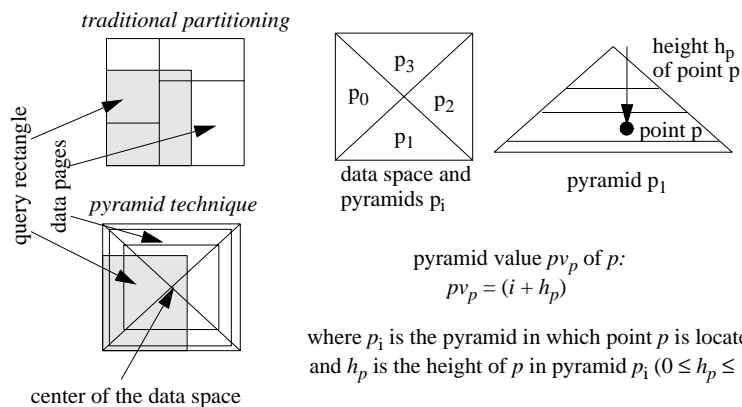


Figure 4: Illustration of the pyramid technique

This approach is primarily intended to be used for hypercube range queries and for this case outperforms all other query processing methods. However, at this time it is not obvious how to extend this technique to support other types of queries, e.g., nearest neighbor queries.

To support similarity queries in general metric spaces, all of the above methods are not applicable since in the general case we only have a function to measure the distance between objects. If the distance function is a metric, so-called *metric trees* (see e.g. [Uhl 91]) can be used for indexing the data. Metric trees only consider relative distances between objects to organize and partition the search space. The fact that the *triangle inequality* property applies to a metric distance function can be used to prune the search tree while processing a similarity query. Most of these structures are, however, static in the sense that they do not allow dynamic insertions and deletions of objects. A recent paper ([CPZ 97]) has introduced a dynamic

metric index structure, the *M-tree*, which is a balanced tree that can be managed on the secondary memory.

The leaf nodes of an M-tree store all the database objects. Directory nodes store so-called *routing objects* which are selected database objects to guide the search operations. Associated with each routing object O_r , are: a pointer to a subtree $T(O_r)$ of O_r , a covering radius $r(O_r)$ and the distance $d(O_r, O_p)$ of O_r to its parent object O_p (see figure 5, right). All objects in the subtree of O_r are within the distance $r(O_r)$ from O_r , $r(O_r) > 0$. A range query $range(Q, r(Q))$ starts from the root of the tree and recursively traverses all paths which cannot be excluded from leading to objects O_j satisfying the condition $d(O_j, Q) \leq r(Q)$. The pruning criterion for excluding subtrees from the search is depicted in figure 5, left. Obviously, the performance of this indexing technique is dependent on the distribution of distances between the database objects.

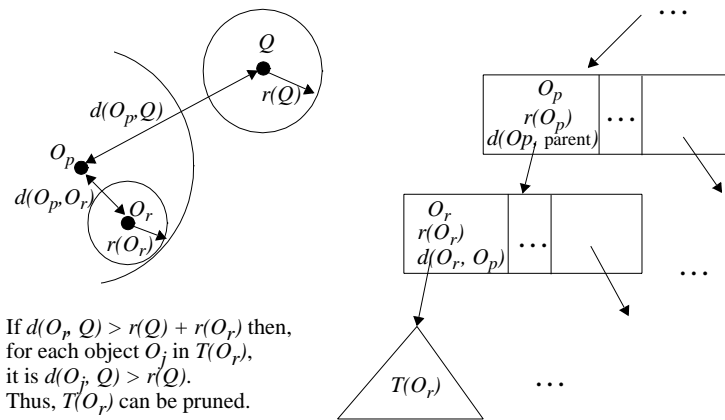


Figure 5: Illustration of the M-tree

Recently, the VA-file [WSB 98] was developed, an index structure that actually is *not* an index structure. The authors prove in the paper that under the assumption

of uniformly distributed data, above a certain dimensionality no index structure can process a nearest neighbor query efficiently. Thus, they suggest to use the sequential scan which obtains at least the benefits of sequential rather than random disk I/O. Clever bit encodings of the data are also devised to make the scan go faster.

The basic idea of the VA-file is to keep two files: a bit-compressed version of the points and the exact representation of the points. Both files are unsorted, however, the ordering of the points in the two files is identical. Bit encodings for the data points are generated by partitioning the space using only a few split lines in each dimension. Then, a point is represented by the the grid cell in which it is contained. This requires only a few bits for the coordinate of the cell in each dimension. Query processing is equivalent to a sequential scan of the compressed file with some look-ups to the second file whenever this is necessary. In particular a look-up occurs, if a point cannot be pruned from a search based only on the compressed representation. Note that a VA-file may perform worse than a true spatial index structure even in high-dimensional space if too many points share the same bit representation. This will be the case if the data is highly skewed and there exist high-density clusters.

As the VA-file is a very simple structure, there are two major problem associated with the VA-file: in case of correlations or clusters, many points in the database will share a single compressed representation and therefore, the number of look-ups will increase dramatically and second, the authors do not even provide a rule of thumb how to determine a good or optimal number of bits to be used for quantization.

Bearing their limitations in mind, spatial index structures and their query methods can nevertheless be used to improve the performance of some clustering algorithms. It is also possible to build clustering algorithms “on top” of index(-like) structures since index structures already perform some kind of pre-clustering of the data. We will focus on these aspects in our review of clustering and related algorithms in the next two sections.

2.2 Clustering and Related Algorithms

Several types of clustering algorithms can be distinguished. One well-known distinction is that of hierarchical and partitioning clustering algorithms [JD 88]. Hierarchical clustering methods organize the data into a nested sequence of groups. These techniques are important for biology, social, and behavioral sciences because of the need to construct taxonomies. Partitioning clustering methods try to recover natural groups in the data and thus construct only a single level partition of the data. Single partitions are more important in engineering applications and are especially appropriate for the efficient representation and compression of large data sets.

Somehow related to clustering are region growing algorithms which are used for image segmentation of raster images. We will see in the next chapter that the groups of pixels constructed by region growing algorithms are connected components that have the same underlying formal description as density-based clusters.

2.2.1 Hierarchical Clustering Algorithms

Whereas partitioning algorithms obtain a single level clustering, *hierarchical algorithms* decompose a database D of n objects into several levels of nested partitionings (clusterings). The hierarchical decomposition is represented by a *dendrogram*, a tree that iteratively splits D into smaller subsets until each subset consists of only one object. In such a hierarchy, each node of the tree represents a cluster of D . The dendrogram can either be created from the leaves up to the root (*agglomerative approach*) or from the root down to the leaves (*divisive approach*) by merging resp. dividing clusters at each step.

Hierarchical algorithms need only a dissimilarity matrix for objects as input. If a single level, i.e. a natural grouping in the data, is needed, a *termination condition* can be defined indicating when the merge or division process should be terminated.

One example of a break condition is a critical distance D_{min} . If no distance between two clusters of Q is smaller than D_{min} , then the construction algorithm for the dendrogram stops. Alternatively, an appropriate level in the dendrogram can be selected manually after the creation of the whole tree.

There are a lot of different algorithms producing the same hierarchical structure. Agglomerative hierarchical clustering algorithms, for instance, basically keep merging the closest pairs of objects to form clusters. They start with the “disjoint clustering” obtained by placing every object in a unique cluster. In every step the two “closest” clusters in the current clustering are merged. The most commonly used hierarchical structures are called “single link”, “complete link”, and “average link”, differing in principle only in the definition of the dissimilarity measure for clusters (see figure 6 for an illustration of the single link method):

- single link: $sim-sl(X, Y) = \inf_{x \in X, y \in Y} \{distance(x, y)\}$
- complete link: $sim-cl(X, Y) = \sup_{x \in X, y \in Y} \{distance(x, y)\}$
- average link: $sim-al(X, Y) = \frac{1}{|X| \cdot |Y|} \times \sum_{x \in X, y \in Y} distance(x, y)$ ¹

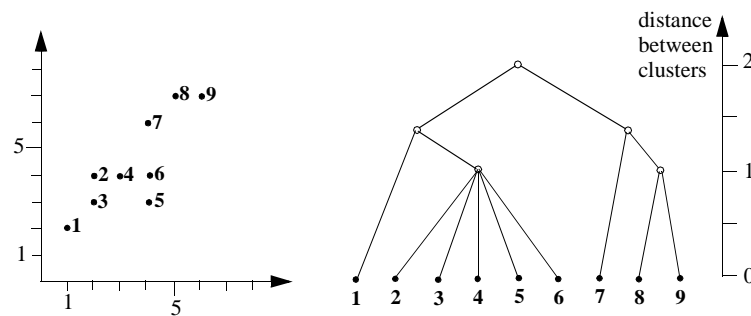


Figure 6: Single link clustering of n = 9 objects

1. Throughout this thesis, the expression $|X|$ denotes the number of elements in X if the argument X of $|\cdot|$ is as set.

Different algorithms, e.g. for the single-link method have been suggested (see e.g. [Sib 73], [JD 88], [HT 93]). The single link hierarchy can also be derived from the *minimum spanning tree* (MST)¹ of a set of points (see [Mur 83] for algorithms constructing the MST and [Rol 73] for an algorithm which transforms the MST into the single-link hierarchy). However, hierarchical algorithms are in general based on the inter-object distances and on finding the nearest neighbors of objects and clusters. Therefore, the run-time complexity of these clustering algorithms is at least $O(n^2)$, if all inter-object distances for an object have to be checked to find its nearest neighbor.

2.2.2 Partitioning Clustering Algorithms

Partitioning algorithms construct a partition of a database D of n objects into a set of k clusters where k may or may not be an input parameter. The objective of a partitioning clustering method is to determine a partition of the set of objects into k groups such that the objects in a cluster are more similar to each other than to objects in different clusters. However, there are a lot of alternatives to state this problem more precisely, i.e. to state formally what should be considered as a cluster and what should be considered as a “good” partitioning. Global as well as local clustering criteria are possible. In this section, we present only the basic ideas of the most common partitioning clustering methods.

Optimization Based Approaches

Optimization based clustering algorithms typically adopt a global clustering criterion. A global clustering criterion represents each cluster by a prototype and assigns an object to the cluster represented by the most similar prototype, i.e. to the prototype that has the smallest distance to the considered object. An iterative control strategy is used to optimize a notion of clustering quality such as the average

1. The use of the MST to find clusters is not restricted to the single link hierarchy. There are also partitioning clustering algorithms based on graph theory which use the MST directly (see the next section).

distances or the squared distances of objects to its prototypes. Depending on the kind of prototypes, we can distinguish so-called k -means, k -modes and k -medoid clustering algorithms.

For k -means algorithms (see e.g. [Mac 67]) each cluster is represented by a prototype computed as the mean value of all objects belonging to that cluster. They typically start with an initial partition of D formed by specifying arbitrary k prototypes. Then, the following steps are performed iteratively until the clustering criterion cannot be improved:

- Assign each object to the closest prototype.
- For each cluster: recalculate the mean (prototype in the next iteration).

In practice k -means type algorithms converge fast. However, they are designed for numerical valued d -dimensional feature vectors only, whereas data mining applications may also consist of categorical valued vectors or objects for which only a dissimilarity measure is given.

The k -modes (see [Hua 97]) algorithm extends the k -means paradigm to categorical domains. To measure the dissimilarity between two categorical vectors X and Y , the simple matching dissimilarity [KR 90], i.e. the total number of mismatches of the values in the corresponding attributes of X and Y , is used. Then it is possible to find a so-called *mode* for a set S of categorical vectors, i.e. a vector Q that minimizes the sum of distances between Q and the elements of S . A mode for categorical values corresponds to the mean for numerical values. Hence, the same algorithmic schema as for k -means can be used to cluster categorical vectors.

In applications where only a dissimilarity measure for objects is defined, the calculation of a mean or a mode of a cluster is not possible. However, there is another type of optimizing clustering algorithm for this kind of data sets:

For k -medoid algorithms (see e.g. [KR 90]) each cluster is represented by a prototype which is identical to one of the objects of the cluster. This object, called the

medoid of a cluster is generally located *near* the “center” of the cluster. Like *k*-means algorithms also *k*-modes algorithms typically start by selecting arbitrary *k* prototypes to form an initial partition. Then, the following steps are performed iteratively until the clustering criterion cannot be improved:

- Assign each object to the closest prototype/medoid.
- Select one of the medoids and try to exchange the medoid with a non-medoid such that the clustering criterion is improved.

Basically, *k*-medoid algorithms differ only in the search strategy for exchanging medoids with non-medoids. Obviously, there is a trade-off between the resulting clustering quality and the run-time of *k*-medoid algorithms. The more exhaustive the search, the better the clustering criterion can be improved.

The first clustering algorithm used for mining in large spatial databases has been a *k*-medoid type algorithm (see [NH 94]). This algorithm called *CLARANS* (*Clustering Large Applications based on RANdomized Search*) is an improved *k*-medoid algorithm with a randomized and bounded search strategy for exchanging prototypes. The experiments in [NH 94] show that the algorithm *CLARANS* is significantly more efficient than the well-known *k*-medoid algorithms *PAM* (Partitioning Around Medoids) and *CLARA* (Clustering LARge Applications) presented in [KR 90] while producing a result of nearly the same clustering quality.

Optimization based clustering algorithms are effective in determining the “correct” clustering if the clusters are of “convex” shape, similar size and if their number *k* can be reasonably estimated. However, they may sometimes suffer from the problem of local minima due to their limited search strategy. Figure 7 depicts an example of a *k*-means and a *k*-medoid clustering for the same $2d$ data set.¹

1. Note that for *d*-dimensional numerical vectors assigning objects to the closest prototype yields a partition that is equivalent to a Voronoi diagram of the prototypes.

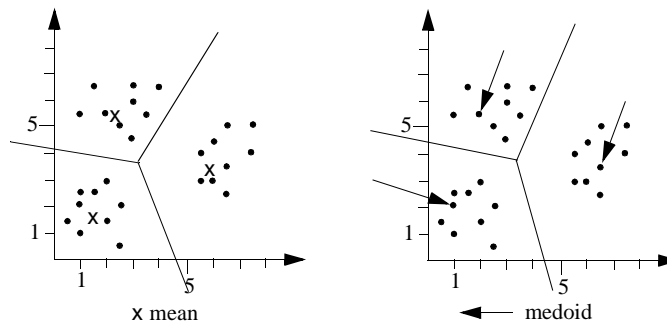


Figure 7: k-means and k-medoid ($k = 3$) clustering for a sample data set

If k is not known in advance, various parameter values can be tried and for each of the discovered clusterings a measure, e.g. the *silhouette coefficient* [KR 90], indicating the suitability of the number of clusters k can be calculated. Another possible strategy is to adjust the number of clusters after the main clustering procedure by merging and splitting existing clusters or by removing small clusters or outliers. For instance, in the clustering algorithms called ISODATA [BH 64], conditions for splitting and merging clusters can be specified by the user. There exists also an adaptive k -means type algorithm that does not need k as an input parameter [Sch 91]. This algorithm starts with an initial clustering consisting of only one prototype - the mean of the whole data set. Then, the multidimensional Voronoi diagram for all prototypes of the current clustering is computed and the Voronoi cell where the largest error occurs is split, i.e. two new clusters/prototypes are inserted in this region. This procedure is repeated until a user-specified termination condition is met.

Clustering Based on Graph Theory

Several kinds of graphs have been used for analyzing multidimensional objects. These graphs consist of nodes which represent the objects to be clustered and edges which represent relations between the nodes. In the simplest case, every node is connected to all the remaining nodes, resulting in the complete graph for a data set.

The edge weights are the distances between pairs of objects. For the purpose of clustering, typically a subset of the edges in the complete graph is selected to reflect the inherent separation among clusters. One of the best known clustering methods based on graph theory is Zahn's method which uses the minimum spanning tree (MST)¹ of a data set [Zah 71].

Zahn demonstrates how the MST can be used with different notions of an inconsistent edge to identify clusters. The basic idea of Zahn's clustering algorithm consists of the following three steps:

- Construct the MST for the set of points.
- Identify inconsistent edges in the MST.
- Remove the inconsistent edges.

The resulting connected components are the clusters. The algorithm can be applied iteratively to these components to identify subclusters. Zahn considers several criteria for the inconsistency of edges. For example, an edge is inconsistent if its interpoint distance is significantly larger than the average of interpoint distances of nearby edges. This notion of inconsistent edges works well in simple situations where the clusters are well separated and the density within a cluster only varies smoothly. Special heuristics are needed for more complex situations and a priori knowledge of the shapes of the clusters is then needed to select the proper heuristics to identify inconsistent edges. Also, the computational costs for constructing the MST and finding inconsistent edges are very high for large data sets.

Figure 8 depicts the MST of a sample data set of two-dimensional points. The points are grouped in three clusters and the inconsistent edges are marked.

1. A *spanning tree* for a set of objects D is a connected graph with no cycles that contains a node for each object of D . The *weight of a tree* is the sum of the edge weights in the tree. A *minimum spanning tree* of D is a spanning tree which has the minimal weight among all other spanning trees of D .

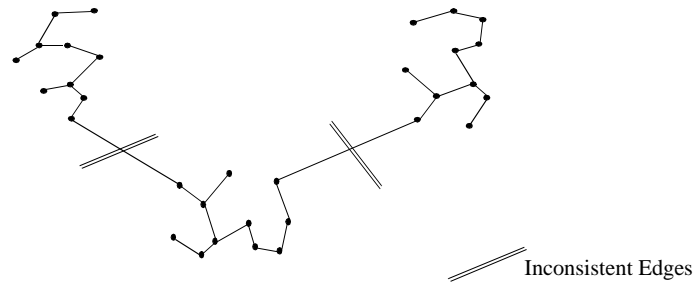


Figure 8: MST of a sample data set

Distribution-Based Approaches

A popular statistical approach to clustering is based on the notion of a “mixture density”. Each object is assumed to be drawn from one of k underlying populations or clusters. In this approach which is known as “mixture decomposition” or “model based clustering”, the form and the number of underlying cluster densities are assumed to be known. Although, in principle arbitrary density functions or distributions are possible, the common practice is to assume a mixture of Gaussian distributions (see e.g. [JD 88], [Sym 81]). The density of a d -dimensional vector x from the i -th cluster is assumed to be $f_i(x; \theta)$ for some unknown vector of parameters θ . In the so-called classification maximum likelihood procedure, θ and the identifying labels c_i for the n objects x_1, \dots, x_n are chosen so as to maximize the likelihood

$$L(\theta, (c_1, \dots, c_n)) = \prod_{i=1}^n f_{c_i}(x_i; \theta)$$

In [BR 93] a solution for a more general model is worked out, additionally allowing the incorporation of noise in the form of a Poisson process. However, even this model is not applicable if we do not know the number of clusters or if the processes generating the data are not Gaussian.

Another approach to clustering which also assumes a certain distribution of the data but which does not require the number of clusters to be known in advance is presented in [XEKS 98]. The clustering algorithm *DBCLASD* (*Distribution Based Clustering of Large Spatial Databases*) assumes that the points inside of a cluster are randomly distributed which is quite reasonable for many applications (see e.g. [AS 91], [BR 96] and [MS 92]). This implies a characteristic probability distribution of the distance to the nearest neighbors for the points of a cluster. The algorithm *DBCLASD* presupposes this distribution and incrementally augments an initial cluster by its neighboring points as long as the nearest neighbor distance set of the resulting cluster still fits the expected distribution. Thus, if the points inside of the clusters are almost randomly distributed, *DBCLASD* is able to detect clusters of arbitrary shape without requiring any further input parameters such as the number of clusters. However, incrementally checking an expected distribution implies an inherent dependency of the discovered clustering from the order in which candidates from the database are generated and tested. Therefore, two heuristics to reduce the effects of this dependency are incorporated in the algorithm: unsuccessful candidates for the current cluster are not discarded but tried again later; points already assigned to some cluster may switch to another cluster later.

Density-Based Approaches

Clusters can be viewed as regions in the data space in which the objects are dense, separated by regions of low object density. The general idea of density-based clustering approaches is to search for regions of high density in the data space. These regions may have an arbitrary shape and the points inside a region may be arbitrarily distributed.

A common way to find regions of high-density in the dataspace is based on grid cell densities (see e.g. [JD 88] for an overview). A histogram is constructed by partitioning the data space into a number of non-overlapping regions or cells. Cells containing a relatively large number of objects are potential cluster centers and the boundaries between clusters fall in the “valleys” of the histogram. In general there

are two possibilities to construct clusters, starting from the potential centers. In the first method, the potential centers are taken as actual cluster centers and each object is then assigned to the cluster with the closest center. In the second method, neighboring cells are merged with the potential centers as long as the density in the neighboring cells is similar enough to the density of the centers. That means that in the second approach, clusters are constructed as connected regions of grid cells.

Obviously, the success of this method depends on the size of the cells which must be specified by the user. Cells of small volume will give a very “noisy” estimate of the density, whereas large cells tend to overly smooth the density estimate. Furthermore, the memory and run-time requirements of storing and searching multidimensional histograms may become very large because the number of cells in a d -dimensional grid grows exponentially with increasing dimension d .

Recently, the density-based clustering technique CLIQUE (CLustering In QUEst) [AGG+ 98] has been proposed for mining in high-dimensional data spaces. This method also relies on a partition of the space into a regular grid. A cluster is defined as a region, i.e. a set of connected grid cells, that has a higher density of points than its surrounding region. More important, the method automatically detects *subspaces* of the highest dimensionality such that high-density clusters exist in those subspaces. A subspace is a projection of the input data into a subset of the attributes. The identification of clusters works in two steps:

- 1.) Determination of dense units, i.e. cells, in all subspaces of interest.
- 2.) Determination of connected dense units in all subspaces of interest.

To check each cell in a high-dimensional grid is computationally unfeasible. Therefore, to determine the dense units in all subspaces a bottom-up procedure is used, based on the monotonicity property for high-density clusters: if S is a cluster in a d -dimensional space, then S is also part of a cluster in $(d-1)$ -dimensional projections of this space. The algorithm starts by determining one-dimensional dense units. Then, having determined $(k-1)$ -dimensional dense units, the candidate k -di-

mensional dense units are determined and a pass over the data is made to determine those candidates which are actually dense. The candidate generation procedure produces a superset of all k -dimensional dense units by self-joining the set of $(k-1)$ -dimensional dense units where the join condition requires that the units share the first $k-2$ dimensions.¹

In the second step, clusters are constructed in each subspace where dense units have been found. A labyrinth-search schema is used to determine regions of connected units, i.e. units that have a common face.

In [EKSX 96] a density-based clustering method is presented which is not grid-based. The basic idea for the algorithm *DBSCAN* (*Density Based Spatial Clustering of Applications with Noise*) is that for each point of a cluster the neighborhood of a given radius (ϵ) has to contain at least a minimum number of points (*MinPts*), i.e. the density in the neighborhood of points in a cluster has to exceed some threshold. A simple heuristic which is effective in many cases to determine the two parameters (ϵ , *MinPts*) can be used to support the user in determining these parameters. The algorithm *DBSCAN* checks the ϵ -neighborhood of each point in the database. If the ϵ -neighborhood $N_\epsilon(p)$ of a point p has more than *MinPts* points the region $N_\epsilon(p)$ is expanded to a cluster by checking the ϵ -neighborhood of all points in $N_\epsilon(p)$. For all points q where $N_\epsilon(q)$ contains more than *MinPts* points, also the neighbors of q are added to the cluster, and their ϵ -neighborhood is checked in the next step. This procedure is repeated until no new point can be added to the current cluster.

This algorithm is based on the formal notion of a cluster as a maximal set of density-connected points. A point p is density-connected to a point q if there exists a point o such that both p and q are density-reachable from o (directly or transitively). A point p is said to be directly density-reachable from o if p lies in the neigh-

1. This bottom-up construction uses the same algorithmic trick as the Apriori algorithm for finding Association Rules presented in [AS 94].

neighborhood of o and the neighborhood of o contains at least $MinPts$ points. We will use this density-based clustering approach as a starting point for our generalization in the following chapters.

2.2.3 Region Growing

Image Segmentation is a task in the field of Computer Vision which deals with the analysis of the spatial content of an image. In particular, it is used to separate regions from the rest of the image in order to recognize them as objects.

Region Growing is an approach to image segmentation in which neighboring pixels are examined and added to a region class if no edges are detected. This process is iterated for each boundary pixel in the region. Several image properties, such as a low gradient or a gray-level intensity value can be used in combination to define the membership of pixels to a region (see e.g. [Nie 90]). In general, all pixels with grey level (or color) 0 are assumed to be the background, while pixels with color > 0 are assumed to belong to foreground objects. A connected component in the image is a maximal collection of uniformly colored pixels such that a path exists between any pair of pixels in the component. Two pixels are adjacent if one pixel lies in one of the eight positions surrounding the other pixel. Each pixel in the image will receive a label; pixels will have the same label if and only if they belong to the same connected component. All background pixels will receive a label of 0.

The definition of a connection between two neighboring pixels depends on the application. In the most simple form, two pixels are adjacent if and only if their grey level values are identical. Another possibility is that two adjacent pixels having gray-level values x and y are defined to be connected if the absolute difference $|x - y|$ is not greater than a threshold (setting the threshold to 0 reduces this case to the simple one mentioned above). Other approaches may take into account additional information about the image or may consider aggregate values such as the average intensity value in the neighborhood of a pixel.

2.3 Exploiting the Clustering Properties of Index Structures

In this section, we show how spatial indexes and similar data structures can be used to support the clustering of very large databases. These structures organize the data objects or the data space in a way that objects which are close to each other are grouped together on a disk page (see section 2.1). Thus, index structures contain useful information about the distribution of the objects and their clustering structure. Therefore, index structures can be used to support and speed-up basic operations in some of the known clustering algorithms. They can be used as a kind of preprocessing for clustering algorithms or even to build special clustering algorithms which take advantage of the information stored in the directory of an index.

2.3.1 Query Support for Clustering Algorithms

Different methods to support the performance of clustering techniques have been proposed in the literature. The techniques discussed in this section rely on the efficient processing of similarity queries (*kNN*-queries and region queries) when using spatial index structures.

The time complexity of hierarchical clustering algorithms is at least $O(n^2)$ if all inter-object distances for an object have to be checked to find its *NN*. Already Murtagh [Mur 83] points out that spatial index structures make use of the fact that finding of *NNs* is a “local” operation because the *NN* of an object can only lie in a restricted region of the data space. Thus, using n-dimensional hash- or tree-based index structures for efficient processing of *NN* queries can improve the overall runtime complexity of agglomerative hierarchical clustering algorithms. If a disk-based index structure, e.g. a grid file or R-Tree, is used instead of a main-memory-based index structure, these clustering algorithms can also be used for larger data sets.

The k -medoid algorithm CLARANS is still too inefficient to be applied to very large databases because its measured run-time complexity seems to be of the order n^2 . In [EKX 95a] two techniques to integrate CLARANS with an SDBS using a spatial index structure are proposed. The first is R^* -tree based sampling (see section 2.3.2), the second is called *focusing on relevant clusters* which uses a spatial index structure to reduce the computational costs for comparing the quality of two clusterings - an operation which is performed in each step of the algorithm. This technique is described in the following:

Typically, k -medoid algorithms try to improve a current clustering by exchanging one of the medoids of the partition with one non-medoid and then compare the quality of this “new” clustering with the quality of the “old” one. In CLARANS, computing the quality of a clustering is the most time consuming step because a scan through the whole database is performed. However, only objects which belonged to the cluster of the exchanged medoid or which will belong to the cluster of the new medoid contribute to the *change* of quality. Thus, only the objects of two (out of k) clusters have to be read from disk. To retrieve exactly the objects of a given cluster, a region query can be used. This region, a Voronoi cell whose center is the medoid of the cluster, can be efficiently constructed by using only the information about the medoids and the minimum bounding box of all objects in the database. Assuming the same average size for all clusters, a performance gain of $k/2$ (measured by the number of page accesses) compared to [NH 94] is expected.

Clustering algorithms which group *neighboring* objects of the database into clusters based on a local cluster condition can be formulated so that only a “single scan” over the database is performed. Each object has to be examined once and its neighborhood has to be retrieved. If the retrieval of the neighborhood of an object can be efficiently supported - for instance, if the neighborhood can be expressed by a region query for which a supporting spatial index structure exists - this algorithmic schema yields efficient clustering algorithms integrated with SDBMS.

The algorithmic schema of a single scan clustering algorithm is as follows:

```
SingleScanClustering(Database DB)
FOR each object o in DB DO
  IF o is not yet member of some cluster THEN
    create a new cluster C;
    WHILE neighboring objects satisfy the cluster condition DO
      add them to C
    ENDWHILE
  ENDIF
ENDFOR
```

Different cluster conditions yield different cluster definitions and algorithms. For example, the clustering algorithms DBCLASD and DBSCAN are instances of this type of algorithm.

2.3.2 Index-Based Sampling

To cluster large databases in a limited main memory, one can select a relatively small number of representatives from the database and apply the clustering algorithm only to these representatives. This is a kind of data sampling, a technique common in cluster analysis [KR 90]. The drawback is that the quality of the clustering will be decreased by considering only a subset of the database and that it depends heavily on the quality of the sample.

Traditional data sampling works only in main memory. In [EKX 95b] a method of selecting representatives from a spatial database system is proposed. From each data page of an R^* -tree, one or several representatives are selected. Since the clustering strategy of the R^* -tree, which minimizes the overlap between directory rectangles, yields a well-distributed set of representatives (see figure 9), the quality of the clustering will increase only slightly. This is confirmed by experimental results [EKX 95b] which show that the efficiency is improved by a factor of 48 to 158 whereas the clustering quality decreases only by 1.5% to 3.2% when comparing the

clustering algorithm *CLARANS* [NH 94] with and without index-based sampling. In principle, other page based spatial index structures could also be used for this kind of sampling technique because their page structure usually adapts to the distribution of the data as well.

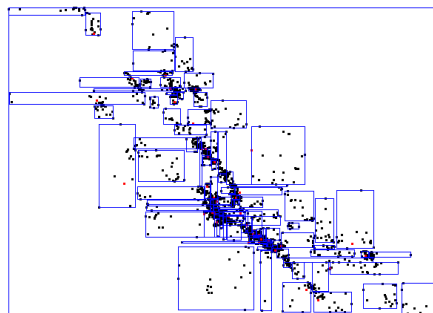


Figure 9: Data page structure of an R^* -tree for a $2d$ -point database

2.3.3 Grid clustering

Schikuta [Sch 96] proposes a hierarchical clustering algorithm based on the grid file (see section 2.1). Points are clustered according to their grid cells in the grid structure. The algorithm consists of 4 main steps:

- Creation of the grid structure
- Sorting of the grid cells according to cell densities
- Identifying cluster centers
- Recursive traversal and merging of neighboring cells

In the first part, a grid structure is created from all points which completely partitions the data space into a set of non-empty disjoint rectangular shaped cells containing the points. Because the grid structure adapts to the distribution of the points

in the data space, the creation of the grid structure can be seen as a pre-clustering phase (see figure 10).

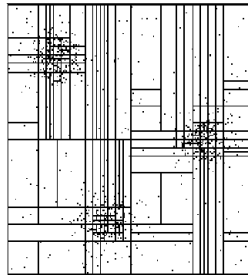


Figure 10: Data page structure of a Grid File for a 2d-point database [Sch 96]

In the second part, the grid data pages (containing the points from one or more grid cells, *c.f.* figure 1) are sorted according to their density, i.e. the ratio of the actual number of points contained in the data page and the spatial volume of the data page. This sorting is needed for the identification of cluster centers in the third part.

Part 3 selects the pages with the highest density as cluster centers (obviously a number of pages may have the same cell density). Step 4 is performed repeatedly until all cells have been clustered. Starting with cluster centers, neighboring pages are visited and merged with the current cluster if they have a lower or equal density than the actual page. Then the neighboring pages of the merged neighbors are visited recursively until no more merging can be done for the current cluster. Then the next unclustered page with the highest density is selected. Experiments [Sch 96] show that this clustering algorithm clearly outperforms hierarchical and partitioning methods of the commercial statistical package SPSS.

The Grid clustering approach is not very specific to the grid file. In fact, it would be possible to apply a similar procedure to the data pages of other spatial index structures as well and thus obtain very similar results.

2.3.4 CF-Tree

[ZRL 96] presents the clustering method *BIRCH* (*Balanced Iterative Reducing and Clustering using Hierarchies*) which uses a highly specialized tree-structure for the purpose of clustering very large sets of d -dimensional vectors. The advantage of this structure is that its memory requirements can be adjusted to the main memory that is available.

BIRCH incrementally computes compact descriptions of subclusters, called Clustering Features CF that contain the number of points, the linear sum and the square sum of all points in the cluster:

$$CF = \left(n, \sum_{i=1}^n \vec{x}_i, \sum_{i=1}^n \vec{x}_i^2 \right)$$

The CF -values are sufficient for computing information about subclusters like centroid, radius and diameter and constitute an efficient storage method since they summarize information about subclusters instead of storing all points.

The Clustering Features are organized in a balanced tree with branching factor B and a threshold T (see figure 11). A non-leaf node represents a cluster consisting of all the subclusters represented by its entries. A leaf node has to contain at most L entries and the diameter of each entry in a leaf node has to be less than T . Thus, the parameter T has the most significant influence on the size of the tree.

In the first phase, BIRCH performs a linear scan of all data points and builds a CF -tree. A point is inserted by inserting the corresponding CF -value into the closest leaf of the tree. If an entry in the leaf can absorb the new point without violating the threshold condition, then the CF -values for this entry are updated, otherwise a new entry in the leaf node is created. In this case, if the leaf node contains more than L entries after insertion, the leaf node and possibly its ancestor nodes are split. In an optional phase 2 the CF -tree can be further reduced until a desired number of

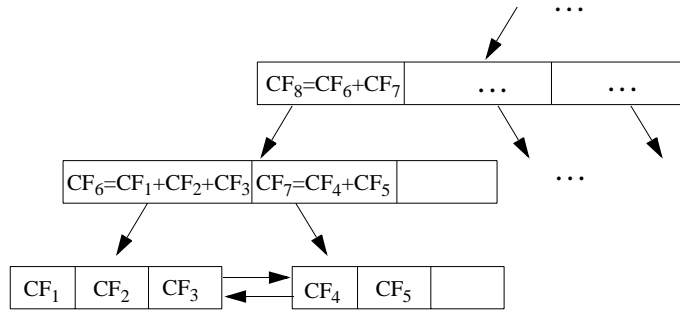


Figure 11: CF-tree structure

leaf nodes is reached. In phase 3 an arbitrary clustering algorithm such as CLARANS is used to cluster the leaf nodes of the CF -tree.

The efficiency of BIRCH is similar to the index based sampling (see section 2.3.2) and experiments with synthetic data sets [ZRL 96] indicate that the quality of the clustering using BIRCH in combination with CLARANS is even higher than the quality obtained by using CLARANS alone.

2.3.5 STING

Wang et al. [WYM 97] propose the *STING* (*Statistical Information Grid based*) method which relies on a hierarchical division of the data space into rectangular cells. Each cell at a higher level is partitioned into a fixed number c of cells at the next lower level. The skeleton of the STING structure is similar to a spatial index structure - in fact, their default value for c is 4, in which case we have an equivalence for two-dimensional data to the well-known *Quadtree* structure [Sam 90]. This tree structure is further enhanced with additional statistical information in each node/cell of the tree (see figure 12). For each cell the following values are calculated and stored:

- n - the number of objects (points) in the cell.

And for each numerical attribute:

- m - the mean of all values in the cell
- s - the standard deviation of all values in the cell
- min - the minimum value in the cell
- max - the maximum value in the cell
- $distr$ - the type of distribution that the attribute values in this cell follow (enumeration type)

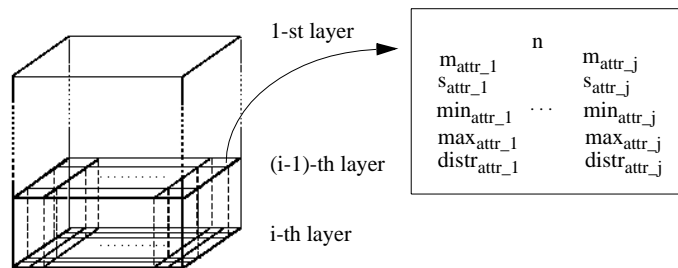


Figure 12: STING structure [WYM 97]

The STING structure can be used to answer efficiently different kinds of region-oriented queries, e.g., finding maximal connected regions which satisfy a density condition and possibly additional conditions on the non-spatial attributes of the points. The algorithm for answering such queries first determines all bottom level cells which are relevant to the query and then constructs the connected regions of those relevant cells.

The bottom level cells that are relevant to a query are determined in a top down manner, starting with an initial layer in the STING structure - typically the root of the tree. The relevant cells in a specific level are determined by using the statistical information. Then, the algorithm goes down the hierarchy by one level, considering only the children of relevant cells at the higher level. This procedure is iterated until the leaf cells are reached.

The regions of relevant leaf cells are then constructed by a breadth first search. For each *relevant* cell, cells within a certain distance are examined and merged with the current cell if the average density within the area is greater than a specified threshold. This is in principle the DBSCAN algorithm [EK SX 96] performed on cells instead of points. Wang et al. prove that the regions returned by STING are approximations of the clusters discovered by DBSCAN which become identical as the granularity of the grid approaches zero.

Wang et al. claim that the run-time complexity of STING is $O(C)$, where C is the number of bottom level cells. C is assumed to be much smaller than the number N of all objects which is reasonable for low dimensional data. However, to assure $C \ll N$ for high dimensions d , the space cannot be divided along all dimensions: even if the cells are divided only once in each dimension, then the second layer in the STING structure would contain already 2^d cells. But if the space is not divided often enough along all dimensions, both the quality of cell-approximations of clusters as well as the run-time for finding them will deteriorate.

2.4 Summary

In this chapter we first gave an overview of spatial indexing methods to support similarity queries which are important building blocks for many clustering and other spatial data mining algorithms. Second, different types of clustering algorithms were reviewed, and we showed how many of these algorithms - traditional as well as new ones - can be integrated into a spatial database management system using the query processing facilities of the SDBS. In the last section, we presented different clustering techniques which exploit the clustering properties of spatial index structures directly.

Chapter 3

Density-Based Decompositions

In this chapter the basic notions of our work are introduced. First, we give a short motivation for the generalization of a density-based clustering to a density-based decomposition (section 3.1.1). Then, we present a set-theoretic definition of density-connected sets which characterize generalized density-based clusters (section 3.1.2). Density-based decompositions are then simply given as classes of density-connected sets (section 3.2.1). Important specializations of these notions include some familiar structures from clustering and pattern recognition as well as new applications which may be appropriate for grouping extended objects, e.g. polygons in geographic information systems (section 3.2.2). We also discuss the determination of parameters which may be required by some specializations and present a simple but in most cases effective heuristic to determine parameters for an important specialization of a density-based decomposition (section 3.3).

3.1 Density-Connected Sets

3.1.1 Motivation

Clustering can be regarded as a basic operation in a two step procedure for spatial data mining (*cf.* figure 13). A first step, implemented by a clustering algorithm, where we look for implicit spatial structures or clusters or groups of “similar” objects, for instance, dense groups of expensive houses in a geographic database. Then, in the second step, the detected structures are further analyzed. We may, for example, explain the groups of expensive houses detected in the first step by other features located in their neighborhood, for instance, rivers or lakes.

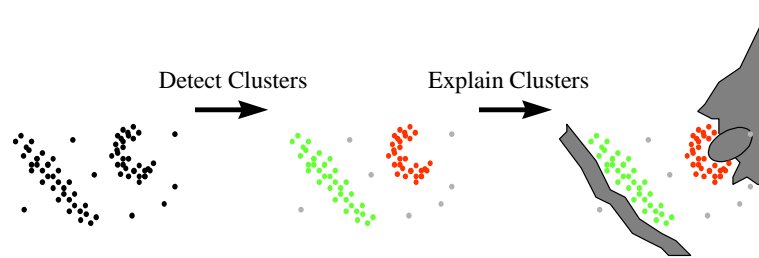


Figure 13: Clustering and spatial data mining

In principle, any clustering algorithm could be used for the first step of this task if it is possible to apply it to the data set under consideration (recall the limitations of traditional clustering algorithms discussed in the previous chapter). However, the well-known clustering algorithms are designed for point-like objects, i.e. objects having no other characteristics - from the viewpoint of the algorithm - than their position in some d -dimensional vector space, or alternatively, their distances to other objects in a metric space.

It is not obvious how to apply these methods to other kinds of objects, for example to spatially extended objects like polygons in a geographic database. Using a traditional clustering algorithm, there are in principle two possibilities: First, the representation of the polygons by distinguished points, for example the centers of gravity, or second, the definition of a distance measure. For instance, define the distance between two polygons by the minimum distance between their edges. However, these methods will often result in a very poor representation of the original distribution of the objects in the database, which means that important information is lost.

In figure 14, an example of the transformation to gravity centers for a set of polygons is depicted. In the original space we can see three clusters or groups and some smaller polygons which may be considered as noise. This structure is obviously not preserved if the polygons are transformed to points.

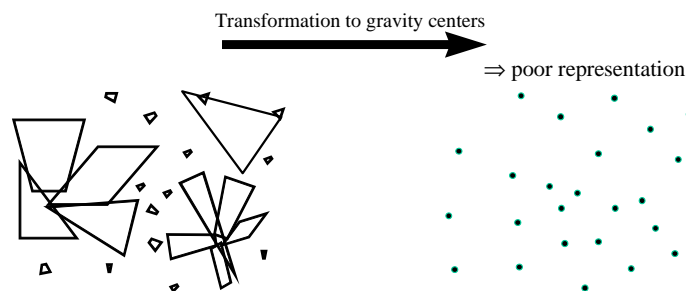


Figure 14: Example for a generalized clustering problem

In the ideal case, to find cluster-like groups of objects in a geographic information system, we want to take into account perhaps the area of the polygons - maybe other non-spatial attributes, like average income for polygons representing communities, would be also useful. And, what the example in figure 14 also suggests

is that using natural notions of connectedness for polygons like intersects or meets instead of distance based measures may be more appropriate.

To conclude, our goal is to extend the notion of a clustering to arbitrary (spatial) objects so that the additional information about the objects given by their (spatial and non-spatial) attributes can be used directly to reveal and analyze hidden structures in a spatial database.

3.1.2 Definitions and Properties

We will use the density-based clustering approach as a starting point for our generalization:

The key idea of a density-based cluster as presented in [EK SX 96] is that for most points of a cluster the ϵ -neighborhood for some given $\epsilon > 0$ has to contain at least a minimum number of points, i.e. the “density” in the ϵ -neighborhood of the points has to exceed some threshold. This idea is illustrated by the sample sets of 2-dimensional points depicted in figure 15. In these very simple examples we see that clusters have a typical density of points inside which is considerably higher than outside of the clusters. Furthermore, the density within areas of noise is lower than the density of the clusters, and clusters may have arbitrary shape, size and location.

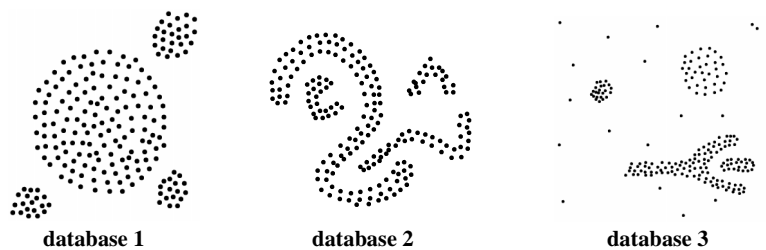


Figure 15: Sample databases of 2d points

The idea of density-based clusters in [EK SX 96] can be generalized in two important ways. First, we can use any notion of a neighborhood instead of an ϵ -neighborhood if the definition of the neighborhood is based on a binary predicate which is symmetric and reflexive. Second, instead of simply counting the objects in the neighborhood of an object we can use other measures to define the “cardinality” of that neighborhood as well. Figure 16 illustrates the intuition and the goal of the following definitions.

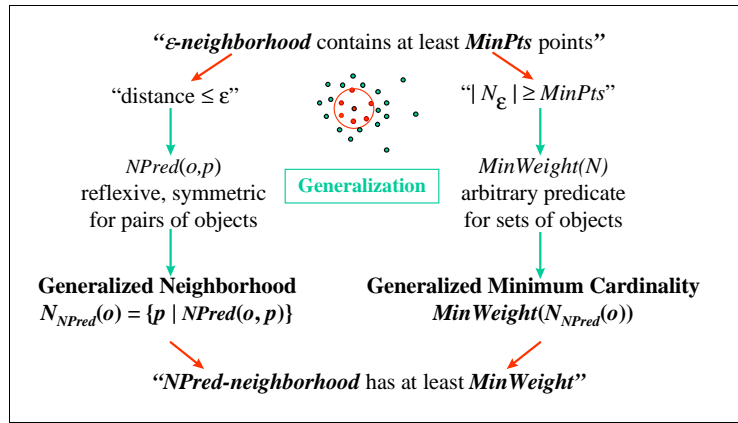


Figure 16: Generalization of density-based clusters

Definition 1: (Notation: *neighborhood of an object*)

Let $NPred$ be a binary predicate on D which is reflexive and symmetric, i.e., for all $p, q \in D$: $NPred(p, p)$, and if $NPred(p, q)$ then $NPred(q, p)$.

Then the $NPred$ -neighborhood of an object $o \in D$ is given as

$$N_{NPred}(o) = \{o' \in D \mid NPred(o, o')\}.$$

The definition of a cluster in [EKSX 96] is restricted to the special case of a distance based neighborhood, i.e., $N_\epsilon(o) = \{o' \in D \mid |o - o'| \leq \epsilon\}$. A distance based neighborhood is a natural notion of a neighborhood for point objects, but if clustering spatially extended objects such as a set of polygons of largely differing sizes it may be more appropriate to use neighborhood predicates like *intersects* or *meets* to detect clusters of polygons (cf. figure 14).

Although in many applications the neighborhood predicate will be defined by using only spatial properties of the objects, the formalism is in no way restricted to purely spatial neighborhoods. As well, we can use non-spatial attributes and combine them with spatial properties of objects to derive a neighborhood predicate (see for instance the application to a geographic database in section 5.4).

Another way to take into account the non-spatial attributes of objects is as a kind of “weight” when calculating the “cardinality” of the neighborhood of an object. For this purpose, we can define an arbitrary predicate expressing the “minimum weight” for sets of objects.

Definition 2: (Notation: *minimum weight of a set of objects*)

Given an arbitrary unary predicate *MinWeight* for sets of objects from a database *D*. We say that a set of objects $N \subseteq D$ has minimum weight (with respect to the predicate *MinWeight*) if $\text{MinWeight}(N)$.

The density threshold condition $|N_\epsilon(o)| \geq \text{MinPts}$ in the definition of density-based clusters is just a special case for the definition of a *MinWeight* predicate. There are numerous other possibilities to define a *MinWeight* predicate for subsets *S* of a database *D*. Simply summing up values of some non-spatial attribute for the objects in *S* is another example. If we want to cluster objects represented by polygons and if the size of the objects should be considered to influence the “density” in the data space, then the area of the polygons could be used as a weight for these objects. A further possibility is to specify ranges for some non-spatial attribute val-

ues of the objects, i.e. specifying a selection condition and counting only those objects which satisfy this selection condition (see e.g. the biology application in section 5.2). Thus, we can realize the clustering of only a subset of the database D by attaching a weight of 1 to those objects that satisfy the selection condition and a weight of 0 to all other objects. Note that using non-spatial attributes as a weight for objects, one can “induce” different densities, even if the objects are equally distributed in the space of the spatial attributes. Note also, that by means of the *MinWeight* predicate the combination of a clustering with a selection on the database can be performed “on the fly” while clustering the database. Under certain circumstances, this may be more efficient than performing the selection first, because the algorithm GDBSCAN to compute generalized clusters can use existing spatial index structures to speed-up the clustering procedure.

We will now define two special properties for *MinWeight* predicates (for later use see chapter 6 and chapter 7):

Definition 3: (incrementally evaluable *MinWeight*, monotonous *MinWeight*)

A *MinWeight* predicate for sets of objects is called *incrementally evaluable* if there is a function $weight: Po(Objects) \rightarrow \mathbf{R}$ and a threshold $T \in \mathbf{R}$ such that $weight(N) = \sum_{o \in N} weight(\{o\})$ and $MinWeight(N)$ iff $weight(N) \geq T$.

A *MinWeight* predicate for sets of objects is called *monotonous* if the following condition holds: if $N_1 \subseteq N_2$ and $MinWeight(N_1)$ then also $MinWeight(N_2)$.

Incrementally evaluable *MinWeight* predicates compare the *weight* of a set N of objects to a threshold and the weight of the set N can be evaluated incrementally in the sense that it is the sum of the weights of the single objects contained in N . *MinWeight* predicates having this property are the most important *MinWeight* predicates for practical use. For a monotonous *MinWeight* predicate it holds that if a set N has minimum weight, then every super-set of N also has minimum weight. Note that if the function *weight* in the definition of an incrementally evaluable *Min-*

Weight predicate is positive, i.e. $weight(N) \geq 0$ for all subsets N of D , the corresponding *MinWeight* predicate is obviously monotonous. The density threshold condition $|N_g(o)| \geq MinPts$ in the definition of density-based clusters is an example for a definition of a *MinWeight* predicate which is both, incrementally evaluable and monotonous.

We can now define density-connected sets, analogously to the definition of density-based clusters, in a straightforward way (see also [SEKX 98]). First, we observe that there are two kinds of objects in the “area” of a density-connected set, objects “inside” (*core objects*) and objects “on the border” (*border objects*) of a density-connected set. In general, the *NPred*-neighborhood of a border object may have a significantly lower weight than the *NPred*-neighborhood of a core object. Therefore, to include all objects belonging to the “area” of the same density-connected set, we would have to define the predicate *MinWeight* in a way, which may not be characteristic for the respective density-connected set. For instance, if we use an incrementally evaluable *MinWeight* predicate, we would have to set the threshold value T to a relatively low value. This value, however, may then also be characteristic for objects which do not belong to any cluster - particularly in the presence of a large amount of noise objects. Core objects and border objects are illustrated in figure 17 for the 2-dimensional case using a distance based neighborhood and cardinality in the definition of the *MinWeight* predicate.

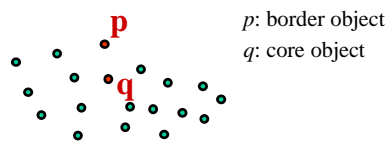


Figure 17: Core objects and border objects

Therefore, we require that for every object p in a density-connected set C there must be an object q in C so that p is inside of the *NPred*-neighborhood of q and

$NPred(q)$ has at least minimum weight. We also require that the objects of the set C have to be somehow “connected” to each other. This idea is elaborated in the following definitions and illustrated by 2-dimensional point objects using a distance based neighborhood for the points and “cardinality of the ϵ -neighborhood $\leq MinPts$ ” for the *MinWeight* predicate.

Definition 4: (*directly density-reachable*)

An object p is *directly density-reachable* from an object q with respect to $NPred$ and *MinWeight* in a database D if

- 1) $p \in N_{NPred}(q)$ and
- 2) $MinWeight(N_{NPred}(q))$ holds (core object condition).

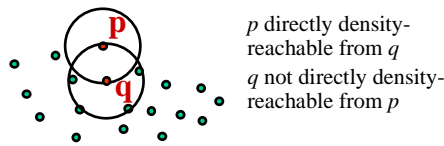


Figure 18: Direct density-reachability

The predicate *directly density-reachable* is symmetric for pairs of *core* objects. In general, however, it is not symmetric if one core object and one border object are involved. Figure 18 shows the asymmetric case.

Lemma 1:

If object p is directly density-reachable from object q with respect to $NPred$ and *MinWeight*, and also p is a core object, i.e. $MinWeight(N_{NPred}(p))$ holds, then also q is directly density-reachable from p .

Proof: Obvious. \square

Definition 5: (*density-reachable*, $>_D$)

An object p is *density-reachable* from an object q with respect to $NPred$ and $MinWeight$ in a database D if there is a chain of objects p_1, \dots, p_n , $p_1 = q$, $p_n = p$ such that for all $i=1, \dots, n-1$: p_{i+1} is directly density-reachable from p_i with respect to $NPred$ and $MinWeight$ in D .

If $NPred$ and $MinWeight$ are clear from the context, we will sometimes denote the fact that p is density-reachable from q in the database D as “ $p >_D q$ ”.¹

Density-reachability is a canonical extension of direct density-reachability. This relation is transitive but it is not symmetric. Figure 20 depicts the relations of some sample objects and, in particular, the asymmetric case.

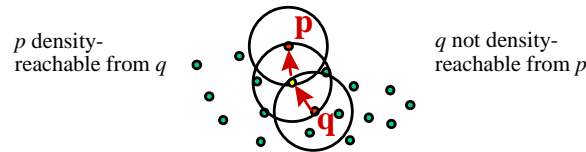


Figure 19: Density-reachability

Although not symmetric in general like *direct* density-reachability, density-reachability is symmetric for core objects because a chain from q to p can be reversed if also p is a core object. For core objects the density-reachability is also reflexive, i.e. a core object is density-reachable from itself.

Lemma 2:

- (a) density-reachability is transitive
- (b) density-reachability is symmetric for core objects
- (c) density-reachability is reflexive for core objects

1. The notation $p >_D q$ will be used extensively in chapter 6 where we must distinguish a database before and after an update.

Proof:

(a) density-reachability is the *transitive hull* of direct density-reachability.

(b) Let p be density-reachable from q with respect to $NPred$ and $MinWeight$. Then, there is a chain of objects p_1, \dots, p_n $p_1 = q, p_n = p$ such that for all $i=1, \dots, n-1$: p_{i+1} is directly density-reachable from p_i with respect to $NPred$ and $MinWeight$. Since $NPred$ is symmetric and each object p_j for $j=1, \dots, n-1$ satisfies the core object condition, q is density-reachable from p_{n-1} . By assumption, p is a core object. But then p_{n-1} is density-reachable from p (lemma 1). Hence, by transitivity (a), q is density-reachable from p .

(c) $NPred$ is a reflexive predicate, thus, $p \in N_{NPred}(p)$. Then, if p also satisfies the core object condition, i.e. $MinWeight(N_{NPred}(p))$ holds, p is density-reachable from p by definition. \square

Two border objects of the same density-connected set C are possibly not density-reachable from each other because the core object condition might not hold for both of them. However, for a density-connected set C we require that there must be a core object in C from which both border objects are density-reachable. Therefore, we introduce the notion of density-connectivity which covers this relation of border objects.

Definition 6: (*density-connected*)

An object p is *density-connected* to an object q with respect to $NPred$ and $MinWeight$ in a database D if there exists an object o such that both, p and q are density-reachable from o with respect to $NPred$ and $MinWeight$ in D .

Density-connectivity (*cf.* figure 20) is a symmetric relation. For density-reachable objects, the relation of density-connectivity is also reflexive. The relation is not transitive. But if p is density-connected to q via o_1 and q is density-connected

to r via o_2 then p is density-connected to r iff either o_1 is density-reachable from o_2 or o_2 is density-reachable from o_1 .

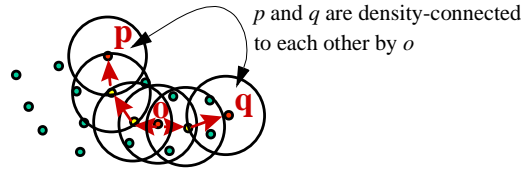


Figure 20: Density-connectivity

Lemma 3:

- (a) density-connectivity is symmetric
- (b) density-connectivity is reflexive for core objects

Proof:

- (a) By definition.
- (b) Because a core object o is density-reachable from itself (lemma 2 (c)). \square

Now, a density-connected set is defined to be a set of density-connected objects which is maximal with respect to density-reachability.

Definition 7: (*density-connected set*)

A *density-connected set* C with respect to $NPred$ and $MinWeight$ in D is a non-empty subset of D satisfying the following conditions:

- 1) Maximalty: For all $p, q \in D$: if $p \in C$ and q is density-reachable from p with respect to $NPred$ and $MinWeight$ in D , then also $q \in C$.
- 2) Connectivity: For all $p, q \in C$: p is density-connected to q with respect to $NPred$ and $MinWeight$ in D .

Note that a density-connected set C with respect to $NPred$ and $MinWeight$ contains at least one core object and has at least minimum weight: since C contains at least one object p , p must be density-connected to itself via some object o (which may be equal to p). Thus, at least o has to satisfy the core object condition. Consequently, the $NPred$ -Neighborhood of o has to satisfy $MinWeight$.

The following lemmata are important for validating the correctness of our clustering algorithm. Intuitively, they state the following. Given the parameters $NPred$ and $MinWeight$, we can discover a density-connected set in a two-step approach. First, choose an arbitrary object from the database satisfying the core object condition as a seed. Second, retrieve all objects that are density-reachable from this seed obtaining the density-connected set containing the seed.¹

Lemma 4: Let p be an object in D and $MinWeight(N_{NPred}(p)) = true$. Then the set $O = \{o \in D \mid o \text{ is density-reachable from } p \text{ with respect to } NPred \text{ and } MinWeight\}$ is a density-connected set with respect to $NPred$ and $MinWeight$.

Proof: 1) O is not empty: p is a core object by assumption. Therefore p is density-reachable from p (Lemma 2 (c)). Then p is in O . 2) Maximality: Let $q_1 \in O$ and q_2 be density-reachable from q_1 with respect to $NPred$ and $MinWeight$. Since q_1 is density-reachable from p and density-reachability is transitive with respect to $NPred$ and $MinWeight$ (Lemma 2 (a)), it follows that also q_2 is density-reachable from p with respect to $NPred$ and $MinWeight$. Hence, $q_2 \in O$. 3) Connectivity: All objects in O are density-connected via the object p . \square

Furthermore, a density-connected set C with respect to $NPred$ and $MinWeight$ is uniquely determined by *any* of its core objects, i.e. each object in C is density-reachable from any of the core objects of C and, therefore, a density-connected set

1. As we will see in the next chapter, retrieving all objects that are density-reachable from a core object o is very simple. Starting with o , iteratively collect all objects that are *directly* density-reachable. The directly density-reachable objects are collected by simply retrieving the $NPred$ -neighborhood of objects.

C contains exactly the objects which are density-reachable from an arbitrary core object of C .

Lemma 5: Let C be a density-connected set with respect to $NPred$ and $MinWeight$. Let p be any object in C with $MinWeight(N_{NPred}(p)) = true$. Then C equals the set $O = \{o \in D \mid o \text{ is density-reachable from } p \text{ with respect to } NPred \text{ and } MinWeight\}$.

Proof: 1) $O \subseteq C$ by definition of O . 2) $C \subseteq O$: Let $q \in C$. Since also $p \in C$ and C is a density-connected set, there is an object $o \in C$ such that p and q are density-connected via o , i.e. both p and q are density-reachable from o . Because both p and o are core objects, it follows by symmetry for core objects (lemma 2 (b)) that also object o is density-reachable from p . With the transitivity of density-reachability wrt. $NPred$ and $MinWeight$ (lemma 2 (a)) it follows that q is density-reachable from p . Then $q \in O$. \square

3.2 Generalized Clustering and Some Specializations

3.2.1 Density-Based Decompositions

A generalized density-based clustering or density-based decomposition DBD of a database D with respect to $NPred$ and $MinWeight$ is the set of *all* density-connected sets in D with respect to specific $NPred$ and $MinWeight$ predicates, i.e. all “clusters” from a density-based decomposition DBD are density-connected sets with regard to the *same parameters* $NPred$ and $MinWeight$. A density-based decomposition additionally contains a set called the *noise* relative to the given clustering DBD of D which is simply the set of objects in D not belonging to any of the clusters of DBD . We will use the notation “density-based decomposition” and “generalized density-based clustering” interchangeable and sometimes abbreviate the notions to

“decomposition”, “generalized clustering” or even shorter “clustering” if the meaning is clear from the context.

The formal requirements for a density-based decomposition of a database D are given in the following definition:

Definition 8: (*density-based decomposition*)

A *density-based decomposition DBD* of a database D with respect to $NPred$ and $MinWeight$ is defined by the following conditions:

- 1) $DBD = \{S_1, \dots, S_k; N\}$, $k \geq 0$
- 2) $S_1 \cup \dots \cup S_k \cup N = D$
- 3) For all $i \leq k$:
 S_i is a density-connected set with respect to $NPred$ and $MinWeight$ in D
- 4) If there exists S such that S is a density-connected set in D with respect to $NPred$ and $MinWeight$ then there also exists an $i \leq k$ such that $S = S_i$
- 5) $N = D \setminus (S_1 \cup \dots \cup S_k)$.

The set N is called the *noise* with respect to the decomposition DBD and is denoted by $noise_{DBD}$.

A *density-based decomposition DBD* of a database D with respect to $NPred$ and $MinWeight$ is also denoted as $DBD(D, MinWeight, NPred)$.

According to condition 2) the union of the sets in a density-based decomposition DBD of D is the database D itself. Condition 3) states that each element of the decomposition, except one, is a density-connected set with respect to the given “parameters” $NPred$ and $MinWeight$. Condition 4) means that *all* density-connected sets in D with respect to $NPred$ and $MinWeight$ must be included in a decomposition. Condition 4) simply states that the set N is the “rest” of objects which do not belong to any of the density-connected sets in the decomposition. This set N is not a density-connected set and may also be empty.

There are other possibilities to define what should be considered as a generalized clustering based on definition 7 (density-connected set). However, our notion of a density-based decomposition has the nice property that two clusters or density-connected sets can at most overlap in objects which are border objects in both clusters. This is the content of the following lemma. Figure 21 illustrates the overlap of two clusters using *cardinality* and $MinPts = 4$.

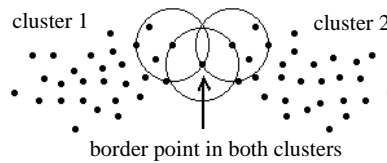


Figure 21: Overlap of two clusters for $MinPts = 4$

Lemma 6: Let DBD be a generalized density-based clustering of D with respect to $NPred$ and $MinWeight$. If $C_1, C_2 \in DBD$ and $C_1 \neq C_2$, then for all $p \in C_1 \cap C_2$ it holds that p is not a core object, i.e. $MinWeight(NPred(p)) = false$.¹

Proof: Since $NPred$ and $MinWeight$ are the same for all clusters in DBD it follows that if $p \in C_1 \cap C_2$ would be a core object for C_1 , then p would also be a core object for C_2 . But then, it follows from Lemma 5 that $C_1 = C_2$ holds, in contradiction to the assumption. Hence, p is not a core object. \square

In the following subsections we will see that “density-based decomposition” is a very general notion which covers familiar structures from clustering, pattern recognition as well as new applications which are appropriate for grouping spatially extended objects such as polygons in geographic information systems.

1. For an ϵ -neighborhood and *cardinality* we even get the stronger result that for $MinPts \leq 3$ there is *no* overlap between clusters. This is true because if a border object o would belong to two different clusters then its ϵ -neighborhood must contain at least 3 objects. Hence, o would be a core object. But then, the two clusters would *not* be different.

We omit the term “with respect to $NPred$ and $MinWeight$ ” in the following whenever it is clear from the context. As already indicated, there are different kinds of objects in a density-based decomposition DBD of a database D : *core objects* (satisfying condition 2 of definition 4) or *non-core objects* otherwise. We refer to this characteristic of an object as the *core object property* of the object. The non-core objects in turn are either *border objects* (no core object but density-reachable from another core object, i.e. member of a density-connected set) or *noise objects* (not a core object and not density-reachable from other objects, i.e. member of the set $noise_{DBD}$).

3.2.2 Specializations

In the following subsections we introduce some specializations of density-based decompositions for several different types of databases. For this purpose, we only have to specify the predicate $NPred$ that defines the neighborhood for objects and the predicate $MinWeight$ for the minimum weight of sets of objects. In the next chapter we will see that all these instances can be detected by the same general algorithmic scheme.

3.2.2.1 Single Link Levels

The clusterings which corresponds to levels in the single-link hierarchy (cf. section 2.2.1) are equivalent to density-based decompositions of the database. A level in the single-link hierarchy determined by a “critical distance” $D_{min} = \varepsilon$ is specified as follows:

- $NPred(o, o')$ iff $|o - o'| \leq D_{min}$
 $N_{NPred}(o) = \{o' \in D \mid |o - o'| \leq D_{min}\}$
- $MinWeight(N) = true$

Remember that the distance between two clusters, i.e. sets of points, for the single link method is defined by the minimum distance between two points in the clusters. Two clusters are merged at a certain level of the single link hierarchy if their minimum interpoint distance is less or equal to the distance associated with the respective level. Therefore, the only requirement for our density-connected sets is that a point q is in the neighborhood of a point p if their distance from each other is less or equal to the “level distance” D_{min} . No special conditions with respect to the predicate *MinWeight* are necessary.

As already mentioned, the well-known “single-link effect” can occur if we use the above definition of a density-connected set. If there is a chain of points between two clusters where the distance of each point in the chain to the neighboring point in the chain is less than or equal to D_{min} , then the two clusters will not be separated. Figure 22 illustrates a clustering determined by a level in the single link hierarchy.

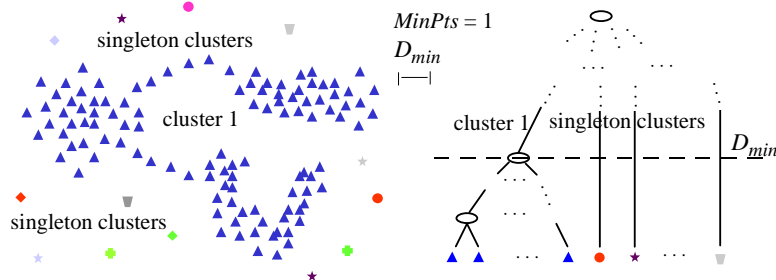


Figure 22: Illustration of single link level

In this example there is one large cluster. By visual inspection, we might say that this cluster consists of three different groups of points. However, these groups of points cannot be separated by the single link method because they are connected by a line of points having an interpoint distance similar to the distances within the three subgroups. To describe the subgroups of points as distinct density-connected

sets, we have to modify the definition of the *MinWeight* predicate (as in the next specialization DBSCAN).

Note that the predicate *MinWeight* could alternatively be defined in the following way to specify a single link level:

- *MinWeight(N)* iff $|N| \geq 1$ or as *MinWeight(N)* iff $|N| \geq 2$ and every point p in the set $noise_{DBD}$ has to be considered as a cluster of its own.

The condition *MinWeight(N)* iff $|N| \geq 1$ is equivalent to '*MinWeight(N) = true*' for neighborhood sets N since a neighborhood set is never empty.

Looking at the second alternative to define the *MinWeight* predicate for a single link clustering level, we can easily see that a level in the single link hierarchy is a very special case of the next specialization of a density-based decomposition.

3.2.2.2 DBSCAN

A density-based clustering as defined for DBSCAN (*cf.* section 2.2.2: Density-Based Approaches) is an obvious specialization of a density-based decomposition since we started our generalization from this instance. A density-based clustering found by DBSCAN is a density-based decomposition determined by the following parameter setting:

- $NPred(o, o')$ iff $|o - o'| \leq \epsilon$
 $N_{NPred}(o) = N_{\epsilon}(o) = \{o' \in D \mid |o - o'| \leq \epsilon\}$
- *MinWeight(N)* iff $|N| \geq MinPts$, i.e. the *cardinality* of N is greater than the density threshold *MinPts*.

Obviously, if we set *MinPts* equal to 1 or 2 in this specialization, we have an equivalence to a level in the single link hierarchy as described in the previous section. In general, however, using higher values for *MinPts* will not be equivalent to

a level in the single-link hierarchy but will avoid or at least significantly weaken the single link effects.

Figure 23 illustrates the effect of using $MinPts = 4$ for the same 2-dimensional dataset as in figure 22. The value for ϵ is depicted and corresponds to the value D_{min} in figure 22. There is a chain of points between the different density-connected sets, but now, these sets are not merged into one single cluster because for the points within the chain the cardinality of the ϵ -neighborhoods is at most three.

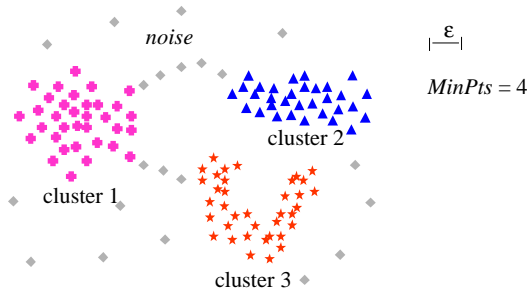


Figure 23: Illustration of DBSCAN result

Our experiments indicate that a value of $2*d$ ($d =$ dimension of the dataspace) for $MinPts$ is suitable for many applications. However, if there is a very large amount of noise in the data or if the database is not really a *set* of objects, i.e. there are objects having identical spatial attributes (see for instance the application in section 5.1), we have to choose a larger value for $MinPts$ to separate some of the meaningful density-connected sets.

3.2.2.3 Density-Based Clustering Using a Grid Partition

In the last chapter we have seen that there are several grid-based approaches to the clustering problem which consider clusters as regions of connected grid cells with a high point density. Clusters which are found by those approaches that define

“high density” by a (typically user-defined) threshold for the point density can be described as density-connected sets. There are different specializations of our definitions depending on the different procedures for finding connected regions of grid cells. We will present two of them which are related to clustering techniques discussed in the previous chapter.

First, the procedure performed by STING (*cf.* section 2.3.5) for finding connected regions of bottom level cells in the STING structure is in principle the same as for DBSCAN, except, that the objects to be clustered are grid cells not points. Let D be the set of all *relevant* bottom layer cells in the STING structure and let l , c , and f denote the side length of bottom layer cells, the specified density, and a small constant number set by STING. Then the regions of bottom layer cells found by STING are density-connected sets with respect to the following predicates $NPred$ and $MinWeight$:

- $NPred(c, c')$ iff $|c - c'| \leq d$, where $d = \max(l, \sqrt{\frac{f}{c\pi}})$
 $N_{NPred}(c) = \{c' \in D \mid |c - c'| \leq d\}$
- $MinWeight(N)$ iff $\frac{\sum_{c \in N} |c|}{|N|} \geq f$, i.e. the average density within the cell area N is greater than the density threshold f .

In the definition of the minimum distance d (see [WYM 97]) usually the side length of l of the cells is the dominant term. As a result, this distance can only reach the neighboring cells. Only when the granularity of the bottom layer grid is very small, the second term may be larger than l and this distance could cover a larger number of cells.

Note that the density-connected sets defined by the above predicates only cover regions of *relevant* cells in the bottom layer of the STING structure. The set D of relevant cells is determined in a top down manner by the STING algorithm before the construction of connected regions begins. A cell is called relevant if it satisfies an additional selection condition. If no such condition is specified, i.e. only the

density condition must be considered, then all cells are “relevant”. In this case, the above specification of density-connected sets by the predicates *NPred* and *MinWeight* also represents a possible definition of clusters for a grid-based clustering approach which is not specific to the STING structure.

The STING algorithm selects certain grid cells using a grid directory. However, we can specify these density-connected sets directly in the set of all bottom layer cells. We simply have to integrate the selection conditions used by the STING algorithm for finding relevant cells into our *NPred* and *MinWeight* predicates. For this purpose, let D now be the set of all bottom layer cells, l , c , and f as before, and let S be a selection condition using the non-spatial attributes of the STING cells (e.g. distribution type), i.e. $S(c)$ holds if the cell c satisfies the condition S . We can specify the connected regions of STING as density-connected sets in the following way:

- *NPred*(c, c') iff ($S(c) \Leftrightarrow S(c')$) and $|c - c'| \leq d$, where $d = \max(l, \sqrt{\frac{f}{c\pi}})$
 $N_{NPred}(c) = \{c' \in D \mid NPred(c, c')\}$
- *MinWeight*(N) iff for all $c \in N$: $S(c)$ and $\frac{\sum_{c \in N} |c|}{|N|} \geq f$, i.e. the average density within the cell area N is greater than the density threshold f and the cells satisfy the selection condition S .

There are several other possibilities to define density-connected sets for grid cells based on point density. Perhaps the most simple version is given by an *NPred* definition which defines neighboring cells by having a common face, and the predicate *MinWeight* using a density threshold f for single cells. The neighborhood of cells can be expressed simply by using the grid indices of the cells: Assume, that we have a d -dimensional grid that partitions the space along a number of split lines m_1, \dots, m_d in each dimension. Then each cell c can be addressed by a vector $adr(c) = (c_1, \dots, c_d)$, where $1 \leq c_i \leq m_i$ for each $i \leq d$. To connect only cells c having a higher point count than f , i.e. $|c| \geq f$, we have to integrate this condition into the definition of the neighborhood above - analogous to the integration of a selection

condition S in the $NPred$ definition for STING. The formal specification for this kind of density-based clusters is as follows: Let D be the set of all cells in the grid and let $\text{adr}(c)=(c_1, \dots, c_d)$ and $\text{adr}(c')=(c'_1, \dots, c'_d)$ be the addresses of the two cells c and c' .

- $NPred(c, c')$ iff $(|c| \geq f \Leftrightarrow |c'| \geq f)$ and there is an i such that for all $j \neq i: c_j = c'_j$ and $c_i = c'_i + 1$ or $c_i = c'_i - 1$. $N_{NPred}(c) = \{c' \in D \mid NPred(c, c')\}$.
- $MinWeight(N)$ iff $|c| \geq f$ for all $c \in N$, i.e. the density of each cell within the area N is greater than the density threshold f .

Note that although there is a universal quantifier in the definition of the predicate $MinWeight$, we do not have to check the density condition for each element of a neighborhood set $N_{NPred}(c)$ to ensure that $MinWeight(N_{NPred}(c))$ holds. From our neighborhood definition it follows that if one cell c' in the set $N_{NPred}(c)$ satisfies the condition $|c'| \geq f$, then the condition holds for all cells in the set.

If the high-density cells are selected in a separate step, before connected regions are constructed - like, for instance, in the CLIQUE algorithm (see section 2.2.2, density-based approaches) - the set D is the set of all cells c already satisfying the condition $|c| \geq f$. Then, the requirement $(|c| \geq f \Leftrightarrow |c'| \geq f)$ and $|c| \geq f$ in the last definition of $NPred$ and $MinWeight$ can be omitted to specify connected regions of such cells. As a consequence, the definition of $MinWeight$ reduces to 'true'. Figure 24 depicts an example of this kind of grid-based clustering for the same dataset as in figure 22 and figure 23. The figure shows connected regions of grid cells using a density-threshold $f=3$ for single cells.

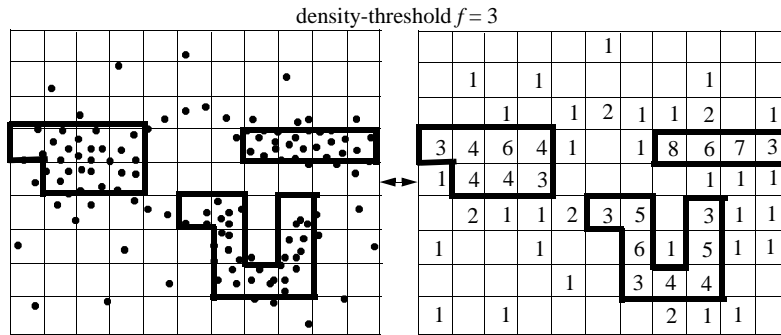


Figure 24: Illustration of density-based clustering using a grid approach

3.2.2.4 Simple Forms of Region Growing

Having seen the specializations to grid-based clustering approaches, it is easy to recognize that there are also specializations of density-based decompositions which are equivalent to structures found by simple forms of region growing (see section 2.2.3). The pixels in an image can be considered as grid cells and the gray-level intensity values correspond to the point densities for grid cells. The difference, however, between clustering based on grid cell densities and region growing is, that for region growing there is not a single density-threshold value to select “relevant” cells. All pixels except background pixels are relevant, and we have to distinguish different gray-levels, i.e. different classes of pixels, to form the connected regions. Pixels are only connected if they are adjacent and have *similar* attribute values, e.g. gray-level intensity.

For the formal specification of connected regions of pixels in an image, the degree of similarity required for the attributes of pixels may be expressed by a value $t \in \mathbf{R}$. The neighborhood of a pixel may then be given by the neighboring pixels in the image having similar attribute values, and the *MinWeight* predicate may be

used to exclude regions in the image that contain only background pixels, i.e. pixels p with $\text{gray-level}(p) = 0$.

Let D be the set of all pixels in the image, $t \in \mathbf{R}$ be a threshold value for the similarity of gray-level values, and let (x_p, y_p) be the coordinates of a pixel p . Then, a connected region of pixels in an image, as produced by a simple region growing algorithm, is a density-connected set with respect to the following *NPred* and *MinWeight* predicates:

- *NPred*(p, p') iff $\text{gray-level}(p) \geq 0$, $\text{gray-level}(p') \geq 0$, $|\text{gray-level}(p) - \text{gray-level}(p')| \leq t$, $|x_p - x_{p'}| \leq 1$ and $|y_p - y_{p'}| \leq 1$, i.e. the gray-levels of pixels p and p' are similar and the pixels are not background pixels and are adjacent.

$$N_{NPred}(p) = \{p' \in D \mid NPred(p, p')\}$$

- *MinWeight*(N) = true

Figure 25 illustrates regions in a raster image which have different gray-level intensities and which can be considered as density-connected sets.

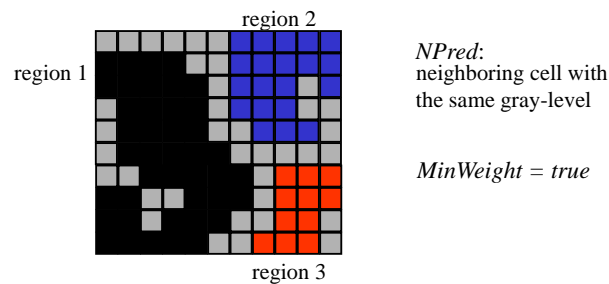


Figure 25: Illustration of simple region growing

3.2.2.5 Clustering polygonal objects

A further specialization of density-connected sets allows the clustering of spatially extended objects such as polygons. We already mentioned this type of application

in the motivation for our generalization of density-based clusters (section 3.1.1). A most simple definition of an $NPred$ and $MinWeight$ predicate for describing cluster-like groups of polygons uses intersection for the neighborhood of polygons and compares the area of the region of connected polygons to a user-specified threshold value $MinArea$. More formally, if D is a set of polygons, then a density-connected set of polygons in D is given by the following $NPred$ and $MinWeight$ predicates:

- $NPred(p, p')$ iff $intersects(p, p')$
 $N_{NPred}(p) = \{p' \in D \mid NPred(p, p')\}$
- $MinWeight(N)$ iff $\sum_{p \in N} area(p) \geq MinArea$

To take into account other non-spatial attributes of polygons, for example attributes used in a geographic information system (see e.g. the application in section 5.4), we can simply integrate corresponding “selection” conditions into the definition of the $NPred$ and/or the $MinWeight$ predicates. In the definitions of the $NPred$ and $MinWeight$ predicates for grid-based clustering and region growing, we have already seen examples of how to use selection conditions. Integrating such conditions into the definition of the $NPred$ and $MinWeight$ predicates for polygons can be accomplished in the same way. Figure 26 illustrates an example for the clustering of polygonal objects using the simple $NPred$ and $MinWeight$ predicates as specified above.

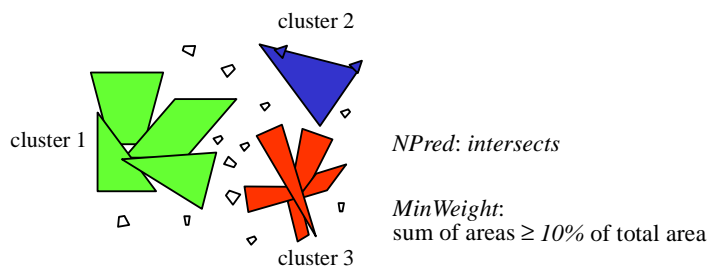


Figure 26: Illustration of clustering polygonal objects

3.3 Determining Parameters

GDBSCAN requires a neighborhood predicate $NPred$ and a predicate for the minimum weight of sets of objects, $MinWeight$. Which parameters will be used for a data set depends on the goal of the application. Though, for some applications it may be difficult to determine the correct parameters, we want to point out that in some applications there may be a natural way to provide values without any further parameter determination, i.e. there is a natural notion of a neighborhood for the application which does not require any further parameter estimation (e.g. *intersects* for polygons). In other cases, we may only know the *type* of neighborhood that we want to use, for example, a distance based neighborhood for the clustering of point objects. Parameters may be re-used in different but similar applications, e.g., if the different datasets are produced by a similar process. And, we will see in chapter 5 that there are even applications where the appropriate parameter values for GDBSCAN can be derived analytically (e.g. section 5.2).

In case of a distance based neighborhood combined with a $MinWeight$ predicate which compares cardinality to a threshold value, we can use a simple but in most cases effective heuristic to determine the specific distance and threshold values that are most suitable for the clustering application. This simple heuristic which is effective in many cases to determine the parameters ϵ and $MinPts$ for DBSCAN (cf. section 3.2.2.2) - which is the most important specialization of GDBSCAN - is presented in the following subsection.

Heuristic for DBSCAN Using a k -distance Plot

DBSCAN uses a distance based neighborhood “distance less or equal than ϵ ” and the comparison of the cardinality of an ϵ -neighborhood to a threshold ($MinPts$) for the $MinWeight$ predicate. Thus, we have to determine appropriate values for ϵ and $MinPts$. The density parameters of the “thinnest”, i.e. least dense, cluster in the da-

tabase are good candidates for these global values specifying the lowest density which is not considered to be noise.

For a given $k \geq 1$ we define a function k -distance, mapping each object to the distance from its k -th nearest neighbor. When sorting the objects of the database in descending order of their k -distance values, the plot of this function gives some hints concerning the density distribution in the database. We call this plot the *sorted k -distance plot* (see figure 27 for an example).

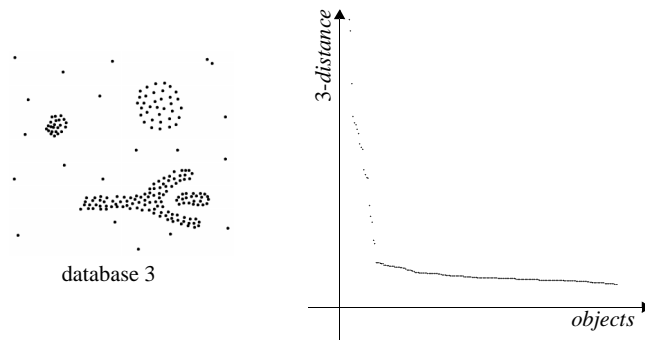


Figure 27: Sorted 3-distance plot for sample database 3

If we choose an arbitrary object p , set the parameter ϵ to k -distance(p) and the parameter $MinPts$ to $k+1$, all objects with an equal or smaller k -distance value are core objects, because there are at least $k+1$ objects in an ϵ -neighborhood of an object p if ϵ is set to k -distance(p). If we can now find a *threshold object* with the maximum k -distance value in the “thinnest” cluster of D , we would obtain the desired parameter values. Therefore, we have to answer the following questions:

- 1) Which value of k is appropriate?
- 2) How can we determine a threshold object p ?

We will discuss the value k first, assuming it is possible to set the appropriate value for ϵ . The smaller we choose the value for k , the lower are the computational costs to calculate the k -distance values and the smaller is the corresponding value for ϵ in general. But a small change of k for an object p will in general only result in a small change of k -distance(p). Furthermore, our experiments indicate that the k -distance plots for “reasonable” k (e.g. $1 \leq k \leq 10$ in $2d$ space) do not significantly differ in shape and that also the results of DBSCAN for the corresponding parameter pairs (“distance $\leq \epsilon$ ”, “cardinality of ϵ -neighborhood $\geq k$ ”) do not differ very much. Therefore, the choice of k is not very crucial for the algorithm. We can even fix the value for k (with respect to the dimension of the dataspace) eliminating the parameter involving *MinPts* for DBSCAN. Considering only the computational cost, we would like to set k as small as possible. On the other hand, if we set $k = 1$, the k -distance value for an object p will be the distance to the nearest neighbor of p and the result will be equivalent to a level in the single-link hierarchy (cf. section 3.2.2.1). To weaken the “single-link effect”, we must choose a value for $k > 1$.

We propose to set k to $2 * dimension - 1$. Our experiments indicate that this value works well for databases D where each object occurs only once, i.e. if D is really a set of objects. Thus in the following, if not stated otherwise, k will be set to this value, and the value for *MinPts* will be fixed according to the above strategy ($MinPts = k + 1$, e.g. $MinPts = 4$ in $2d$ space).

To determine ϵ , we have to know an object in the “thinnest” cluster of the database with a high k -distance value for that cluster. Figure 27 shows the sorted 3-distance plot for sample database 3 (cf. figure 15) which is very typical for databases where the density of clusters and the density of noise differ significantly. Our experiments indicate that the threshold object is an object near the first “valley” of the sorted k -distance plot. All objects with a higher k -distance value (to the left of

the threshold) will then be noise, all other objects (to the right of the threshold) will be assigned to some cluster (see figure 28 for an illustration).

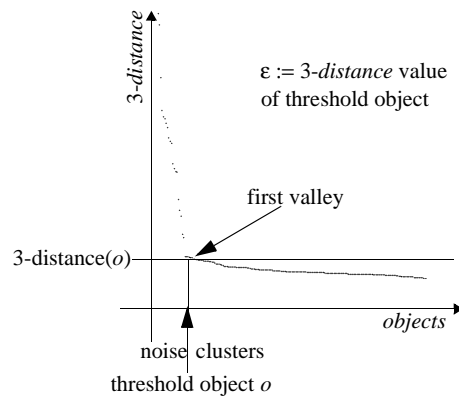


Figure 28: Determining ϵ using the sorted 3-distance plot

In general, it is very difficult to detect the first “valley” automatically, but it is relatively simple for a user to recognize this valley in a graphical representation. Additionally, if the user can estimate the percentage x of noise, a proposal for the threshold object can be derived, because we know that most of the noise objects have a higher k -distance value than objects of clusters. The k -distance values of noise objects are located on the left of the k -distance plot, so that we have to select an object after x percent of the sorted k -distance plot.

There is in general a *range* of values for the parameter ϵ that yield the same clustering because not all objects of the “thinnest” cluster need to be core objects. They will also belong to the cluster if they are only density-reachable. Furthermore, the ϵ -value may be larger than needed if the clusters are well separated and the density of noise is clearly lower than the density of the thinnest cluster. Thus, the robustness of the parameter determination, i.e. the width of the range of appropriate ϵ -values, depends on the application. However, in general the width of this range is wide enough to allow the parameters to be determined in a sorted k -distance plot

for only a very small sample of the whole database (1% - 10%). Figure 29 depicts the range for ϵ yielding the same clustering for sample database 3 in the sorted 3-distance plots for 100% and a 10% sample of the database.

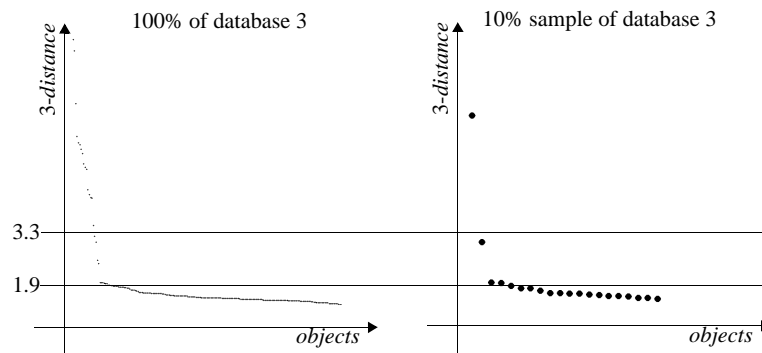


Figure 29: Range for ϵ in sorted 3-distance plot yielding the same clustering

Obviously, the shape of the sorted k -distance plot and hence, the effectiveness of the proposed heuristic depends on the distribution of the k -nearest neighbor distances. For example, the plot will look more “stairs-like” if the objects are distributed regularly within clusters of very different densities or the first “valley” will be less clear if the densities of the clusters differ not much from the density of noise (which also means that the clusters are not well separated). Then, knowing the approximate percentage of noise in the data may be helpful.

Figure 30 illustrates the effects of different cluster densities and unclear cluster borders on the k -distance plot for two example databases using $k=3$. The arrows in the figure point at different 3-distance values indicating which of the depicted clusters A, B, C or D are density-connected sets if we use $MinPts = 4$ and set ϵ equal to the respective 3-distance values.

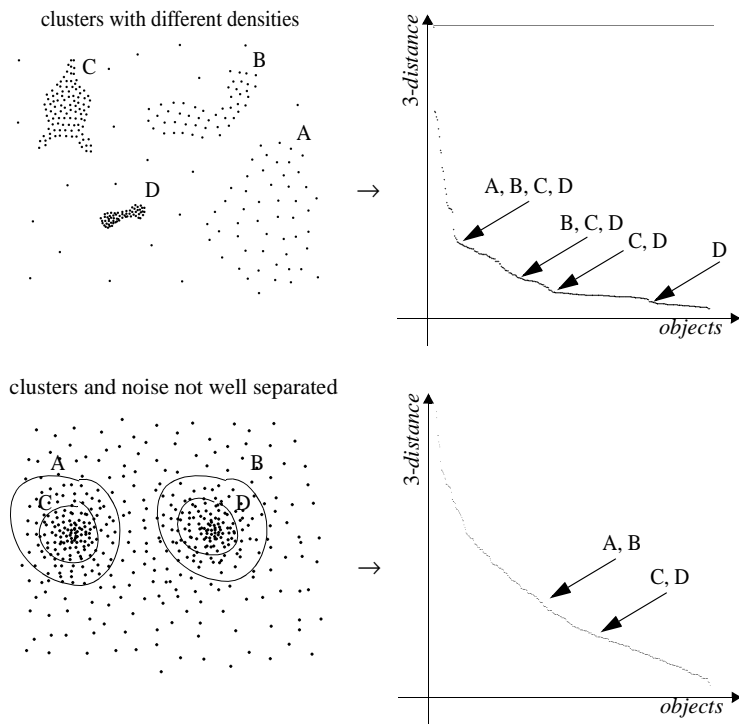


Figure 30: Examples of sorted 3-distance plots for different data distributions

To conclude, we propose the following interactive approach for determining the parameters for DBSCAN:

- The user gives a value for k (default value is $k = 2 * dimension - 1$).
- The system computes and displays the k -distance plot for a small sample of the database.
- The user selects an object as the threshold object and the k -distance value of this object is used as the ϵ -value (if the user can estimate the percentage of noise, the system can derive a proposal for the threshold object from it).
- $MinPts$ is set to $k+1$.

3.4 Summary

In this chapter, we have seen that the notion of a density-based clustering can be generalized to cover structures produced by different clustering algorithms, patterns recognized by region growing algorithms as well as “connected” groups of objects of - in principle - arbitrary data-type satisfying certain conditions, for instance polygons from a geographic information system. Some of these instances have been discussed in greater detail.

The generalization of a density-based clustering, i.e. a density-based decomposition, was introduced formally as a set of density-connected sets plus a set of outliers, called ‘noise’. Density-connected sets have been defined with respect to two “parameters”: a neighborhood predicate $NPred$ which has to be symmetric and reflexive, and a predicate $MinWeight$ for the minimum weight of neighborhood sets. These predicates are necessary to define the notions of density-reachability and density-connectedness which in turn are needed to define the notion of a density-connected set for objects of arbitrary data-type.

A density-connected set was defined as a set of density-connected objects which is maximal with respect to density-reachability. Two objects p and q from a database D are density-connected in D if there is an object o in D such that p and q are both density-reachable from o (directly or transitive). An object p is called directly density-reachable from an object q if p is in the $NPred$ -neighborhood of q and this $NPred$ -neighborhood of q has minimum weight, i.e. $MinWeight(N_{NPred}(q))$ holds.

We also discussed in this chapter how to determine additional parameters which may be required by some specializations. Then a simple but in most cases effective heuristic was presented to determine these additional parameters for a density-based clustering as defined for DBSCAN, which is the most important specialization of a density-based decomposition.

Chapter 4

GDBSCAN: An Algorithm for Generalized Clustering

In this chapter, the algorithm GDBSCAN to construct density-based decompositions is introduced (section 4.1). This is basically an algorithmic schema which is independent from the specific predicates for the neighborhood of objects, and the minimum weight for sets of objects. We shortly discuss some implementation issues (section 4.2), and present both an analytical and experimental performance evaluation (section 4.3). After that, we introduce and evaluate some advanced database techniques, i.e. neighborhood indices and multiple neighborhood queries, to support not only our algorithm but also a broader class of spatial data mining applications of which GDBSCAN is just a special member (section 4.4).

4.1 Algorithmic Schema GDBSCAN

To find a density-connected set, GDBSCAN starts with an arbitrary object p and retrieves all objects density-reachable from p with respect to $NPred$ and $MinWeight$. Density-reachable objects are retrieved by performing successive $NPred$ -neighborhood queries and checking the minimum weight of the respective results. If p is a core object, this procedure yields a density-connected set with respect to $NPred$ and $MinWeight$ (see lemmata 4 and 5). If p is not a core object, no objects are density-reachable from p and p is assigned to *NOISE*. This procedure is iteratively applied to each object p which has not yet been classified. Thus, a density-based decomposition and the noise according to definition 8 is detected.

In figure 31, we present a basic version of GDBSCAN omitting details of data types and generation of additional information about clusters:

```
GDBSCAN (SetOfObjects, NPred, MinWeight)
// SetOfObjects is UNCLASSIFIED; Object.Processed = FALSE
ClusterId := nextId(NOISE);
FOR i FROM 1 TO SetOfObjects.size DO
  Object := SetOfObjects.get(i);
  IF NOT Object.Processed THEN
    IF ExpandCluster(SetOfObjects, Object, ClusterId, NPred, MinWeight)
      THEN ClusterId := nextId(ClusterId)
    END IF
  END IF
END FOR;
END; // GDBSCAN
```

Figure 31: Algorithm GDBSCAN

`SetOfObjects` is either the whole database or a discovered cluster from a previous run. `NPred` and `MinWeight` are the global density parameters. The cluster identifiers are from an ordered and countable data-type (e.g. implemented by Integers)

where UNCLASSIFIED < NOISE < “other Ids”, and each object will be marked with a cluster-id `Object.CId`. The function `nextId(clusterId)` returns the successor of `clusterId` in the ordering of the data-type (e.g. implemented as `Id := Id+1`). The function `SetOfObjects.get(i)` returns the i -th element of `SetOfObjects`. In figure 32, function `ExpandCluster` constructing a density-connected set for a core object `Object` is presented in more detail.

```

ExpandCluster(SetOfObjects, Object, CId, NPred, MinWeight):Boolean;
neighbors := SetOfObjects.neighborhood(Object, NPred);
Object.Processed := TRUE;
IF MinWeight(neighbors) THEN // object is a core object
  Seeds.init(NPred, MinWeight, CId);
  Seeds.update(neighborhood, Object);
  WHILE NOT Seeds.empty() DO
    currentObject := Seeds.next();
    neighbors := SetOfObjects.neighborhood(currentObject, NPred);
    currentObject.Processed := TRUE;
    IF MinWeight(neighbors) THEN
      Seeds.update(neighbors, currentObject);
    END IF; // MinWeight(neighbors)
  END WHILE; // seeds.empty()
  RETURN True;
ELSE // Object is NOT a core object
  SetOfObjects.changeCId(Object, NOISE);
  RETURN False;
END IF; // MinWeight(neighbors)
END; // ExpandCluster

```

Figure 32: Function ExpandCluster

A call of `SetOfObjects.neighborhood(Object, NPred)` returns the $NPred$ -neighborhood of `Object` in `SetOfObjects` as a set of objects (`neighbors`). If the $NPred$ -neighborhood of `Object` has minimum weight, the objects from this $NPred$ -neighborhood are inserted into the set `Seeds` and the function `ExpandCluster` succes-

sively performs $NPred$ -neighborhood queries for each object in `Seeds`, thus finding all objects that are density-reachable from `Object`, i.e. constructing the density-connected set that contains the core object `Object`.

The class `Seeds` controls the main loop in the function `ExpandCluster`. The method `Seeds.next()` selects the next element from the class `Seeds` and deletes it from the class `Seeds`. The method `Seeds.update(neighbors, centerObject)` inserts into the class `Seeds` all objects from the set `neighbors` which have not yet been considered, i.e. which have not already been found to belong to the current density-connected set. This method also calls the method to change the cluster-id of the objects to the current `clusterId`. Figure 33 presents the pseudo-code for the method `Seeds.update`.

```

Seeds::update(neighbors, CenterObject);
SetOfObjects.changeCId(CenterObject,CId);
FORALL Object FROM neighbors DO
  IF NOT Object.Processed THEN
    Object.Processed := TRUE;
    insert(Object);
  END IF; // Object is "new"
  IF Object.CId IN {UNCLASSIFIED, NOISE} THEN
    SetOfObjects.changeCId(Object,CId);
  END IF; // Object is UNCLASSIFIED or NOISE
END FORALL;
END; // Seeds::update

```

Figure 33: Method `Seeds::update()`

In this version of GDBSCAN it does not matter in which order the elements are inserted or selected from the class `Seeds`. We may use for instance a *stack* or alternatively a *queue* to implement the class `Seeds` without changing the result of the algorithm. In all cases, the principle idea of the algorithm, i.e. the kind of procedure that is performed to construct connected groups of “neighboring” objects, is similar to the idea of a region growing algorithm. Note, however, that region growing al-

gorithms are highly specialized to pixels in an image and therefore presuppose a “grid-based” neighborhood, whereas density-connected sets can be defined for any data type.

The cluster-id of some objects p which are marked as NOISE because they do not have the minimum weight may be changed later if they are density-reachable from some other object of the database. This may happen only for border objects of a cluster. Those objects are then not added to `Seeds` because we already know that an object with a cluster-id of NOISE is not a core object, i.e. no other objects are density-reachable from them.

If two clusters C_1 and C_2 are very close to each other, it might happen that some object p belongs to both C_1 and C_2 . Then p must be a border object in both clusters according to Lemma 6. In this case, object p will only be assigned to the cluster discovered first. Except from these rare situations, the result of GDBSCAN is independent of the order in which the objects of the database are visited due to lemmata 4 and 5.

Obviously, the efficiency of the above algorithm depends on the efficiency of the neighborhood query because such a query is performed exactly once for each object in `SetOfObjects`. The performance of GDBSCAN will be discussed in detail in section 4.3. There, we will see that neighborhood predicates based on spatial proximity like distance predicates or intersection can be evaluated very efficiently by using spatial index structures.

There may be reasons to apply a post-processing to a clustering obtained by GDBSCAN. According to definition 8, each set of objects having *MinWeight* is a density-connected set. In some applications (see for example chapter 5), however, density-connected sets of this minimum size are too small to be accepted as clusters. Furthermore, GDBSCAN produces clusters and noise. But for some applications a non-noise class label for each object is required. For this purpose, we can re-assign each noise object and each object of a rejected cluster to the closest of the

accepted clusters. This post-processing requires just a simple scan over the whole database without much computation, in particular no region queries are necessary. Therefore, such post-processing does not increase the run-time complexity of GDBSCAN.

Specializations of the algorithmic schema GDBSCAN could be defined for all the parameter specializations introduced in the previous chapter (see section 3.2). In general, we will specify instances of GDBSCAN simply by introducing the parameter specializations whenever it is needed. We only name one specialization explicitly which is DBSCAN ([EKSX 96]), because this is the most important specialization with respect to our applications.

Definition 9: (*DBSCAN*)

DBSCAN is a specialization of the algorithm GDBSCAN using the parameter specializations introduced in section 3.2.2.2, i.e.

- *NPred*: “distance $\leq \epsilon$ ”
- *MinWeight(N)*: $|N| \geq \text{MinPts}$.

4.2 Implementation

In this section, we shortly discuss some implementation issues. The algorithm GDBSCAN from the previous section is an algorithmic schema which needs to be specialized for different databases/data-types, special neighborhood predicates, and predicates to determine the minimum weight of neighborhood sets. Therefore, to provide a flexible, easily extendible and portable implementation of the algorithmic schema GDBSCAN, we choose an object-oriented programming approach using the C++ programming language and the LEDA¹ library. The code has been de-

1. “Library of Efficient data-types and Algorithms”. For documentation and code see <http://www.mpi-sb.mpg.de/LEDA>

veloped and tested on HP workstations under HP-UX 10.X using g++ 2.7.X, and on Intel Pentium PCs under Win95/NT using Borland C++ 5.01. The graphical user-interface needs LEDA, version 3.6, which is available for these (and other) platforms.

Figure 34 illustrates the main parts interacting in the generalized clustering procedure using the algorithmic schema GDBSCAN.

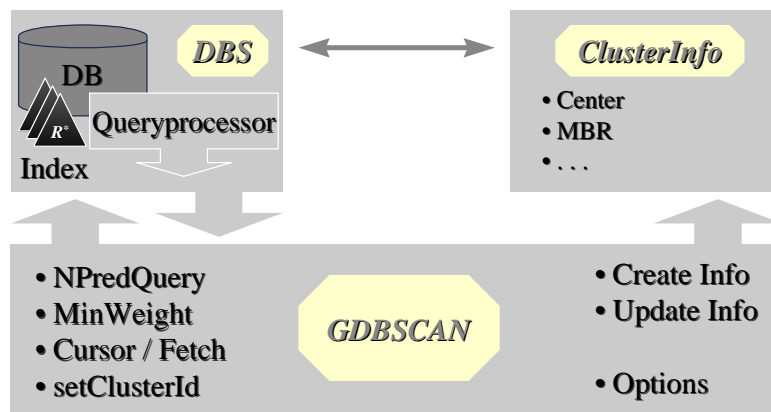


Figure 34: Software Architecture of GDBSCAN

The major components are:

- a (maybe external) database DBS storing objects of a specific data type, providing indices and query-processors.
- a ClusterInfo component responsible for further information about the discovered density-connected sets, and for the presentation of results.
- GDBSCAN, the central component for the general clustering procedure.

GDBSCAN interacts with the database to access objects, retrieve neighborhoods and to store the generated information about cluster membership for single objects. GDBSCAN also interacts with ClusterInfo to provide the information

needed to generate summary descriptions of density-connected sets, for instance, centers and minimum bounding boxes in case of d -dimensional data points. Such additional information about a generalized clustering may also be stored in and retrieved from the external database.

The three components roughly correspond to three main classes in our implementation: GDBSCAN_DBS, CLUSTER_INFO, and GDBSCAN. There is only one more important class needed to make our algorithm independent from a specific data type: GDBSCAN_DATA. Usually the database has its own internal data type that is returned by its functions. This data type might contain more information than GDBSCAN requires (only a few fields). So another data type is created for GDBSCAN. This data type can be also used to provide, for example, a drawing function for visualization, an info function to let the user retrieve information online and so on.

To model and to implement the interaction of the basic classes GDBSCAN_DATA, GDBSCAN_DBS, CLUSTER_INFO, and GDBSCAN, the following object-oriented concepts and “design patterns” were used (see [GHJ+ 95] for a detailed catalog of many object-oriented design patterns):

- *Abstract Classes / Templates*

Abstract Classes / Templates are used to define the skeleton of an algorithm. Subclasses have to define the functions specific to the particular application. This also means that the base classes cannot be instantiated directly but have to be derived first. The template parameter $\langle \text{DATA} \rangle$, modeling the objects which are processed by the algorithm GDBSCAN, is common to all our template classes. This allows the same code to operate on different data types.

- *Class Adapter*

Creates a compatible interface by combining classes using multiple inheritance. In our case this is the interface to a database because the interface provided by a DBMS itself may be different for different database systems.

- *Object Adapter*

Allows to exchange the adaptee at run-time, using a pointer to the adapted class. This pattern is used to allow classes derived from GDBSCAN_DBS as well as different CLUSTER_INFO classes to be chosen and exchanged at run-time.

In the following, the above mentioned classes are shortly described. For this purpose, only the most important public methods and fields are presented.

- **GDBSCAN_DATA** (*abstract class*)

The data objects passed to GDBSCAN should be derived from this class. All fields/methods used by GDBSCAN are defined here. These are

- ObjectId to identify objects
- ClusterId to store the cluster membership of an object
- compare(GDBSCAN_DATA) to compare two objects. This function is defined purely virtual making this class abstract. Usually compare should return zero if the compared data objects occupy the same location in the feature space. If this case is excluded by the semantics of the specializations, this function is not needed and can then be redefined to do nothing.

- **GDBSCAN_DBS** (*abstract class / template <class DATA>*)

This class provides the basis for the connection/interface to a database:

- GDBSCAN_cursor() and GDBSCAN_fetch() are methods to enable the algorithm to scan each object of the database.
- GDBSCAN_set_clusterId(DATA, CLUSTER_ID) is a method that sets the cluster-id of an object of type DATA to an identifier of type CLUSTER_ID.
- GDBSCAN_MinWeight(DataList) is the method which evaluates the predicate *MinWeight* for a neighborhood set, given by a list of type DataList.

- GDBSCAN_SET_Neighborhood() selects the NPred-neighborhood to be used for the generalized clustering. If the selected neighborhood requires additional parameters, methods must also be implemented to determine them; at least the user must be asked for the values of these parameters.
- GDBSCAN_GET_Neighborhood(DATA, DataList) is the central method which implements an NPred-neighborhood. It returns a list (DataList) of all objects satisfying the neighborhood predicate.

• **CLUSTER_INFO** (*abstract class / template <class DATA>*)

This class can be used to collect additional information about clusters, such as centers and minimum bounding rectangles, during the scanning process.

- createTemp(newId) is called whenever a new cluster is found to create a new entry of some sort.
- updateTemp(DATA) is called for each object assigned to a cluster to update the entry. Online visualization during the scan can be accomplished through a clusterInfo class that not only updates its internal state by this method, but also draws a visualization of the object.
- commitTemp() closes the entry. Entries may be saved, for instance in the external database.

• **GDBSCAN** (*template <class DATA>*)

This class provides the actual algorithm:

- GDBSCAN(DB, clusterInfo) is the constructor for the class. It takes two arguments and sets the database used by GDBSCAN equal to DB and the clusterInfo equal to clusterInfo. These settings can be changed anytime by using the following two methods:

GDBSCAN_set_database(DB)

GDBSCAN_set_clusterInfo(clusterInfo).

- GDBSCAN_doGDBSCAN() scans the whole database for clusters.
- GDBSCAN_expand_cluster(DATA) tries to construct a density-connected set starting with DATA as first origin of an $NPred$ -neighborhood query.

Figure 35 depicts the graphical user interface of our implementation¹ and illustrates the setting of parameters for GDBSCAN in this environment.

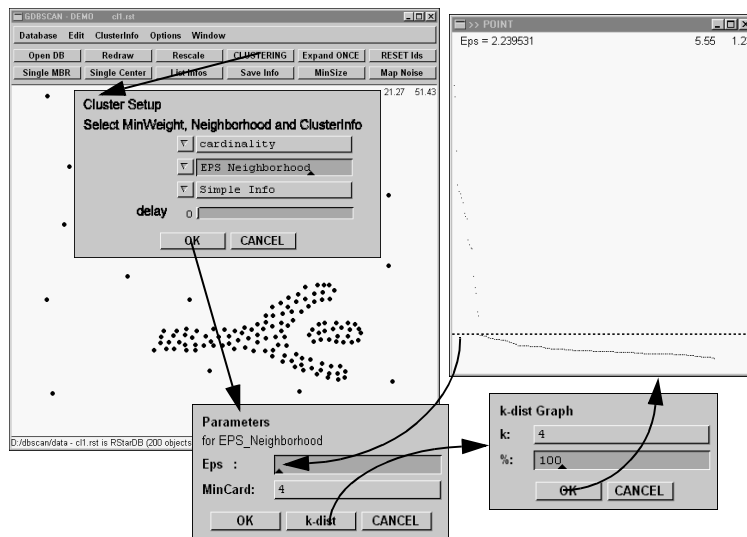


Figure 35: Graphical user interface and parameter setting

1. This implementation of GDBSCAN is available from the author. Request should be made by sending an e-mail to sander@dbs.informatik.uni-muenchen.de.

4.3 Performance

In this section, we evaluate the performance of GDBSCAN. In section 4.3.1 we discuss the performance of GDBSCAN with respect to the underlying spatial index structure. In section 4.3.2 an experimental evaluation and a comparison with the well-known clustering algorithms CLARANS and BIRCH is presented.

4.3.1 Analytical Evaluation

The run-time of GDBSCAN obviously is $O(n \cdot \text{run-time of a neighborhood query})$: n objects are visited and exactly one neighborhood query is performed for each of them. The number of neighborhood queries cannot be reduced since a cluster-id for each object is required. Thus, the overall run-time depends on the performance of the neighborhood query. Fortunately, the most interesting neighborhood predicates are based on spatial proximity - like distance predicates or intersection - which can be efficiently supported by spatial index structures. For complex objects like polygons, we can perform additional filter-refinement steps. These filters typically use approximations such as minimum bounding rectangles to reduce the number of objects for which an expensive test, e.g. the intersection of polygons, will be performed on the exact and eventually very complex geometry. Such a multi-step filter-refinement procedure using at least a spatial index structure (see figure 36) is assumed to be available in an SDBS for efficient processing of several types of spatial queries (see e.g. [BKSS 94]).

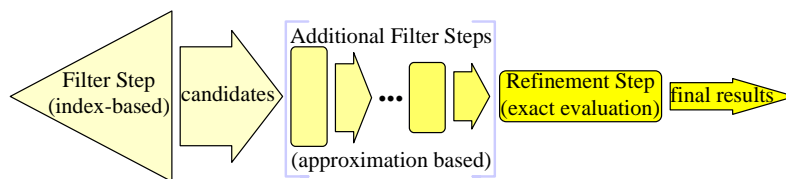


Figure 36: Multi-step filter-refinement procedure for spatial query processing

Table 1 lists the run-time complexity of GDBSCAN with respect to the underlying spatial index structure.

run-time complexity of:	- a single neighborhood query	- the GDBSCAN algorithm
without index	$O(n)$	$O(n^2)$
with spatial index	$O(\log n)$	$O(n * \log n)$
with direct access	$O(1)$	$O(n)$

Table 1: Run-time complexity of GDBSCAN

Without any index support, to answer a neighborhood query a scan through the whole database has to be performed. Thus, the run-time of GDBSCAN would be $O(n^2)$. This does not scale well with the size of the database. But if we use a tree-based spatial index like the R*-tree, the run-time is reduced to $O(n \log n)$: the height of an R*-tree is $O(\log n)$ for a database of n objects in the worst case. At least in low-dimensional spaces, a query with a “small” query region has to traverse only a limited number of paths in the R*-tree. Since most $NPred$ -neighborhoods are expected to be small compared to the size of the whole database, the average run-time complexity of a single neighborhood query using such an index structure is then $O(\log n)$. Furthermore, if we have a direct access to the $NPred$ -neighborhood, e.g. if the objects are organized in a grid, the run-time is further reduced to $O(n)$ because in a grid the complexity of a single neighborhood query is $O(1)$.

Techniques to improve the performance of GDBSCAN further by exploiting special properties of the algorithm are presented in section 4.4. Such techniques are important, especially for very complex objects where the refinement step of a neighborhood query is very expensive or in very high-dimensional spaces where the query performance of all known spatial index structures degenerates.

4.3.2 Experimental Evaluation

GDBSCAN is implemented in C++ allowing various *NPred*-neighborhoods and *MinWeight* predicates to be specified. In addition, different index structures to support the processing of the *NPred*-neighborhood queries can be used.

The following experiments were run on HP 735 / 100 workstations. In order to allow a comparison with CLARANS and BIRCH - which both use a distance based neighborhood definition - the specialization to DBSCAN (*cf.* definition 9) is used. Processing of neighborhood queries is based on an implementation of the R*-tree ([BKSS 90]). For an evaluation of other instances of GDBSCAN see the applications in chapter 5.

To compare DBSCAN with CLARANS in terms of effectiveness (accuracy), our three synthetic sample databases are used which are depicted in figure 15. Since DBSCAN and CLARANS are clustering algorithms of different types, they have no common quantitative measure of the classification accuracy. Therefore, we evaluate the accuracy of both algorithms by visual inspection. In sample database 1, there are four ball-shaped clusters of significantly differing sizes. Sample database 2 contains four clusters of non-convex shape. In sample database 3, there are four clusters of different shape and size with a small amount of additional noise. To show the results of both clustering algorithms, we visualize each cluster by a different color (grayscale). For the result of CLARANS, we also indicate the cluster centers and the corresponding partition of the data space.

For CLARANS, we provided the correct number of clusters, i.e. we set the parameter k (number of clusters) to 4 for these sample databases. For DBSCAN, the parameter ϵ is set, giving a noise percentage of 0% for sample databases 1 and 2, and 10% for sample database 3, respectively (see the heuristic described in the previous section).

The clusterings discovered by CLARANS are depicted in figure 37, the clusterings discovered by DBSCAN are depicted in figure 38. DBSCAN discovers all clusters and the noise points (according to definition 8) from all sample databases. CLARANS, however, splits clusters if they are relatively large or if they are close to some other cluster. Furthermore, CLARANS has no explicit notion of noise. Instead, all points are assigned to their closest medoid.

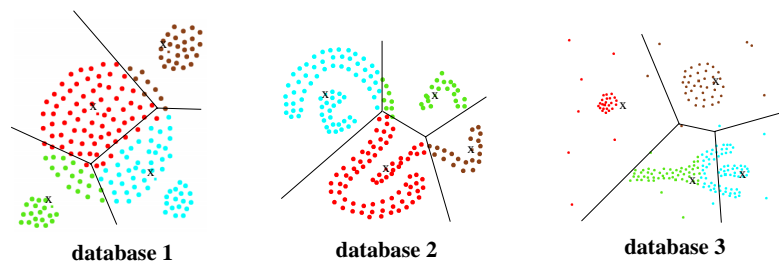


Figure 37: Clusterings discovered by CLARANS

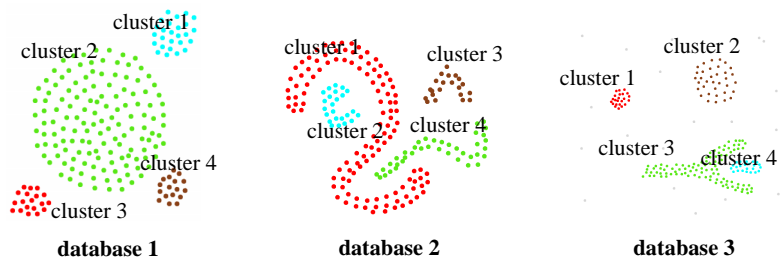


Figure 38: Clusterings discovered by DBSCAN

These examples are rather “hard” for k -medoid (and k -means) type clustering algorithms. They are also intended to illustrate some drawbacks of these types of algorithms when applied to data sets containing clusters of non-convex shape and of largely differing sizes.

To test the efficiency of DBSCAN and CLARANS, we used the SEQUOIA 2000 benchmark data ([SFGM 93]). The SEQUOIA 2000 benchmark database uses real data sets that are typical for Earth Science tasks. There are four types of data in the database: raster data, point data, polygon data and directed graph data. The point data set contains 62,584 Californian landmarks, extracted from the US Geological Survey's Geographic Names Information System, together with their location. The point data set occupies about 2.1 MB. Since the run-time of CLARANS on the whole data set is very high, we have extracted a series of subsets of the SEQUOIA 2000 point data set containing from 2% to 20% representatives of the whole set. The run-time comparison of DBSCAN and CLARANS on these databases is presented in table 2 and depicted in figure 39 (note that in this figure the *log* of the run-times is depicted). The results of our experiments show that the run-time of DBSCAN is almost linear to the number of points. The run-time of CLARANS, however, is close to quadratic to the number of points. Thus, DBSCAN outperforms CLARANS by a factor of between 250 and 1,900 which grows with increasing size of the database.

number of points	DBSCAN	CLARANS
1,252	3	758
2,503	7	3,026
3,910	11	6,845
5,213	16	11,745
6,256	18	18,029
7,820	25	29,826
8,937	28	39,265
10,426	33	60,540
12,512	42	80,638
62,584	233	???

Table 2: Comparison of run-time for DBSCAN and CLARANS (in sec.)

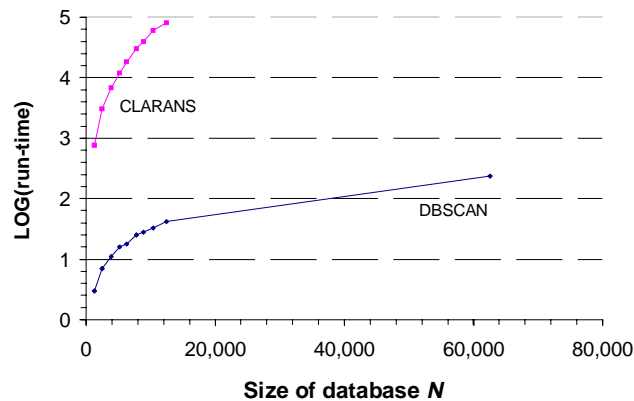


Figure 39: run-time comparison with CLARANS

Since we found it rather difficult to set the parameters of BIRCH appropriately for the SEQUIOA 2000 point data, we used the test data sets DS1, DS2 and DS3 introduced by Zhang et al. ([ZRL 96]) to compare DBSCAN with BIRCH. All three data sets consist of 100,000 2-dimensional points which are randomly distributed within 100 ball-shaped clusters. The data sets differ in the distribution of the cluster centers and their radii. The cluster centers are placed on a grid on DS1, placed along a sine curve in DS2, and placed randomly in DS3. The data sets are depicted in figure 40.

The implementation of BIRCH - using CLARANS in phase 3 - was provided by its authors. The run-time of DBSCAN (see table) was 1.7, 1.9 and 11.6 times the run-time of BIRCH on database 1, 2 and 3 respectively which means that also BIRCH is a very efficient clustering algorithm. Note, however, that in general the same restrictions with respect to cluster shapes and/or their size and location apply to BIRCH as they apply to CLARANS. Furthermore, the clustering features - on which BIRCH is based - can only be defined in a Euclidean vector space implying a limited applicability of BIRCH compared to GDBSCAN (and compared to CLARANS)

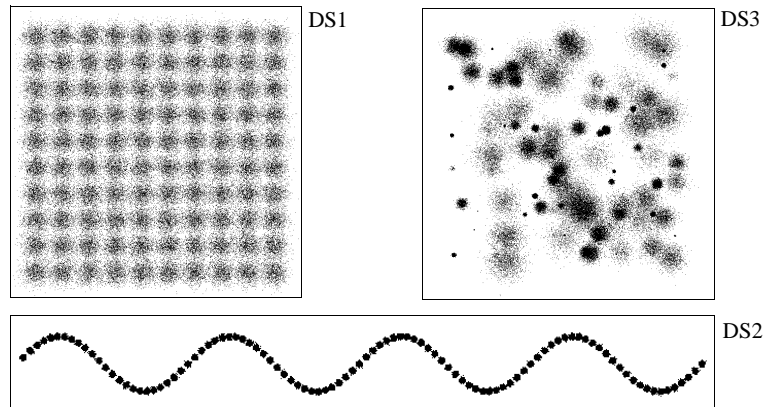


Figure 40: Data sets for the performance comparison with BIRCH

data set	DBSCAN	BIRCH
DS1	82.37	48.48
DS2	79.78	41.01
DS3	520.45	44.21

Table 3: Comparison of run-time for DBSCAN and BIRCH (in sec.)

4.4 Database Support for GDBSCAN

Neighborhood queries using predicates which are based on spatial proximity are not only the most important query type for many spatial data mining algorithms but also the basic query type for exploratory data analysis in spatial databases. To speed-up the processing of these types of queries, typically, spatial index structures combined with a multistep filter-refinement procedure are used in an SDBMS (*cf.* section 4.3.1). There are however several reasons to consider more sophisticated

strategies to improve the performance of spatial data mining algorithms, including the performance of our GDBSCAN algorithm, because spatial databases have the following characteristics with respect to data mining.

1. *Expensive neighborhood queries:*

If the spatial objects are fairly complex or if the dimension of the data space is very high, retrieving the neighbors of some object using the standard technique is still very time consuming. Computing, for instance, the intersection of two complex polygons is an expensive operation that must be performed for almost all candidates retrieved in the first filter step using, for example, an R-tree. Also, retrieving the neighborhood for simple objects like points using for instance an ϵ -range query may require many page accesses for high-dimensional data because the performance of spatial index structures degenerates with increasing dimension d of the data space (see chapter 2).

2. *Large number of neighborhood queries:*

For a typical spatial data mining algorithm, a very large number of neighborhood queries has to be performed to explore the data space. An extreme example is our GDBSCAN algorithm which has to perform a neighborhood query for each object in the database.

3. *Sequences of neighborhood queries for objects contained in the database:*

Many spatial data mining algorithms investigate the neighborhood of objects which are already contained in the database (in contrast to “new” objects, i.e. objects which are located in the same data space as the database objects but which are not stored in the database). Furthermore, the order in which the objects are investigated is often determined by a control structure which is similar to the control structure of our GDBSCAN algorithm. That means, starting from some objects, the algorithm repeatedly retrieves the neighborhood of object which have been located in the neighborhood of objects which have already been processed.

This property applies as well to a typical *manual* data analysis procedure

where short response times for similarity queries are also required. Typically, manual data exploration is started with a similarity query such as a k -nearest neighbor query for an arbitrary object o , i.e. an object which may not be stored in the database. Then, the neighborhood of the starting object o may be further explored by searching for other similar objects. Then, however, we will use the answers of previous queries as starting objects. These objects are all stored in the database. For example, in an image database where the images are represented by high-dimensional feature vectors, we may look for a suitable image that must be similar to a given picture.

In the following subsections we discuss strategies to support the performance of spatial data mining algorithms, including GDBSCAN, in spatial databases where the above mentioned properties hold. First - if they exist - we can use materialized neighborhood indices as proposed in [EKS 97]. These neighborhood indices will speed-up the performance of GDBSCAN significantly for spatial databases containing spatially extended objects or containing high-dimensional data. Second, since a neighborhood query must be performed for a large number of objects of the database, we can develop and utilize techniques to perform multiple neighborhood queries. The gain in performance will be a large factor, in comparison with the use of single independent neighborhood queries.

4.4.1 Neighborhood Indices

In [EKS 97] a general approach to spatial data mining based on a small set of database primitives for spatial data mining is presented (including some new algorithms for spatial characterization and spatial trend detection). The motivation of these database primitives is as follows. First, in spatial databases the explicit location and extension of objects define implicit relations of spatial neighborhood. Second, most data mining algorithms for *spatial* databases will make use of those neighborhood relationships. The reason is that the main difference between data mining in relational DBS and in SDBS is that attributes of the neighbors of some

object of interest may have an influence on the object and therefore have to be considered as well.

We will shortly introduce some of the basic concepts of this approach to see that the techniques used to support the database primitives for spatial data mining can also be applied to our generalized clustering approach.

Neighborhood Graphs and Database Primitives for Spatial Data Mining

The database primitives for spatial data mining are based on the concepts of neighborhood graphs and neighborhood paths which in turn are defined with respect to neighborhood relations between objects. There are different types of neighborhood relations which are important for spatial data mining in general: topological, distance and direction relations. For our purpose it is not necessary to discuss these types of relations here (see [EFKS 98] for more details). We will move directly to the notion of neighborhood graphs and paths.

Definition 10: (*neighborhood graph*)

Let *neighbor* be a neighborhood relation and *DB* be a database of spatial objects.

A *neighborhood graph* $G_{neighbor}^{DB} = (N, E)$ is a graph with nodes $N = DB$ and edges $E \subseteq N \times N$ where an edge $e = (n_1, n_2)$ exists iff *neighbor*(n_1, n_2) holds.

We assume the standard operations from relational algebra like *selection*, *union*, *intersection* and *difference* to be available for sets of objects and sets of neighborhood paths (e.g. the operation *selection*(*set*, *predicate*) returns the set of all elements of a *set* satisfying the predicate *predicate*). Only the following important operations are briefly described:

- *neighbors*: $Graphs \times Objects \times Predicates \rightarrow Sets_of_objects$
The operation *neighbors*(*graph*, *object*, *predicate*) returns the set of all objects connected to *object* in *graph* satisfying the conditions expressed by the predicate *predicate*.

- *paths*: $\text{Sets_of_objects} \rightarrow \text{Sets_of_paths}$
The operation *paths(objects)* creates all paths of length 1 formed by a single element of *objects*. Typically, this operation is used as a type cast for selected starting objects which are investigated by using a data mining algorithm.
- *extensions*: $\text{Graphs} \times \text{Sets_of_paths} \times \text{Integer} \times \text{Predicates} \rightarrow \text{Sets_of_paths}$
The operation *extensions(graph, paths, max, predicate)* returns the set of all paths extending one of the elements of *paths* by at most *max* nodes of the graph. The extended paths must satisfy the predicate *predicate*. The elements of *paths* are not contained in the result implying that an empty result indicates that none of the elements of *paths* could be extended. This operation can be implemented as an iterative application of the *neighbors* operation.

The number of neighborhood paths in a neighborhood graph may become very large. However, for the purpose of KDD, we are mostly interested in paths “leading away” from the start object. We conjecture that a spatial KDD algorithm using a set of paths which are crossing the space in arbitrary ways will not produce useful patterns. The reason is that spatial patterns are most often the effect of some kind of influence of an object on other objects in its neighborhood. Furthermore, this influence typically decreases or increases more or less continuously with an increasing or decreasing distance. To create only relevant paths, the argument *predicate* in the operations *neighbors* and *extensions* can be used to select only a subset of all paths. The definition of *predicate* may use spatial as well as non-spatial attributes of the objects or paths (see [EFKS 98] for the definition of special filter predicates selecting only “starlike” sets of path, i.e. paths “leading away” from a start object.).

Many different spatial data mining algorithms can be expressed in terms of these basic notions, because they basically investigate certain neighborhood paths in an appropriately defined neighborhood graph. This is also true for our generalized clustering algorithm GDBSCAN (see [EFKS 98] for other examples, including spatial characterization and spatial trend detection). For these algorithms, the objects of a spatial database are viewed as the nodes of a neighborhood graph, and the

relevant edges and paths of the graph are created on demand - given the definition of the respective neighborhood relation. Therefore, the efficiency of many spatial data mining algorithms depends heavily on an efficient processing of the *neighbors* operation since the neighbors of many objects have to be investigated in a single run of a data mining algorithm.

Neighborhood Indices

There may be two important characteristics for data mining in certain types of spatial databases such as geographic information systems that can justify the materialization of relevant information about the neighborhood relations. These characteristics may hold for a spatial database, in addition to the properties 1. to 3. stated above (expensive neighborhood queries, large number of neighborhood queries, and sequences of neighborhood queries for objects contained in the database):

4. *Similar neighborhood graphs for many data mining operations:*

Different spatial data mining algorithms may use very similar neighborhood graphs. Thus, very similar *neighbors* operations will be performed again and again. An example is to combine spatial trend detection and spatial characterization, i.e. first, detect objects which are the center for interesting spatial trends and then find a spatial characterization for the regions around these objects (see [EFKS 98] for details). In this case, both algorithms will perform nearly the same *neighbors* operations.

5. *Rare updates:*

Many spatial databases are rather static since there are not many updates on objects such as geographic maps or proteins.

The idea of *neighborhood indices* is to avoid database accesses to the spatial objects themselves by materializing relevant information about the neighborhoods of the objects in the database. This approach is similar to the work of [Rot 91] and [LH 92]. [Rot 91] introduces the concept of *spatial join indices* as a materialization of a spatial join with the goal of speeding-up spatial query processing. This paper,

however, does not deal with the questions of efficient implementation of such indices. [LH 92] extends the concept of spatial join indices by associating each pair of objects with their distance. In its basic form, this index requires $O(n^2)$ space because it needs one entry not only for pairs of neighboring objects but for each pair of objects. Therefore, in [LH 92] a hierarchical version of distance associated join indices is proposed. In general, however, we cannot rely on such hierarchies for the purpose of supporting spatial data mining.

We define a neighborhood index for spatially extended objects including information about distance, direction and topological relations between objects in the following way:

Definition 11: (*neighborhood index*)

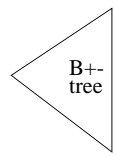
Let DB be a set of spatial objects and let c and $dist$ be real numbers. Let D be a direction predicate and T be a topological predicate. Then the *neighborhood index* for DB with maximum distance c , denoted by N_c^{DB} , is defined as follows:

$$N_c^{DB} = \{(o_1, o_2, dist, D, T) \mid |o_1 - o_2| \leq dist \wedge dist \leq c \wedge D(o_1, o_2) \wedge T(o_1, o_2)\}.$$

A neighborhood index stores information about pairs of objects up to a certain distance c . The distance $dist$ between any two objects in a neighborhood index N_c^{DB} is less than the maximum distance c which means that a neighborhood index does not contain information about *all* pairs of objects.

For storing and retrieving the information contained in a neighborhood index, usual one-dimensional index structures such as B-trees or Hashing can be used. A simple implementation of a neighborhood index using a B^+ -tree on the attribute *Object-ID* is illustrated in figure 41. Implementing neighborhood indices for high-dimensional spatial data by one-dimensional index structures offers a large speed-

up in the processing of neighborhood queries - if the neighborhood index is applicable.



Object-ID	Neighbor	Distance	Direction	Topology
o_1	o_2	2.7	southwest	disjoint
o_1	o_3	0	northwest	overlap
...

Figure 41: Sample Implementation of a Neighborhood Index

A neighborhood index N_c is *applicable* to the neighborhood graph G_r if the maximum distance c of N_c is greater than the maximum possible distance between objects fulfilling the neighborhood relation r , because then all neighbors with respect to r can be found in this neighborhood index. For many neighborhood relations, there is an upper bound for the distance between pairs of objects. Clearly, this is true for all distance based neighborhoods which may combine spatial proximity with conditions on non-spatial attributes (recall section 3.2.2 for examples).

Obviously, if two indices N_{c_1} and N_{c_2} , $c_1 < c_2$, are available and applicable, using N_{c_1} is more efficient because in general it will be smaller than the index N_{c_2} . In figure 42 the algorithm for processing the *neighbors* operation selecting the smallest available index is sketched.

Updates of the database, i.e. insertions, deletions or modifications of spatial objects, require updates of the derived neighborhood indices. These updates on a derived neighborhood can be performed easily because the updates on an object are in general restricted to the neighborhood of this object. This relevant neighborhood can be retrieved by simply using *neighbors* operations.

<p>neighbors (graph G_r, object o, predicate $pred$)</p> <ul style="list-style-type: none"> - Index Selection: Select the smallest available neighborhood index NI applicable to G_r. - Filter Step: If NI exists, use it and retrieve as candidates c the neighbors of o stored in NI. Else, use the ordinary spatial index and retrieve as candidates the set of objects c satisfying $o r c$. - Refinement Step: From the set of candidates, return all objects o' that fulfill $o r o'$ as well as $pred(o')$.
--

Figure 42: Algorithm neighbors using neighborhood indices

Performance Evaluation

Obviously, if available, a neighborhood index can be used for the GDBSCAN algorithm. Typically, neighborhood indices may be available for spatial databases such as geographic information systems. Geographic information systems offer many opportunities for very different kinds of spatial data mining algorithms - all using *neighbors* operations - because in general much information for each object is stored in these databases (objects may have a spatial extension and many non-spatial attributes). Consequently, we used a geographic information system on Bavaria to evaluate the performance of neighborhood indices.

To determine the performance gain for GDBSCAN using neighborhood indices, it is sufficient to measure the speed-up for single *neighbors* operations. Therefore, we compared the performance of single *neighbors* operations with and without a materialized neighborhood index. Although, the expected speed-up may be theoretically obvious, we performed some experiments for the Bavaria database representing Bavarian communities with one spatial attribute ($2-d$ polygon) and 52 non-spatial attributes such as average rent or rate of unemployment. The performance of the *neighbors* operation (*intersects*) using a neighborhood index was compared

with an operation based on a usual multi-step filter-refinement procedure including an R-tree. The neighborhood index was implemented by using a B⁺-tree. Figure 43 depicts the results of these experiments.

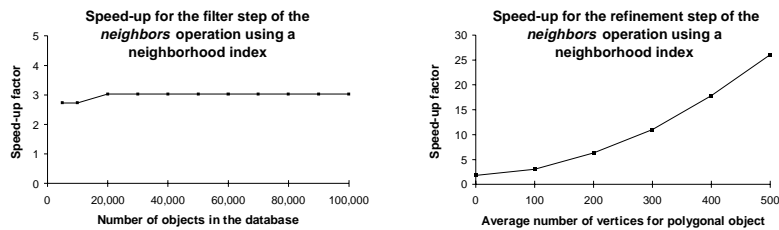


Figure 43: Speed-up for the neighbors-operation using a neighborhood index

Figure 43, on the left, presents the speed-up factor for the filter step in the execution of the neighbors operation, varying the number N of objects in the database. As we can see, this is more or less a constant factor for the 2-dimensional data sets since the performance of one- and two-dimensional range queries in B⁺-trees and R-trees differ in general only by a factor as depicted. This factor, i.e. the performance gain, will increase dramatically with increasing dimension d because the performance of spatial indices degenerates for high-dimensional data (*cf.* section 2.1).

The speed-up factors for the refinement step, varying the average number of vertices, i.e. the complexity of the polygons, is depicted in figure 43, on the right. The speed-up factor for the refinement step obviously increases with increasing complexity of the objects because the evaluation of a neighborhood predicate like *intersects* is much more expensive using the exact object representation than using the information stored in a neighborhood index.

To conclude, a large performance gain for databases containing very complex objects or high-dimensional data can be achieved by using a neighborhood index. However, the cost for creating a neighborhood index, i.e. materializing the neigh-

neighborhoods in a spatial database, are nearly the same as the cost for a single run of GDBSCAN without a neighborhood index (or any other spatial data mining algorithm looking at the neighborhood of all objects). Therefore, creating a neighborhood index will only pay off if it will be used several times by different algorithms.

If no materialized neighborhood index is available, there are other possibilities to improve the performance of our algorithm. One such technique, i.e. multiple neighborhood queries, is presented in the next subsection.

4.4.2 Multiple Neighborhood Queries

The three characteristics of spatial data mining described on page 93 are fulfilled for our GDBSCAN algorithms, i.e. we have a large number of sequences of expensive neighborhood queries. The neighborhood queries are expensive, mostly because for very large databases the processing of a neighborhood query has to be disk-based. Furthermore, with increasing dimension of the data space, an increasingly large part of the file containing the data has to be loaded from disk to answer a single neighborhood query because the performance of index structures degenerates with increasing dimension. Therefore, we introduce a technique called “multiple neighborhood query” to decrease the run-time of GDBSCAN dramatically.

The basic idea for the application of multiple neighborhood queries is rather simple. Assume, there are a many neighborhood queries which must be performed for a specific data mining task. Instead of processing these neighborhood queries separately, we can design an operation which processes several neighborhood queries simultaneously. As we will see, this operation will reduce the overall run-time for all neighborhood queries by a large factor. This is due to the fact that the number of disk accesses is reduced significantly by our technique.

Before we will describe multiple neighborhood queries in detail, we want to point out that this is an important technique not only for GDBSCAN. In fact, the

algorithmic scheme GDBSCAN is an instance of an even more general scheme which covers many algorithms performing spatial data mining tasks. All instances of this so called “*ExploreNeighborhoods*-scheme” can benefit from an operation performing multiple neighborhood queries because this scheme can be easily transformed into an equivalent scheme that uses multiple neighborhood queries.

The *ExploreNeighborhoods*-scheme for spatial data mining which is based on the exploration of neighborhoods is depicted in figure 44. The points in the argument list of some functions indicate additional arguments which may be necessary for different instances of this algorithmic scheme.

```

ExploreNeighborhoods(db, startObjects, NPred, ...)
  contolList := startObjects;
  WHILE ( condition_check(contolList, ...) = TRUE )
    object := contolList.choose(...);
    proc_1(object, ...);
    answers := db.neighborhood(object, NPred);
    proc_2(answers, ...);
    contolList := ( contolList  $\cup$  filter(answers, ...) ) - {object};
  ENDWHILE

```

Figure 44: Algorithmic scheme *ExploreNeighborhoods*

Starting from some objects which are passed to the procedure as a parameter *startObjects*, the algorithm repeatedly retrieves the neighborhood of objects taken from the class *contolList* as long as the function *condition_check* returns TRUE for the *contolList*. In the most simple form, the function checks whether *contolList* is not empty. If the neighborhood of objects should only be investigated up to a certain “depth”, then an additional parameter for the number of steps that have to be performed can be used in the function *condition_check*. The control structure of the main loop works as follows: objects are selected from the *contolList*, one at a time, and a neighborhood query is performed for this object. The procedures *proc_1* and *proc_2* perform some processing on the selected object as well as on the answers

to the neighborhood query that will vary from task to task. Then, the `controlList` is updated. Some or all of the answers which are not yet processed are simply inserted into the class `controlList`. The function `filter(answers, ...)` removes from the set of answers at least those objects which have already been in the `controlList` in previous states of the algorithm, if there any exists. This must be done to guarantee the termination of the algorithm.

It is easy to see that GDBSCAN obviously is an instance of an `ExploreNeighborhoods`-algorithm. Also the manual data exploration in a spatial database described above (point 3 on page 93) as well as spatial characterization and spatial trend detection can be subsumed by our `ExploreNeighborhoods`-scheme. For these tasks, the loop is additionally controlled by the number of steps (i.e. the length of the neighborhood path) and the procedures `proc_1` and `proc_2` perform the actual analysis of the neighborhoods for the purpose of characterization and trend detection (for details of these algorithms see [EFKS 98]). Note that even the materialization of a spatial neighborhood index (cf. section 4.4.1) can be regarded as a special case of the `ExploreNeighborhoods`-scheme. In this case, the argument `startObjects` represents the whole database, `proc_1` is empty, i.e. it performs nothing; `proc_2` simply inserts the neighborhood set into the neighborhood index, and the `filter` function always returns the empty set. Another typical application of an explore neighborhood scheme is the classification of a set of objects simultaneously using a k -nearest neighbor classifier.

Algorithms which follow the `ExploreNeighborhoods`-scheme can be easily reformulated in a way that they use multiple neighborhood queries instead of a single neighborhood query at a time. Figure 45 presents the transformed algorithmic scheme “*ExploreNeighborhoodsMultiple*” using multiple neighborhood queries. As we can see, the reformulation can be done in a purely syntactical way which means that a multiple neighborhood query would be very easy to use if this feature would be available, for instance, as a basic operation provided by a spatial database management system.


```

ExploreNeighborhoodsMultiple(db, startObjects, NPred, ...)
contolList := startObjects;
WHILE ( condition_check(contolList, ...) = TRUE )
  setOfObjects := contolList.choose_multiple(...);
  // setOfObjects = [object1, ..., objectm]
  proc_1(setOfObjects.first(), ...);
  setOfAnswers := db.multiple_neighborhoods(setOfObjects, NPred);
  // setOfAnswers = [answers1, ..., answersm]
  proc_2(setOfAnswers.first(), ...);
  contolList := ( contolList  $\cup$  filter(setOfAnswers.first(), ...) ) - {object};
ENDWHILE

```

Figure 45: Algorithmic scheme *ExploreNeighborhoodsMultiple*

Obviously, the algorithmic scheme *ExploreNeighborhoodsMultiple* performs exactly the same task as the original *ExploreNeighborhoods* scheme. The only differences are that a set of objects is selected from the control-list instead of a single object and a multiple neighborhood query is performed instead of a single neighborhood query. However, in one execution of the main loop, the algorithm will only make use of the first element of the selected objects and the corresponding set of answers.

Our transformed algorithmic scheme may seem “odd” because some kind of buffering for the elements of *setOfAnswers* must be implemented in the query processor of the database - if the run-time should be improved compared to the non-transformed algorithmic scheme. One may have expected a transformed scheme in which all neighborhoods for all the selected objects are processed after the execution of the method *multiple_neighborhoods* and thus no special buffering in the database would be necessary. This approach, however, would have two important disadvantages.

First, there exists no general syntactical transformation for this approach in which the semantics of the non-transformed algorithm is preserved. The reason is

that the argument lists of the procedures `proc_1`, `proc_2` and `filter` are not determined completely in the scheme `ExploreNeighborhoods`. The procedures may, for instance, be dependent on the `controlList` which is possibly changed in each execution of the main loop. This dependency can in general not be modeled by purely syntactical means in a transformed scheme where all selected objects and the corresponding answers are processed in one execution of the main loop. On the other hand, in the transformed scheme that we actually propose, this problem does not exist. The procedures `proc_1`, `proc_2` and `filter` must not be changed and they operate under the same pre-conditions as in the non-transformed scheme.

The second advantage of our approach is that a multiple neighborhood query must only produce a complete answer to the first element in the argument `setOfObjects` instead of complete answers to *all* elements of `setOfObjects`. This allows us to devise implementations of multiple neighborhood queries which compute the neighborhoods of selected objects incrementally. As we will see, this may be more efficient if we consider the overall run-time of the `ExploreNeighborhoodsMultiple` algorithm.

So far, we have argued that it is highly desirable to have efficient techniques for multiple neighborhood queries integrated into a spatial database management system. In the following, we describe two strategies for the implementation of multiple-neighborhood operations which are intended to reduce the number of disk I/O.¹

The first implementation of multiple neighborhood queries is based on the linear scan. This strategy is very simple but most effective in terms of a possible speed-up. Furthermore, the linear scan is applicable to retrieve the *NPred*-neighborhood for arbitrary neighborhood predicates defined on objects having an arbitrary data type. For instance, if only a dissimilarity distance function (which is not a metric)

1. Under special assumptions, further improvements may be possible by reducing the number of main memory operations. For instance, if the neighborhood is distance-based and the distance function is a metric, we can exploit inter-object distances between the query centers and use the triangle inequality to possibly reduce the number of distance computations.

is given for a clustering problem, we must use the linear scan to retrieve the neighborhood of an object because there exists no suitable index structure for this type of application. But even in case of an Euclidean vector space, if the dimension of the space is very high, it may be most efficient to use a kind of optimized linear scan such as the VA-file (*cf.* chapter 2, section 2.1).

Implementation on top of a linear scan (e.g. the VA-file)

Using the linear scan, the method `db.neighborhood(object, NPred)` retrieves the *NPred*-neighborhood of a single object o from a database `db` by simply checking the condition $NPred(o, o')$ for each object $o' \in db$ and returning those objects o' which fulfil this condition as result. That means that each page of the database must be read from disk. Obviously, we can perform a condition check on more than one object while performing a single scan over the database. Therefore, the implementation of the method `db.multiple_neighborhoods(setOfObjects, NPred)` just performs a single scan over the database `db` and checks the condition $NPred(o, o')$ for each object o in `setOfObjects` and each object o' in the database `db`, returning a set of neighborhoods as result. If m is the number of objects contained in `setOfObjects` and m answers can be held in main memory at the same time, then the speed-up factor with respect to disk I/O is exactly equal to m for a multiple neighborhood query compared to m single neighborhood queries.

Implementation on top of a true index structure (e.g. the X-tree)

For true index structures, e.g. an X-tree, there are several possibilities to implement the method `db.multiple_neighborhood(object, NPred)`. Here, we introduce an implementation which is very similar to the technique for the linear scan. In fact, our method will become identical to the method for the linear scan if the performance of the index structure degenerates to the performance of the linear scan.

When answering a single neighborhood query for an object o using an X-tree, a set of data pages which cannot be excluded from the search is determined from the directory of the tree. These pages are then examined and the answers to the query are determined. The amount of pages to be read from disk depends on the size of

the database, the degree of clustering, the dimension of the data space and on the size of the neighborhood (e.g. the distance in case of a range query).

To answer a multiple neighborhood query for a set of objects $O = \{o_1, \dots, o_m\}$, we propose the following procedure. First, we determine the data pages to be read as if answering only a single neighborhood query to determine the neighborhood of o_1 . However, when processing these pages, we do not only collect the answers in the neighborhood of o_1 but also collect answers for the objects o_i ($i=2, \dots, m$) if the pages loaded for o_1 would also be loaded for o_i . After this first step, the query for o_1 is completely finished and the neighborhoods for all the other objects are partially determined. Then, in the next step, we determine the remaining data pages for the object o_2 , i.e. we consider only those data pages relevant for o_2 which have not been processed in the first step. Again, we determine answers for all remaining objects and at least the answer-set for o_2 will be completed. This procedure is repeated until the set O is empty, i.e. the neighborhoods for all objects in O are determined.

This procedure for a multiple neighborhood query may seem to be equivalent to a more simple non-incremental scheme: determine all data pages which have to be read from disk for all objects in O and collect the *complete* answer-sets for all objects o_1, \dots, o_m from these pages in a single pass. The number of data pages which have to be read from disk is actually the same for both methods if we consider only *one* multiple neighborhood query for some objects o_1, \dots, o_m . However, if we consider the overall run-time of `ExploreNeighborhoodsMultiple`, the incremental computation of the neighborhood sets may be more efficient with respect to disk I/O. The reason is that objects which are inserted into the control-list in one execution of the main loop can be additionally selected for a multiple neighborhood query in the next execution of the loop.

Assume that in the first execution of the loop the neighbors p_1, \dots, p_k of o_1 are inserted into the control-list and that these objects are additionally selected at the

beginning of the second execution. Then, the multiple neighborhood query is executed for the set $O = \{o_2, \dots, o_m, p_1, \dots, p_k\}$ which means that now all data pages are considered which have not been processed for object o_1 and, therefore, have to be loaded for object o_2 . It is very likely for an *ExploreNeighborhoods*-algorithm - especially for GDBSCAN - that some of these pages must also be considered for some objects p_i ($i=1, \dots, k$) because the objects contained in an instance of the control-list are usually not very far away from each other. Then, the answers for the objects p_i are (partially) collected from the current data pages determined by the object o_2 . These pages will not be loaded again when p_i becomes the first element of the selected objects. If we had used the non-incremental evaluation of a multiple neighborhood query, to find the neighbors of p_1, \dots, p_k , we would have to load these pages again, resulting in an overall higher number of disk I/O.

Note that, furthermore, our implementation of a multiple neighborhood query, on top of a multi-dimensional index, converges to the method for the linear scan when the page selectivity decreases, e.g. with increasing dimension of the data space. In the worst case, the index has no selectivity at all, which means that no data page can be excluded from a neighborhood-search for a single object. Then, all pages will be read to compute the neighborhood for the first object o_1 in $O = \{o_1, \dots, o_m\}$ and therefore - as for the linear-scan-method - the answers for all objects o_2, \dots, o_m can also be determined completely. This yields the maximum possible speed-up with respect to disk I/O for this case.

Now, it is obvious that in any case our incremental method will load at most as many data pages as the non-incremental method and thus it will never perform worse than the non-incremental alternative.

Performance Evaluation

We performed several experiments to measure the speed-up factors for DBSCAN with respect to disk I/O using multiple neighborhood queries compared to single neighborhood queries. Figure 46 presents the results: the average run-time for de-

termining the neighborhood of an object when using a single neighborhood query compared to multiple neighborhood queries and the corresponding speed-up factors - for both, the linear scan and the X-tree.

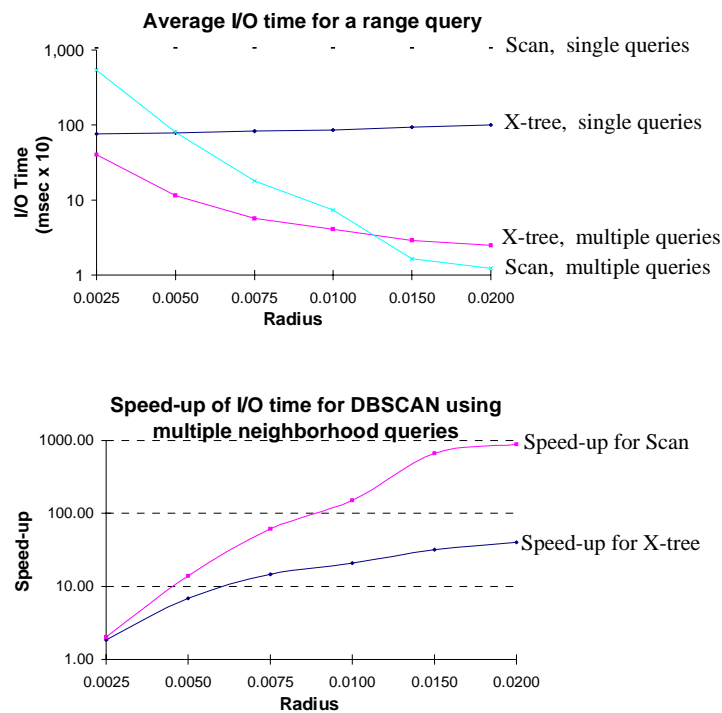


Figure 46: Performance of the multiple-neighbors operation

The average values for the run-time of a neighborhood query depicted in figure 46 were determined by using DBSCAN. For this purpose we clustered several 2-dimensional test databases and selected as many objects as possible from the seed-list of DBSCAN for a multiple neighborhood query. Each database contained 100,000 points. The points were randomly distributed within clusters having a significantly higher density than the noise which was also randomly distributed out-

side the clusters. Different numbers of cluster (100 - 300) and different amounts of noise (10 % - 30%) were used.

We can see in figure 46 that the average I/O time for a multiple range query decreases dramatically with increasing size of the range yielding very large speed-up factors for both the X-tree and the linear scan - up to 50 using the X-tree and up to 850 using the linear scan. These speed-up factors correspond to the average size of the seeds-list of DBSCAN while clustering our test databases.

There are two interesting points concerning the differences between the X-tree and the linear scan. First, the speed-up factors for the linear scan are much larger than the speed-up factors for the X-tree. Second, when using multiple neighborhood queries, the scan outperforms the X-tree with respect to I/O time if the size of the neighborhood exceeds a certain value. In our experiments, this point occurred when the ϵ -neighborhoods contained about 80 objects on the average. These differences are for the most part due to the following two facts: first, in a single execution of `multiple_neighborhoods(setOfObjects, NPred)` the answers for all objects contained in `setOfObjects` are completely determined by the linear scan method while the X-tree - except for the "first" object in `setOfObjects` - generates only partial answers; second, in two dimensional space the X-tree has a very high page selectivity for a single range query, i.e. only very limited number of data pages are considered to answer the range query for the "top object" in `setOfObjects`. However when increasing the dimension d of the data space the page selectivity of the X-tree will degenerate. That means also that for higher dimensions, the curves in figure 46 for the X-tree will converge to the curves of the linear scan.

Note that the speed-up of the I/O time using multiple neighborhood queries increases significantly with the size of the neighborhoods. Therefore, multiple neighborhood queries will yield the largest benefit for the hierarchical version of GDBSCAN (see chapter 7) where we use large neighborhood queries to generate a hierarchy of density-based decompositions.

There are, however, two limits for the speed-up factors which can be obtained by using multiple neighborhood queries in an *ExploreNeighborhoods*-algorithm. The first limit is determined by the average size of the control-list during the execution of an *ExploreNeighborhoodsMultiple*-algorithm. Obviously, the speed-up using a multiple neighborhood query can be at most as large as the number of objects which are processed collectively. Consequently, the maximum possible speed-up factor for an *ExploreNeighborhoodsMultiple*-algorithm is equal to the average number of objects contained in the control-list during the execution of the algorithm. The second limit for the speed-up factors is given by the size of the main memory needed to hold the answer-sets for all neighborhood queries. That means, we may not be able to execute the neighborhood queries for *all* objects contained in the control-list simultaneously if their answers would not fit into the main memory. In this case, only a subset of all possible neighborhood queries will be executed simultaneously.

4.5 Summary

In this chapter, the algorithmic schema GDBSCAN to construct density-based decompositions was introduced. We indicated how GDBSCAN can be implemented independently from the specific predicates for the neighborhood of objects, and the minimum weight for sets of objects. Furthermore, a performance evaluation showed that GDBSCAN is efficient for large databases if the neighborhood queries can be supported by spatial access structures.

We also introduced advanced database techniques such as neighborhood indices and multiple neighborhood queries to speed-up the performance of GDBSCAN by large factors. Especially, we showed that multiple neighborhood queries can be applied to all instances of an even more general algorithmic schema called *ExploreNeighborhoods*. This schema does not only cover our GDBSCAN algorithm as an instance but also a broader class of different spatial data mining algorithms.

Chapter 5

Applications

In this chapter, we present four typical applications of GDBSCAN. In the first application we cluster a spectral space ($5d$ points) created from satellite images in different spectral channels which is a common task in remote sensing image analysis (section 5.1). The second application comes from molecular biology. The points on a protein surface ($3d$ points) are clustered to extract regions with special properties. To find such regions is a subtask for the problem of protein-protein docking (section 5.2). The third application uses astronomical image data ($2d$ points) showing the intensity on the sky at different radio wavelengths. The task of clustering is to detect celestial sources from these images (section 5.3). The last application is the detection of spatial trends in a geographic information system. GDBSCAN is used to cluster $2d$ polygons creating so-called influence regions which are used as input for trend detection (section 5.4).

5.1 Earth Science (*5d* points)

In this application, we use a 5-dimensional feature space obtained from several satellite images of a region on the surface of the earth covering California. These images are taken from the raster data of the SEQUOIA 2000 Storage Benchmark ([SFGM 93]). After some preprocessing, five images containing 1,024,000 intensity values (8 bit pixels) for 5 different spectral channels (1 visible, 2 reflected infrared, and 2 emitted (thermal) infrared) for the same region were combined. Thus, each point on the surface, corresponding to an earth surface area of 1,000 square meters, is represented by a 5-dimensional vector, for example.

```
p1: 222 217 222 155 222
p2: 243 240 243 235 243
...
```

Finding clusters in such feature spaces is a common task in remote sensing digital image analysis (e.g. [Ric 83]) for the creation of thematic maps in geographic information systems. The assumption is that feature vectors for points of the same type of underground on the earth are forming groups in the high-dimensional feature space (see figure 47 illustrating the case of *2d* raster images).

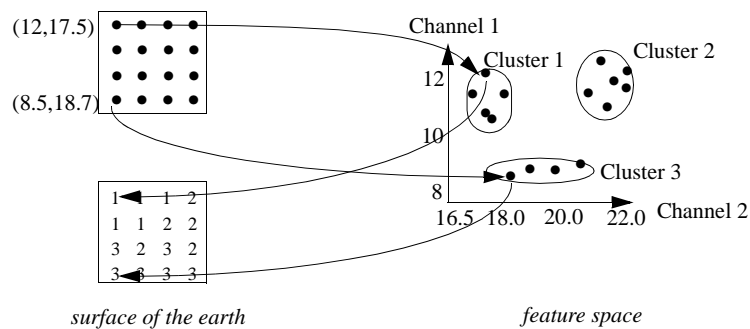


Figure 47: Relation between *2d* image and feature space

Mapping a sample of about 50,000 of the 1,024,000 5-dimensional vectors to 3-dimensional vectors using the FastMap method ([FL 95]) yields a visualization as shown in figure 48. This visualization gives an impression of the distribution of points in the feature space indicating that there are in fact clusters.

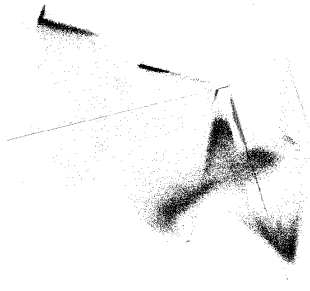


Figure 48: Visualization of the SEQUOIA 2000 raster data

Application 1 has two characteristics which did not exist in the synthetic databases used in the previous chapters. First, the coordinates of points can only be integers with values between 0 and 255 in each dimension. Second, many of the raster points have exactly the same features, i.e. are represented by the same 5-dimensional feature vector. Only about 600,000 of the 1,024,000 feature vectors are different from each other.

We used the specialization DBSCAN for this application. The parameters were determined manually. For reasons of efficiency, we computed the sorted 9 -*dist* graph only for a 1% sample of all points and selected 1.42 as the value for ϵ . These neighborhoods are very small due to the first characteristic of the application, e.g. for about 15% of the points the distance to the 9th nearest neighbor is 0. To take into account the second characteristic of the data, we increased the default value for *MinPts*, i.e. we set $MinPts = 20$. To summarize, the following setting was used:

$$NPred(X, Y) \text{ iff } |X - Y| < 1.42$$

$$MinWeight(N) \text{ iff } |N| \geq 20$$

There are several reasons to apply a post-processing to improve the clustering result of GDBSCAN. First, GDBSCAN only ensures that a cluster contains at least *MinPts* points, but a minimum size of 20 points is too small for this application, especially because many points have the same coordinates. Therefore, we accepted only the clusters containing more than 200 points. This value seems arbitrary but a minimum size can be chosen reasonably after the size of all clusters is known. Second, GDBSCAN produces clusters and noise. But for this application a non-noise class label for each raster point is required. Therefore, we reassigned each noise point and each point of a rejected cluster to the closest of the accepted clusters. We obtained 9 clusters with sizes ranging from 598,863 to 2,016 points.

To visualize the result, each cluster was coded by a different color/grayscale. Then each 2-dimensional point in the image of the surface of the earth was colored according to the identifier of the cluster containing the corresponding 5-dimensional vector. The resulting image is shown in figure 49. A high degree of correspondence between the obtained image and a physical map of California can easily be seen. A detailed discussion of this correspondence is beyond the scope of this work.

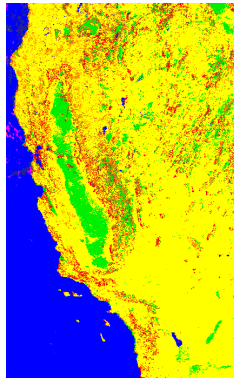


Figure 49: Visualization of the clustering result for the SEQUOIA 2000 raster data

5.2 Molecular Biology (3d points)

Proteins are biomolecules consisting of some hundreds to thousands of atoms. Their mode of operation lies in the interaction with other biomolecules, e.g. proteins, DNA or smaller partner molecules. These interactions are performed by the so-called *docking*, i.e. the process of connecting the partner molecules.

Molecular biologists point out that the geometry of the molecular surfaces at the interaction site plays an important role along with the physicochemical properties of the molecules. A necessary condition for protein-protein docking is the complementarity of the interaction site with respect to surface shape, electrostatic potential, hydrophobicity, etc. We use the crystallographically determined atom coordinates of proteins and protein complexes from the Brookhaven Protein Data Bank ([BKW+ 77], [PDB 94]) and derive for each protein a surface with some 10,000 equally distributed 3d points. For each point on the protein surface, several geometric and physicochemical features are computed. The *solid angle* (*SA*), for example, is a geometric feature describing the degree of convexity or concavity of the surface in the neighborhood of the considered point (see [Con 86]).

A database system for protein-protein docking has to process queries for proteins with complementary surfaces. This search is performed at the level of surface *segments*, defined as a set of neighboring surface points with similar non-spatial attributes, e.g. with similar *SA* values. The segments should have a good correlation with the known docking sites of the proteins, i.e. a docking site on a protein surface should consist of a small number of segments. Therefore, finding a segmentation of protein surfaces is an important subtask for a protein docking database. We applied GDBSCAN for this task.

The parameters *NPred* and *MinWeight* were determined analytically. We used *MinWeight* predicates similar to the one used for the specialization DBSCAN, i.e. comparing a value derived from a neighborhood set to a threshold *MinPts*. The difference is that we did not use “simple” cardinality of the neighborhood set but we

simultaneously performed a selection on the *SA* values. The *SA* values are normalized in the interval $[0, 1]$ such that high *SA* values indicate points on a *convex* surface segment, and low *SA* values indicate points on a *concave* surface segment. To find the convex segments, we used *SA values* between 0.75 and 1.00; for points on a concave surface segment, we used *SA values* between 0.00 and 0.65. The *NPred*-neighborhood is distance based. Since the surface points are equally distributed with a density of 5 points per \AA^2 , we calculated the average 5th-nearest-neighbor distance and used this value of 0.6 for ϵ in the definition of the *NPred*-neighborhood. Consequently, *MinPts* was set to 5 in the definition of the *MinWeight* predicate. To summarize, the following settings were used:

For convex segments:

$NPred(X, Y)$ iff $|X - Y| < 0.6$

$MinWeight(N)$ iff $|\{ p \in N \mid 0.75 \leq SA(p) \leq 1.00 \}| \geq 5$

For concave segments:

$NPred(X, Y)$ iff $|X - Y| < 0.6$

$MinWeight(N)$ iff $|\{ p \in N \mid 0.00 \leq SA(p) \leq 0.65 \}| \geq 5$

Note that if we would use the specialization DBSCAN with “simple” cardinality, only a single cluster containing all points of the protein surface would be found. In applications with equally distributed points, GDBSCAN can only find reasonable clusters if the *MinWeight* predicate is defined appropriate, i.e. the *MinWeight* predicate must “simulate” regions of different density. We searched for clusters covering at least 1% of the surface points of the protein. For example, for the protein 133DA consisting of 5,033 surface points, only clusters with a minimum size of 50 surface points were accepted. For this protein 8 convex and 4 concave clusters (segments) were found by using the above parameter settings. Figure 50 depicts the clustering results of GDBSCAN for this protein. Note that some of the clusters are hidden in the visualization because only one view angle for the protein

is depicted. GDBSCAN discovered the most significant convex and concave surface segments of the protein, which can easily be verified by visual inspection.

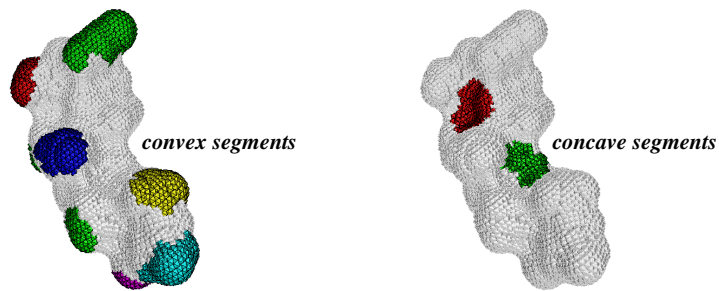


Figure 50: Visualization of the clustering results for protein 133DA

5.3 Astronomy (2d points)¹

Surveys of the sky form an integral part of astronomy. Celestial sources detected in a survey are typically classified by the domain scientists; large surveys will detect many objects and enable statistical studies of the objects in a given classification. Surveys may also reveal exotic or anomalous objects or previously unidentified classes of objects. A typical result of a survey is a 2-dimensional grid of the intensity on the sky (though additional dimensions, e.g. frequency or velocity, polarization, may also be acquired). The measured intensity is typically the sum of the emission from discrete sources, diffuse emission (e.g. from the atmosphere, interplanetary medium or interstellar medium), and noise contributed by the surveying instrument itself. Modern surveys are capable of producing thousands of images of the sky, consuming 10 GB - 1 TB of storage space, and may contain 10^5 to 10^6 or more sources (e.g. [BWH 95], [CCG+ 95]).

1. Special thanks to T. Joseph W. Lazio for making the astronomy data available and for his substantial help in understanding and modeling this application.

Maximizing the yield from a survey requires an accurate and efficient method of detecting sources. The traditional method of separating the discrete sources from the noise and other emissions is to require that the sources exceed a predefined threshold, e.g. 5σ , where σ is an estimate of the *rms* intensity in the image (e.g. [BWH 95]). Recently, alternate methods which utilize the expected statistics of the intensity ([ZCW 94]) or classifier systems ([WFD 95]) have been applied.

An extreme example of a noisy image is shown on the left side of figure 51. The image shows the intensity, as measured by the Very Large Array (information on the VLA is available at <URL:<http://info.aoc.nrao.edu/doc/vla/html/VLAhome.shtml>>), in a direction towards the Galactic center at a radio wavelength of 4,865 MHz. The image is dominated by a celestial source near the center, and the sidelobes which appear as radial spokes and are produced by the optics of the instrument. A second image of the same area at a slightly different wavelength was also given for this application. Because of its similarity to the first image, it is not depicted. The intensity values in the images range from -0.003084 to 0.040023 and from -0.003952 to 0.040509 respectively.

grayscale representation of one image

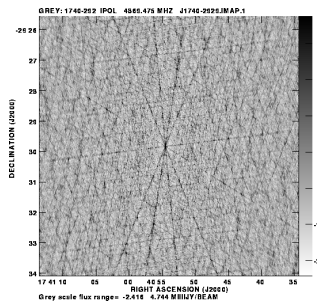


Figure 51: Visualization of the astronomy data

We applied GDBSCAN using the same parameter settings for both images. The neighborhood of a raster point (pixel) is defined as a 3×3 array of points (pixels).

For a region in the image to be of special interest we require an average intensity of 0.005 for each pixel in the region. This requirement is integrated into the definition of our *MinWeight* predicate which compares the total intensity for all 9 pixels in a neighborhood with a threshold equal to 0.045:

$$NPred(X,Y) \text{ iff } |X - Y| < 1.42$$

$$MinWeight(N) \text{ iff } \sum_{p \in N} intensity(p) \geq 0.045$$

The resulting clusterings for both images are given in figure 52. For example, the brightest celestial source can easily be identified as the cluster in the center.



Figure 52: Clustering results for both astronomy images

For the other clusters, it is not so easy to verify that they are in fact celestial sources. But this is a traditional problem with source detection in astronomy. The only way to confirm a weak source is to detect it again in different images, e.g. if it can be detected again by looking at it at slightly different frequencies. A source is required to appear at the same position, maybe with a shift of a pixel or two, at

all frequencies. Therefore, we extracted only the clusters which are present in *both* images. There are 20 of them. The result of this procedure is depicted in figure 53.

cluster present in both images

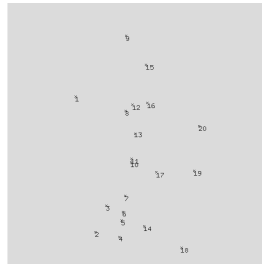


Figure 53: Potential sources in astronomy data

5.4 Geography (*2d polygons*)¹

In the following, we present a simple method - based on GDBSCAN - for detecting “hot spots” on a geographic map possibly containing spatial trends (see [EKSX 97]; for a more comprehensive approach to detect spatial trends see [EFKS 98]). The specialization of GDBSCAN to polygonal data (see chapter 3, section 3.2.2.5) is used to extract regions of interest from a geographic information system on Bavaria.

A *geographic information system* is an information system to manage data representing aspects of the surface of the earth together with relevant facilities such as roads or houses. The Bavaria information system is a database providing spatial and non-spatial information on Bavaria with its administrative units such as communities, its natural facilities such as the mountains and its infrastructure such as

1. Special thanks to Henning Brockfeld (Institute of Economic Geography, University of Munich) for introducing us into the KDD needs of economic geographers.

roads. The database contains the ATKIS 500 data ([Atkis 96]) and the Bavarian part of the statistical data obtained by the German census of 1987. The implementation of the information system follows the SAND (Spatial And Non-spatial Database) architecture ([AS 91]): the spatial extension of all objects (e.g. polygons and lines) is stored and manipulated by using an R*-tree, the non-spatial attributes of the communities (54 different attributes such as the rate of unemployment and the average income) are managed by a relational database management system. The Bavaria database may be used, for example, by economic geographers, to discover different types of knowledge. In the following, we shortly discuss the tasks of spatial trend detection.

A *trend* has been defined as a temporal pattern in some time series data such as network alarms or occurrences of recurrent illnesses ([BC 96]), e.g. “rising interest rates”. We define a *spatial trend* as a pattern of systematic change of one or several non-spatial attributes in 2D or 3D space.

To discover spatial trends of the economic power, an economic geographer may proceed as follows. Some non-spatial attribute such as the rate of unemployment is chosen as an indicator of the economic power. In a first step, areas with a locally minimal rate of unemployment are determined which are called *centers*, e.g. the city of Munich. The theory of central places ([Chr 68]) claims that the attributes of such centers influence the attributes of their neighborhood to a degree which decreases with increasing distance. For example, in general it is easy to commute from some community to a close by center. This will result in a lower rate of unemployment in this community. In a second step, the theoretical trend of the rate of unemployment in the neighborhood of the centers is calculated, e.g.

- when moving away from Munich, the rate of unemployment increases (confidence 86%)

In a third step, deviations from the theoretical trends are discovered, e.g.

- when moving away from Munich in south-west direction, then the rate of unemployment is stable (confidence 97%)

The goal of the fourth step is to explain these deviations. For example, if some community is relatively far away from a center, but is well connected to it by train, then the rate of unemployment in this community is not as high as theoretically expected.

We conjecture that this process of trend detection is relevant not only for economic geography but also for a broader class of applications of geographic information systems, e.g. for environmental studies. The steps are summarized as follows and are illustrated by figure 54:

- 1) *discover centers*, i.e. local extrema of some non-spatial attribute(s).
- 2) *determine theoretical trend as well as observed trend* around the centers.
- 3) *discover deviations* of the observed from the theoretical trend.
- 4) *explain deviations* by other spatial objects in that area and direction.

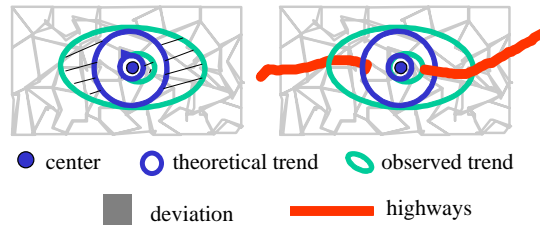


Figure 54: Trend analysis in geographic geography

GDBSCAN is used to extract density-connected sets of neighboring objects having a similar value of non-spatial attribute(s). In order to define the similarity on an attribute, we partition its domain into a number of disjoint classes, e.g. “very high”, “high”, “medium”, “low”, “very low”. A function *attribute-class* maps at-

tribute values to the respective class values, i.e. $attribute-class(X)$ denotes the class of the attribute value X . Values in the same class are considered as similar to each other. The sets with the highest or lowest attribute value(s) are most interesting and are called *influence regions*, i.e. the maximal neighborhood of a center having a similar value in the non-spatial attribute(s) as the center itself. Then, the resulting influence region is compared to the circular region representing the theoretical trend to obtain a possible deviation.

Different methods may be used for this comparison, e.g. difference-based or approximation-based methods. A *difference-based method* calculates the difference of both, the observed influence region and the theoretical circular region, thus returning some region indicating the location of a possible deviation. An *approximation-based method* calculates the optimal approximating ellipsoid of the observed influence region. If the two main axes of the ellipsoid differ in length significantly, then the longer one is returned indicating the direction of a deviation. These methods are illustrated in figure 55.

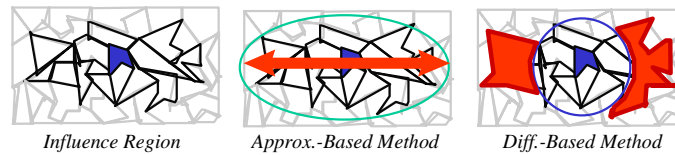


Figure 55: Comparison of theoretical and observed trends

GDBSCAN can be used to extract the influence regions from an SDBS by using the following parameter setting (where we exclude sets of less than 2 objects):

$$NPred(X, Y) \text{ iff } intersect(X, Y) \wedge attribute-class(X) = attribute-class(Y)$$

$$MinWeight(N) \text{ iff } |N| \geq 2$$

Seven centers with respect to a high average income are present in the Bavaria database. Figure 56 depicts the influence regions of these centers in the Bavaria database with respect to high average income detected by GDBSCAN; some of which are discussed in the following:

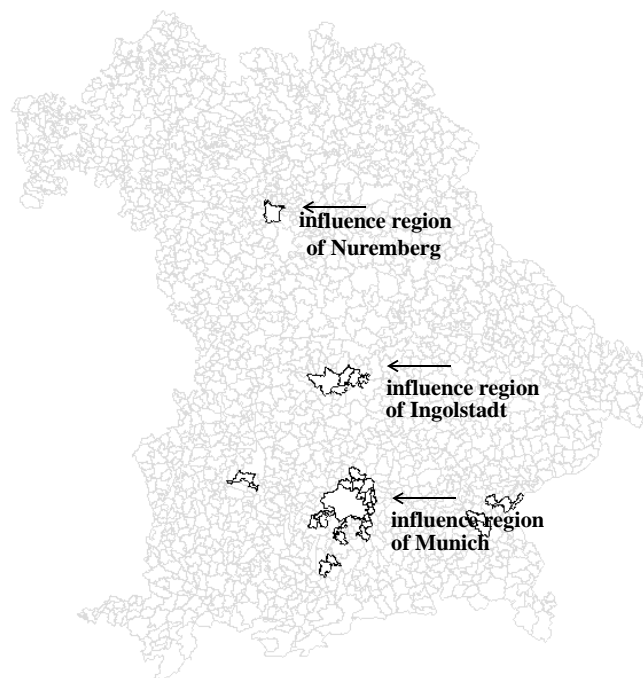


Figure 56: Influence regions with respect to average income extracted from the Bavaria database

The influence region of Nuremberg is circle-shaped showing no significant deviation - in contrast to the influence regions of Nuremberg and Munich.

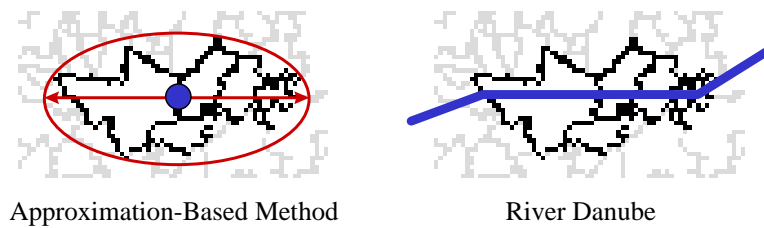


Figure 57: Explanation of the influence region of Ingolstadt

The influence region of Ingolstadt is elongated, indicating a deviation in west-east direction caused by the river Danube traversing Ingolstadt in this direction. Figure 57 shows the approximating ellipsoid and the significantly longer main axis in west-east direction.

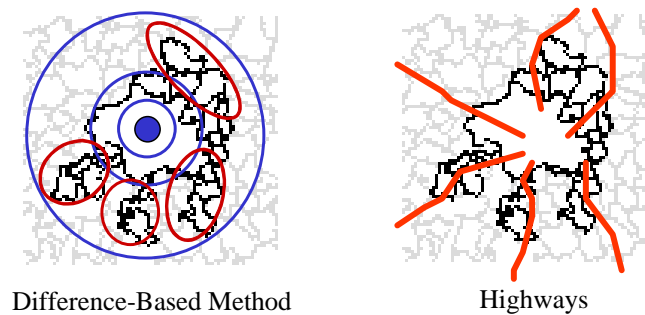


Figure 58: Explanation of the influence region of Munich

The influence region of Munich has four significant deviations from the theoretical region (NE, SW, S and SE). Figure 58 illustrates the difference between the observed influence region and the theoretical circular region. These areas coincide with the highways originating from Munich.

5.5 Summary

In this chapter we presented several applications for different parameter specializations of GDBSCAN. First, we presented an application of DBSCAN to a 5-dimensional spectral space. To determine the clusters in such a spectral space is an important task for the creation of, for example, land-use maps. Second, we extracted concave and convex surface segments on 3-dimensional protein data. In this application we applied a specialization of GDBSCAN that used a selection condition on non-spatial attributes in the definition of the *MinWeight* predicate. In the third application we applied GDBSCAN to 2-dimensional astronomical images to detect celestial sources. In this application GDBSCAN uses the intensity values of the objects/pixels as a weight in the definition of the *MinWeight* predicate. In the last application, we used GDBSCAN to find interesting regions for trend detection in a geographic information system on Bavaria., i.e. a database of 2-dimensional polygons also having several non-spatial attributes. A spatial trend was defined as a pattern of systematic change of one or several non-spatial attributes in $2d$ or $3d$ space. Additionally, we discussed how the discovered knowledge can be useful for economic geographers. The neighborhood predicate *NPred* for objects was defined by using intersection and the similarity of non-spatial attribute values.

Chapter 6

Incremental GDBSCAN

In this chapter we present an incremental version of GDBSCAN (see also [EKS+ 98] for a short presentation with respect to the specialization DBSCAN). After motivating incremental clustering applications (section 6.1), we show that due to the density-based nature of GDBSCAN the insertion or deletion of an object affects the current clustering only in the neighborhood of this object (section 6.2). Thus, efficient algorithms can be given for incremental insertions and deletions to an existing clustering (section 6.3, 6.4, 6.5) which yield the same result as the application of non-incremental GDBSCAN to the whole updated database. For a performance evaluation, we compare the incremental version of our algorithm with the specialization to DBSCAN using a $2d$ spatial database as well as a WWW-log database (section 6.6). The incremental version of DBSCAN yields significant speed-up factors compared to non-incremental DBSCAN, even for large numbers of updates. This demonstrates the efficiency of the proposed algorithm.

6.1 Motivation

Many companies have recognized the strategic importance of the knowledge hidden in their large databases and, therefore, have built data warehouses. A *data warehouse* is a collection of data from multiple sources, integrated into a common repository and extended by summary information (such as aggregate views) for the purpose of analysis [MQM 97]. When speaking of a data warehousing environment, we do not anticipate any special architecture but we address an environment with the following two characteristics:

- Derived information is present for the purpose of analysis.
- The environment is dynamic, i.e. many updates occur.

Typically, a data warehouse is not updated immediately when insertions and deletions on the operational databases occur. Updates are collected and applied to the data warehouse periodically in a batch mode, e.g. each night [MQM 97]. Then, all patterns derived from the warehouse by data mining algorithms have to be updated as well. This update must be efficient enough to be finished when the warehouse has to be available for users again, e.g. the next morning. Due to the very large size of the databases, it is highly desirable to perform these updates incrementally ([FAAM 97], [Huy 97]).

Maintenance of derived information such as views and summary tables has been an active area of research [MQM 97], [Huy 97]. The problem of incrementally updating mined patterns after making changes to the database has just recently started to receive more attention.

For example, in a medical database, one may seek associations between treatments and results. The database is constantly updated and at any given time, the medical researcher is interested in obtaining the current associations. In a database containing news articles, for example, patterns of co-occurrence amongst the top-

ics of articles may be of interest. An economic analyst receives a lot of new articles every day and he would like to find relevant associations based on all current articles. In a WWW access log database [MJHS 96], we may want to find and monitor groups of similar access patterns by clustering the access sequences of different users. These patterns may change in the course of time because each day new log-entries are added to the database and old entries (past a user-supplied expiration date) are deleted. The groups of similar access patterns may correspond to user groups and/or groups of logically connected Web pages.

Up to now, only a few investigations on the problem of incrementally updating mined patterns on changes of the database are available. [CHNW 96] and [FAAM 97] propose efficient methods for incrementally modifying a set of association rules mined from a database.¹ [EW 98] introduce generalization algorithms for incremental summarization in a data warehousing environment.² The task which we consider in this chapter is the incremental clustering.

The clustering of earthquake epicenters stored in an earthquake catalog, for instance, could be done incrementally. Earthquake epicenters occur along seismically active faults, and are measured with some errors, so that in the course of time the observed earthquake epicenters should be clustered along such seismic faults [AF 96]. When clustering this type of database incrementally, there are no deletions but only insertions of new earthquake epicenters over time.

1. The task of mining association rules has been introduced by [AS 94]. An *association rule* is a rule $I_1 \Rightarrow I_2$ where I_1 and I_2 are disjoint subsets of a set of items I . For a given database DB of transactions (i.e. each record contains a set of items bought by some customer in one transaction), all association rules should be discovered having a support of at least *minsupport* and a confidence of at least *minconfidence* in DB . The subsets of I that have at least *minsupport* in DB are called *frequent sets*.

2. Summarization, e.g. by generalization, is another important task of data mining. Attribute-oriented generalization (see [HCC 93]) of a relation is the process of replacing the attribute values by a more general value - one attribute at a time, until the number of tuples of the relation becomes less than a specified threshold. The more general value is taken from a concept hierarchy which is typically available for many attributes in a data warehouse.

Another important application for incremental clustering may be the clustering of WWW access log databases. These databases typically contain access log entries following the *Common Log Format* specified as part of the HTTP protocol [Luo 95]. Such log entries mainly contain information about the machine, the user, the access date and time, the Web page accessed, and the access method. Figure 59 depicts some sample log entries from the WWW access log database of the Institute for Computer Science at the University of Munich.

```
romblon.informatik.uni-muenchen.de lopa - [04/Mar/1997:01:44:50 +0100] "GET /~lopa/ HTTP/1.0" 200 1364
romblon.informatik.uni-muenchen.de lopa - [04/Mar/1997:01:45:11 +0100] "GET /~lopa/x/ HTTP/1.0" 200 712
fixer.sega.co.jp unknown - [04/Mar/1997:01:58:49 +0100] "GET /dbs/porada.html HTTP/1.0" 200 1229
scooter.pa-x.dec.com unknown - [04/Mar/1997:02:08:23 +0100] "GET /dbs/kriegel_e.html HTTP/1.0" 200 1241
```

Figure 59: Sample WWW access log entries

In this application, the goal of clustering is to discover groups or clusters of similar access patterns. Access patterns can be described by the sequence of Web pages accessed by a user in a single session.

A *session* is constructed from the basic access log database by restructuring the log entries: all log entries with identical IP address and user-id within a given maximum time gap are grouped into a session, and redundant entries, i.e. entries with file name suffixes such as “gif”, “jpeg”, and “jpg” are removed [MJHS 96]. A session has the following general structure:

$$\text{session} ::= \langle \text{ip_address}, \text{user_id}, [\text{url}_1, \dots, \text{url}_k] \rangle$$

Then, the task of discovering groups of similar access pattern can be handled by clustering the user sessions of a Web log database. A WWW provider may use the discovered clusters of sessions as follows:

- The users associated with the sessions of a cluster form some kind of user group. The topics which are contained in the Web pages accessed by a user group can be interpreted as a user profile. This kind of information may, for example, be used to develop marketing strategies.
- The URLs of the sessions contained in a cluster represent topics which are “connected by user interests”. This information could be used to reorganize the local Web structure. For example, URLs contained in a cluster can be made easily accessible from each other via appropriate new links.

The access patterns may change in the course of time. New entries are inserted into the WWW access log database each day and they will expire after a certain time, i.e. they are deleted from the database after a specified period, for instance, after six months. Assuming a constant daily number of WWW accesses, the numbers of insertions and deletions in this type of application are the same for each day.

GDBSCAN is applied to static databases. In a data warehouse, however, the databases may have frequent updates and thus may be rather dynamic. After insertions and deletions to the database, the clustering discovered by GDBSCAN has to be updated. Incremental clustering means to consider only the old clusters and the objects inserted or deleted during the day instead of applying the clustering algorithm to the (very large) updated database.

Due to the density-based nature of GDBSCAN, the insertion or deletion of an object affects the current clustering only in the neighborhood of this object. In section 6.2 we examine which part of an existing clustering is affected by an update of the database. Then, we present algorithms for incremental updates of a clustering after insertions (section 6.3) and deletions (section 6.4). It is an important advantage of our approach that, based on the formal notion of clusters, it can be easily seen that the incremental algorithm yields the same result as the non-incremental GDBSCAN algorithm. In section 6.6, we demonstrate the high efficiency of incremental clustering on a spatial database as well as on a WWW access log database.

6.2 Affected Objects

Let D be a database, $NPred$ be a binary neighborhood predicate and let $MinWeight$ be a predicate for the minimum weight of sets of objects. Recall that we denote by $N_{NPred}(o)$ the $NPred$ -neighborhood of an object o . Additionally, we introduce the notation $N_{2NPred}(o)$ for an enhanced neighborhood of o , i.e. the set of all objects which are in the neighborhood of objects o' which in turn are in the neighborhood of object o :

Definition 12: (enhanced neighborhood $N_{2NPred}(o)$)

$$N_{2NPred}(o) = \{q \in D \mid \exists o' \in N_{NPred}(o) \wedge q \in N_{NPred}(o')\}$$

We want to show that changes to a clustering of a database D are restricted to a neighborhood of an inserted or deleted object p . Objects contained in $N_{NPred}(p)$ can change their core object property, i.e. core objects may become non-core objects and vice versa. This is due to the fact that for all objects $p' \in N_{NPred}(p)$ also the property $p \in N_{NPred}(p')$ holds. Therefore, insertion or deletion of object p may affect $MinWeight(N_{NPred}(p'))$. Objects in $N_{2NPred}(p) \setminus N_{NPred}(p)$ keep their core object property, but non-core objects may change their connection status, i.e. border objects may become noise objects or vice versa, because their $NPred$ -neighborhood may contain objects with a changed core object property. For all objects outside of $N_{2NPred}(p)$ it holds that neither these objects themselves nor objects in their $NPred$ -neighborhood change their core object property. Therefore, the connection status of these objects is unchanged.

Figure 60 illustrates the possible changes with respect to core object property and connection status in a sample database of two-dimensional objects using parameter values for DBSCAN as depicted. In this figure, the object $a \in N_{NPred}(p)$ is a core object only if object p is contained in the database. Otherwise, object a is a

border object. As a consequence, object $c \in N_{NPred}(a)$ will be density-reachable, depending on the presence of object p . The core object property of object b is not affected by the insertion or deletion of p .

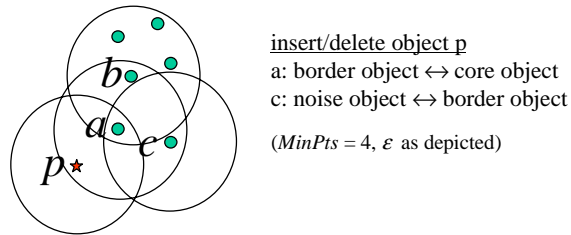


Figure 60: Changing core object property and connection status

All objects outside $N_{2NPred}(p)$ keep their core object property and their connection status. They may, however, change their *cluster membership* because new density-connections may be established or removed - in case of an insertion resp. deletion.

After the insertion of some object p , non-core objects (border objects or noise objects) in $N_{NPred}(p)$ may become core objects implying that new density-connections may be established, i.e. chains $p_1, \dots, p_n, p_1 = r, p_n = s$ with p_{i+1} directly density-reachable from p_i for two objects r and s may arise which were not density-reachable from each other before the insertion. Then, one of the p_i for $i < n$ must be contained in $N_{NPred}(p)$.

When deleting some object p , core objects in $N_{NPred}(p)$ may become non-core objects implying that density-connections may be removed, i.e. there may no longer be a chain $p_1, \dots, p_n, p_1 = r, p_n = s$ with p_{i+1} directly density-reachable from p_i for two objects r and s which were density-reachable from each other before the deletion. Again, one of the p_i for $i < n$ must be contained in $N_{NPred}(p)$.

Figure 61 illustrates our discussion using a sample database of $2d$ objects and an object p to be inserted or to be deleted. $N_{NPred}(o) = \{o' \in D \mid |o - o'| \leq \epsilon\}$, ϵ is as depicted, and $MinWeight(N)$ iff $|N| \geq 4$. The objects a and b are then density-connected without using one of the elements of $N_{NPred}(p)$. Therefore, a and b belong to the same cluster independently from p . On the other hand, the objects d and e in $D \setminus N_{NPred}(p)$ are only density-connected via c in $N_{NPred}(p)$ if the object p is contained in the database, so that the cluster membership of d and e is affected by p .

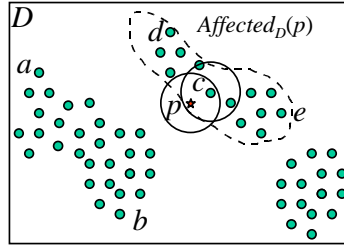


Figure 61: Affected objects in a sample database

In general, on an insertion or deletion of an object p , the set of *affected objects*, i.e. objects which may potentially change their cluster membership after the update, is the set of objects in $N_{NPred}(p)$ plus all objects density-reachable from one of these objects in $D \cup \{p\}$. The cluster membership of all other objects not in the set of *affected objects* will not change. This is the intuition of the following definition and lemma. In particular, the lemma states that a cluster or density-connected set in the database is independent of an insertion or deletion of an object p if a core object of the density-connected set is outside the set $Affected_D(p)$. Note that a density-connected set is uniquely determined by any of its core objects. Therefore, by definition of $Affected_D(p)$ it follows that if one core object of a density-connected set is outside (inside) $Affected_D(p)$, then *all* core objects of the density-connected set are outside (inside) the set $Affected_D(p)$.

Definition 13: (*affected objects*)

Let D be a database of objects and p be some object (either in or not in D). We define the set of objects in D affected by the insertion or deletion of p as

$$\text{Affected}_D(p) := N_{NPred}(p) \cup \{q \mid \exists o \in N_{NPred}(p) \wedge q >_{D \cup \{p\}} o\}.$$

Lemma 7: Let D be a set of objects and p be some object.

Then $\forall o \in D: o \notin \text{Affected}_D(p) \Rightarrow \{q \mid q >_{D \setminus \{p\}} o\} = \{q \mid q >_{D \cup \{p\}} o\}$.

Proof (sketch): 1) \subseteq : because $D \setminus \{p\} \subseteq D \cup \{p\}$. 2) \supseteq : if $q \in \{q \mid q >_{D \cup \{p\}} o\}$, then there is some chain q_1, \dots, q_n , $q_1 = o$, $q_n = q$, $q_{i+1} \in N_{NPred}(q_i)$ and q_i is a core object in $D \cup \{p\}$ for all $i < n$ and, for all i , it holds that $q_i >_{D \cup \{p\}} o$. Because q_i is a core object for all $i < n$ and the density-reachability is symmetric for core objects, it also holds that $o >_{D \cup \{p\}} q_i$. If there exists an $i < n$ such that $q_i \in N_{NPred}(p)$, then $q_i >_{D \cup \{p\}} p$ implying also $o >_{D \cup \{p\}} p$ due to the transitivity of density-reachability. By definition of the set $\text{Affected}_D(p)$ it now follows that $o \in \text{Affected}_D(p)$ in contrast to the assumption. Thus, $q_i \notin N_{NPred}(p)$ for all $i < n$ implying that all the objects q_i , $i < n$, are core objects independent of p and also $q_n \neq p$ because otherwise $q_{n-1} \in N_{NPred}(p)$. Thus, the chain q_1, \dots, q_n exists also in the set $D \setminus \{p\}$ and then $q \in \{q \mid q >_{D \setminus \{p\}} o\}$. \square

Due to lemma 7, after inserting or deleting an object p , it is sufficient to reapply GDBSCAN to the set $\text{Affected}_D(p)$ in order to update the clustering. For that purpose, however, it is not necessary to retrieve the set first and *then* apply the clustering algorithm. We simply have to start a restricted version of GDBSCAN which does not loop over the whole database to start expanding a cluster, but only over certain “seed”-objects which are all located in the neighborhood of p . These “seed”-objects are core objects *after* the update operation which are located in the $NPred$ -neighborhood of a core object in $D \cup \{p\}$ which in turn is located in $N_{NPred}(p)$. This is the content of the next lemma.

Lemma 8: Let D be a set of objects. Additionally, let $D^* := D \cup \{p\}$ after insertion of an object p or $D^* = D \setminus \{p\}$ after deletion of p and let c be a core object in D^* .

$C = \{o \mid o \succ_{D^*} c\}$ is a cluster in D^* and $C \subseteq \text{Affected}_D(p) \Leftrightarrow \exists q, q' : q \in N_{NPred}(q'), q' \in N_{NPred}(p), c \succ_{D^*} q, q$ is core object in D^* and q' is core object in $D \cup \{p\}$.

Proof (sketch): If $D^* := D \cup \{p\}$ or $c \in N_{NPred}(p)$, the lemma is obvious by definition of $\text{Affected}_D(p)$. Therefore, we consider only the case $D^* := D \setminus \{p\}$ and $c \notin N_{NPred}(p)$.

“ \Rightarrow ”: $C \subseteq \text{Affected}_D(p)$ and $C \neq \emptyset$. Then, there exists $o \in N_{NPred}(p)$ and $c \succ_{D \cup \{p\}} o$, i.e. there is a chain of directly density-reachable objects from o to c . Now, because $c \notin N_{NPred}(p)$ we can construct a chain $o = o_1, \dots, o_n = c$, $o_{i+1} \in N_{NPred}(o_i)$ with the property that there is $j \leq n$ such that for all $k, j \leq k \leq n$, $o_k \notin N_{NPred}(p)$ and for all $k, 1 \leq k < j$, $o_k \in N_{NPred}(p)$. Then $q = o_j \in N_{NPred}(o_{j-1})$, $q' = o_{j-1} \in N_{NPred}(p)$, $c \succ_{D^*} o_j$, o_j is a core object in D^* and o_{j-1} is a core object in $D \cup \{p\}$.

“ \Leftarrow ”: obviously, $C = \{o \mid o \succ_{D^*} c\}$ is a density-connected set (see lemma 4). By assumption, c is density-reachable from a core object q in D^* and q is density-reachable from an object $q' \in N_{NPred}(p)$ in $D \cup \{p\}$. Then also c and hence all objects in C are density-reachable from q' in $D \cup \{p\}$. Thus, $C \subseteq \text{Affected}_D(p)$. \square

Due to lemma 8, the general strategy for updating a clustering would be to start the GDBSCAN algorithm only with core objects that are in the $NPred$ -neighborhood of a (previous) core object in $N_{NPred}(p)$. However, it is not necessary to rediscover density-connections which are known from the previous clustering and which are not changed by the update operation. For that purpose, we only need to look at core objects in the $NPred$ -neighborhood of those objects that change their core object property as a result of the update. In case of an insertion, these objects may be connected after the insertion. In case of a deletion, density-connections between them may be lost. In general, this information can be determined by using

very few region queries. The remaining information needed to adjust the clustering can be derived from the cluster membership before the update. Definition 14 introduces the formal notions which are necessary to describe this approach. Remember: objects with a changed core object property are all located in $N_{NPred}(p)$.

Definition 14: (*seed objects for the update*)

Let D be a set of objects and p be an object to be inserted or deleted. Then, we define the following notions:

$$UpdSeed_{Ins} = \{q \mid q \text{ is a core object in } D \cup \{p\}, \exists q': q' \text{ is core object in } D \cup \{p\} \text{ but not in } D \text{ and } q \in N_{NPred}(q')\}$$

$$UpdSeed_{Del} = \{q \mid q \text{ is a core object in } D \setminus \{p\}, \exists q': q' \text{ is core object in } D \text{ but not in } D \setminus \{p\} \text{ and } q \in N_{NPred}(q')\}$$

We call the objects $q \in UpdSeed$ “seed objects for the update”.

6.3 Insertions

When inserting a new object p , new density-connections may be established but none are removed. In this case, it is sufficient to restrict the application of the clustering procedure to the set $UpdSeed_{Ins}$. If we have to change the cluster membership for an object from C to D , we perform the same change of the cluster membership for all other objects in C . Changing the cluster membership of these objects does not involve the application of the clustering algorithm but can be handled by simply storing the information which clusters have been merged.

When inserting an object p into the database D , we can distinguish the following cases:

- (1) (*Noise*)
 $UpdSeed_{Ins}$ is empty, i.e. there are no “new” core objects after insertion of p . Then, p is a noise object and nothing else is changed.
- (2) (*Creation*)
 $UpdSeed_{Ins}$ contains only core objects which did not belong to a cluster before the insertion of p , i.e. they were noise objects or equal to p , and a new cluster containing these noise objects as well as p is created.
- (3) (*Absorption*)
 $UpdSeed_{Ins}$ contains core objects which were members of exactly one cluster C before the insertion. The object p and possibly some noise objects are absorbed into cluster C .
- (4) (*Merge*)
 $UpdSeed_{Ins}$ contains core objects which were members of several clusters before the insertion. All these clusters, the object p and possibly some noise objects are merged into one cluster.

Figure 62 illustrates the most simple forms of the different cases when inserting an object p into a sample database of $2d$ points using parameters $NPred(o, o')$ iff $|o - o'| \leq \epsilon$ (ϵ as depicted) and $MinWeight(N)$ iff $|N| \geq 3$.

In case one, there are no other objects in the ϵ -neighborhood of the inserted object p . Therefore, p will be assigned to noise and nothing else is changed. In case two, objects p and c are the objects with a changed core object property. But, since no other core objects are contained in their neighborhood the set $UpdSeed_{Ins}$ contains only these two objects. The points p and c are “new” core objects and therefore a new cluster is created, containing p and c as well as a , b , and e which were noise objects before the insertion, but which are now density-reachable from p or from c . In case three, after the insertion of p , object d changed its core object property and object a is a core object contained in the neighborhood of d . The object p

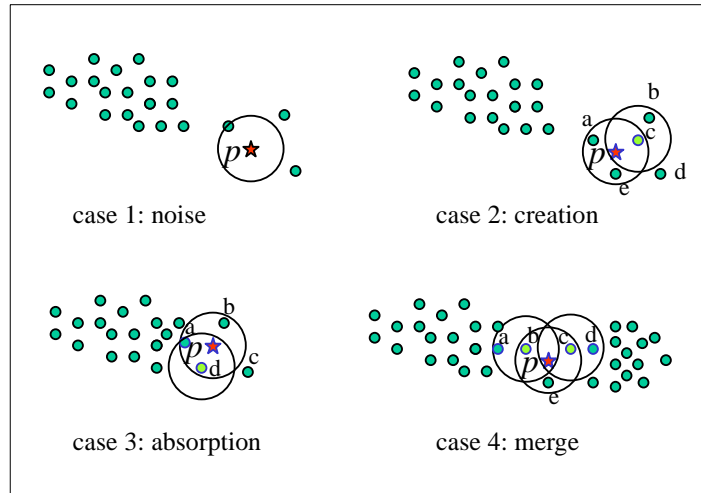


Figure 62: The different cases of the insertion algorithm

is also an object with changed core object property. Hence, the set $UpdSeed_{ms}$ consists of the objects p , a and d . In this case, the object p and the previous noise object b are absorbed into the existing cluster. In case four, the inserted point p is a core object. Points b and c have a changed core object property, and points a and d are core objects located in their neighborhood. These two objects are now density-connected via point p and consequently the two former clusters are merged. Also the point e is included into this new cluster. The point e was a noise object before the insertion, but is now directly density-reachable from p .

Figure 63 presents a more complicated example of merging clusters when inserting an object p . In this example the value for ϵ is as depicted and the threshold value for the minimum weight (using cardinality) is equal to 6. Then, the inserted point p is not a core object, but o_1 , o_2 , o_3 and o_4 are core objects after the update. The previous clustering can be adapted by analyzing only the ϵ -neighborhood of these objects: cluster A is merged with cluster B and C because o_1 and o_4 as well

as o_2 and o_3 are mutual directly density-reachable implying the merge of B and C. The changing of cluster membership for objects in case of merging clusters can be done very efficiently by simply storing the information about the clusters that have been merged. Note that, using cardinality, this kind of “transitive” merging can only occur if the threshold value is larger than 5, because otherwise p would be a core object and then all objects in $N_{\epsilon}(p)$ would already be density-reachable from p .

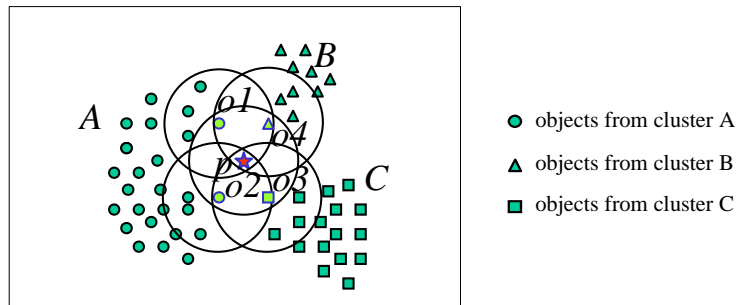


Figure 63: “Transitive” merging of clusters A, B, C by the insertion algorithm

6.4 Deletions

As opposed to an insertion, when deleting an object p , density-connections may be removed, but no new connections are established. The difficult case for deletion occurs when the cluster C of p is no longer density-connected via (previous) core objects in $N_{NPred}(p)$ after deleting p . In this case, we do not know in general how many objects we have to check before it can be determined whether C has to be split or not. In most cases, however, this set of objects is very small because the split of a cluster is not very frequent and in general a non-split situation will be detected in a small neighborhood of the deleted object p . An actual split is obviously the most expensive operation of incremental clustering.

When deleting an object p from the database D , we can distinguish the following cases:

- (1) (*Removal*)
 $UpdSeed_{Del}$ is empty, i.e. there are no core objects in the neighborhood of objects that may have lost their core object property after the deletion of p . Then p is deleted from D and eventually other objects in $N_{NPred}(p)$ change from a former cluster C to noise. If this happens, the cluster C is completely removed because then C cannot have core objects outside of $N_{NPred}(p)$.
- (2) (*simple Reduction*)
 All objects in $UpdSeed_{Del}$ are *directly* density-reachable from each other. Then p is deleted from D and some objects in $N_{NPred}(p)$ may become noise.
- (3) (*potential Split*)
 The objects in $UpdSeed_{Del}$ are *not* directly density-reachable from each other. These objects belonged to exactly one cluster C before the deletion of p . Now we have to check whether or not these objects are density-connected by other objects in the former cluster C . Depending on the existence of such density-connections, we can distinguish a *split* and a *non-split* situation. Note that these situations may occur simultaneously.

Figure 64 illustrates the different cases when deleting p from a sample database of $2d$ points using parameters $NPred(o, o')$ iff $|o - o'| \leq \epsilon$ (ϵ as depicted) and $MinWeight(N)$ iff $|N| \geq 3$.

Case one is inverse to the example for a creation of a new cluster (see figure 62). The point p is deleted and as a consequence the point c loses its core object property. There are no further core objects in the neighborhood of p and c . Therefore the remaining points a and b will become noise. Analogously, the second case is inverse to an absorption. When deleting p , also point d changes its core object property. In this example, the points b and c are assigned to noise because they are no

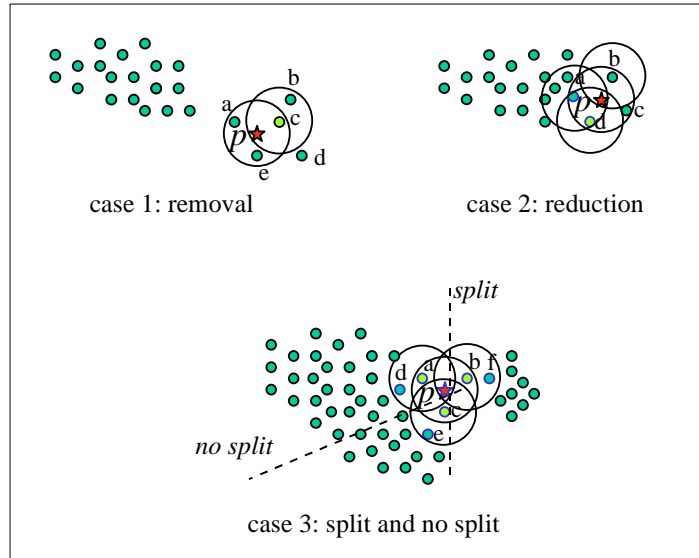


Figure 64: The different cases of the deletion algorithm

longer density-connected to any core object of the cluster. Case three is the most difficult situation. If p is deleted, the points $a, b,$ and c lose their core object property and the points $d, e,$ and f are core objects located in their neighborhood. That means, that the set $UpdSeed_{Del}$ contains the three objects, $d, e,$ and f . However, these three objects are not directly density-connected. Therefore, we must try to find a density-connection between them. In the example, we can easily see that points d and e are still density-connected to each other, but f is not density-connected to any of these two points. The cluster has to be split into two parts. One containing the object f , the other containing the objects d and e . This is done by a procedure similar to GDBSCAN (see section 4.1). However, the main loop corresponding to GDBSCAN has only to consider the three points $d, e,$ and f , i.e. a function corresponding to `ExpandCluster` constructs density-connected sets, starting only from one of these objects.

The implementation of both, the incremental insertion and the incremental deletion is discussed in greater detail in the next section.

6.5 Implementation

Although, we have distinguished different cases for insertions and deletions, the implementation of all these cases can be roughly described by the following single sequence of steps:

1. Detect objects with a changed core object property.
2. Collect the core objects in the $NPred$ -neighborhood of those objects detected in step 1.
3. (Try to) connect the core objects collected in step 2.

In step one and two simply the set $UpdSeed_{ins}$ resp. $UpdSeed_{del}$ is constructed. Trying to connect the different core objects from $UpdSeed$ is more or less simple in case of an insertion, and may fail in case of a deletion if a cluster is actually split.

For an efficient implementation of incremental insertions and incremental deletions, we have to keep the number of $NPred$ -neighborhood queries needed for the update of the clustering as small as possible.

In general, step one requires one $NPred$ -neighborhood query for the inserted or deleted object p plus an additional $NPred$ -neighborhood query for each object q contained in this neighborhood of p to determine which of those objects q have changed their core object property. Then, step two requires additional $NPred$ -neighborhood queries for all those objects that actually changed their core object property to collect the relevant core objects for the set $UpdSeed$. However, when

using incrementally evaluable *MinWeight* predicates which are most frequently used for clustering, the sets $UpdSeed_{Ins}$ and $UpdSeed_{Del}$ can be computed very fast.

Incrementally evaluable *MinWeight* predicates (cf. definition 3) compare the *weight* of a set of objects N to a threshold, i.e. $MinWeight(N)$ iff $weight(N) \geq T$. Furthermore, the weight of the set N can be evaluated incrementally, i.e.

$$weight(N) = \sum_{o \in N} weight(\{o\}).$$

If using an incrementally evaluable *MinWeight* predicate, we store for each object in the database the weight of its *NPred*-neighborhood and the number of objects contained in this neighborhood, when initially clustering the database. Then, we only have to perform a single *NPred*-neighborhood query for the object p to be inserted or deleted to detect all objects q' with a changed core object property. Such objects can be determined by simply analyzing the new weights for the objects in the neighborhood of p , because an object $q \in N_{NPred}(p)$ has a changed core object property if

- in case of inserting p :
 $weight_{stored}(N_{NPred}(q)) \leq T$ and $weight_{stored}(N_{NPred}(q)) + weight(\{p\}) \geq T$
- in case of deleting p :
 $weight_{stored}(N_{NPred}(q)) \geq T$ and $weight_{stored}(N_{NPred}(q)) - weight(\{p\}) \leq T$.

In the second step, we have to determine all core objects $o \in N_{NPred}(q')$ for those objects q' satisfying the above condition (if there are any). Since after step one the *NPred*-neighborhood of p is still in main memory, we check this set for neighbors of q' first and perform an additional *NPred*-neighborhood query only if we know that there are more objects in the neighborhood of q' than already contained in $N_{NPred}(p)$. After step two, we have to update the stored sum of weights and the stored number of objects for the neighborhood of the retrieved objects.

The above strategy is a major performance improvement for incrementally evaluable *MinWeight* predicates because objects with a changed core object property after an update (different from the inserted or deleted object p) are not very frequent (see section 6.6). Since this fact can already be detected in the *NPred*-neighborhood of p , a lot of *NPred*-neighborhood queries can be saved in step one and two.

Step three, i.e. trying to connect the core objects in the set *UpdSeed*, requires a different number of *NPred*-neighborhood queries depending on the situation. In fact, when inserting an object p into the database, no additional *NPred*-neighborhood query is necessary. All new density-connections can be detected in the neighborhoods which have already been retrieved in step one and two.

To achieve that no further accesses to objects of the database are necessary to change cluster membership - even in the case of merging clusters - we introduce equivalence classes of cluster-identifiers. Each equivalence class of cluster-identifiers represents the identifiers for a single density-connected set. A merge situation is then characterized by a set $UpdSeed_{ms}$ containing objects having cluster-identifiers from different equivalence classes. In the beginning, each equivalence class contains exactly one cluster-identifier corresponding to the density-connected sets in the initial clustering. Then, if a merge situation occurs, i.e. if we find core objects with cluster-identifiers from different equivalence classes, for example A and B , we simply unite these classes, and thus get a new equivalence class C replacing the classes A and B .

Unlike an incremental insertion, which is a very efficient operation in any case, an incremental deletion requires additional *NPred*-neighborhood queries to be performed in step three if a “potential split” occurs. If a “potential split” occurs, then the clustering procedure must also consider objects outside the set $UpdSeed_{Del}$. However, it can stop in case of a non-split situation as soon as the objects from the set $UpdSeed_{Del}$ are density-connected to each other.

The procedure to detect density-connections between the objects in $UpdSeed_{Del}$ is implemented by a function which is similar to `ExpandCluster` in the algorithm GDBSCAN (see figure 32). To reduce the number of $NPred$ -neighborhood queries in case of a “potential split”, however, we perform a kind of “breadth first search”. The main difference is that the candidates for further expansion of a current density-connected set are explicitly managed in a queue. Furthermore, the expansion starts in parallel from each object contained in $UpdSeed_{Del}$. This is more efficient than for instance a depth-first search, due to the following reasons:

- In a non-split situation, which is more frequent than a split, we stop as soon as all members of $UpdSeed_{Del}$ are found to be density-connected to each other. The breadth-first search implies that the shortest density-connections, i.e. consisting of a minimum number of objects and thus requiring the minimum number of region queries, are detected first.
- A split situation is in general the more expensive case because the parts of the cluster to be split actually have to be discovered. The algorithm stops when all but the last part have been visited. Usually, a cluster is split only into two parts and one of them is relatively small. Using breadth-first search, we can save many $NPred$ -neighborhood queries on the average because then we only have to visit the smaller part of the cluster and a small percentage of the larger one.

The procedure for handling a potential split uses a new cluster-identifier for each part that is expanded. Because we do not want to undo this labeling of objects in case of a non-split situation, we simply insert the new cluster-identifier(s) into the existing equivalence class representing the cluster under consideration. On the other hand, if a part is actually separated from the current cluster, a new equivalence class is created, containing only the new cluster-identifier.

Obviously, from time to time, we have to reorganize the cluster-identifiers for the whole database. This must be done after the occurrence of many split and merge situations in order to keep the computational overhead for managing the

equivalence classes of cluster-identifiers small. Such a reorganization, however, requires only a single scan over the database, and the split as well as the merge situations are the most rare cases when inserting or deleting an object more or less randomly.

6.6 Performance Evaluation

In this section, we evaluate the efficiency of IncrementalGDBSCAN versus GDBSCAN. As we will see, surprisingly few *NPred*-neighborhood queries have to be performed on the average if we can use all the above features of our implementation, especially for incrementally evaluable *MinWeight* predicates. For this purpose, the specialization to DBSCAN, i.e. a distance based neighborhood and cardinality for the weight of sets of objects, is used. This yields an incrementally evaluable *MinWeight* predicate.

We present an experimental evaluation using a $2d$ spatial database as well as a WWW access log database. For this purpose, we implemented both algorithms in C++ based on implementations of the R*-tree (for the $2D$ spatial database) and the M-tree (for the WWW log database) respectively. Furthermore, an analytical comparison of both algorithms is presented and the speed-up factors are derived for typical parameter values depending on the database size and the number of updates.

For the first set of experiments, we used a synthetic database of 1,000,000 $2d$ points with $k = 40$ clusters of similar sizes. 21.7% of all points are noise, uniformly distributed outside of the clusters, and all other points are uniformly distributed inside the clusters with a significantly higher density than the noise. In this database, the goal of clustering is to discover groups of neighboring objects. A typical real

world application for this type of database is clustering earthquake epicenters stored in an earthquake catalog (*cf.* section 6.1).

In this type of application, there are only insertions. The Euclidean distance was used as distance function and an R*-tree as an index structure. The radius ϵ for the neighborhood of objects was set to 4.48 and the threshold value *MinPts* for the minimum weight was set to 30. Note that the *MinPts* value had to be rather large to avoid 'single link effects' (see the discussion in section 3.2.2.2). This is due to the high percentage of noise. We performed experiments on several other synthetic $2d$ databases with n varying from 100,000 to 1,000,000, k varying from 7 to 40 and with the noise percentage varying from 10% up to 20%. Since we always obtained similar results, we restrict the discussion to the above database.

For the second set of experiments, we used a WWW access log database of the Institute for Computer Science at the University of Munich. This database contains 1,400,000 entries. All log entries with identical IP address and user-id within a time gap of one hour were grouped into a *session* and redundant entries, i.e. entries with file name suffixes such as "gif", "jpeg", and "jpg" were removed. This preprocessing yielded about 69,000 sessions.

Entries are deleted from the WWW access log database after six months. Assuming a constant daily number of WWW accesses, there are 50% of insertions and 50% of deletions in this type of application. Note that this is the largest value for incremental deletions to be expected in real-world applications, because a higher value would mean that the database size converges to zero in the course of time.

We used the following distance function for pairs of URL lists s_1 and s_2 from the WWW sessions:

$$dist(s_1, s_2) = \frac{|s_1 \cup s_2| - |s_1 \cap s_2|}{|s_1 \cup s_2|}$$

The domain of $dist$ is the interval $[0 . . 1]$, $dist(s,s) = 0$ and $dist$ is symmetric and fulfills the triangle inequality¹, i.e. $dist$ is a metric function. Therefore, we can use an M-tree to index the database and to support the performance of ϵ -range queries. In our application, the radius ϵ for the neighborhood of objects was set to 0.4 and the threshold value $MinPts$ for the minimum weight was set to 2. Thus, the clustering corresponds to a “single link level” (cf. section 3.2.2.1).

The function $dist$ is very simple because it does not take into account any kind of ordering in the sequence of Web pages accessed by a user in a single session. For practical applications, it may be worthwhile to develop other distance functions which may for instance use the hierarchy of the directories to define the degree of similarity between two URL lists.

In the following, we compare the performance of IncrementalDBSCAN versus DBSCAN. Typically, the number of page accesses is used as a cost measure for database algorithms because the I/O time heavily dominates CPU time. In both algorithms, region queries are the only operations requiring page accesses. Since the

1. To prove the condition $\frac{|a \cup b| - |a \cap b|}{|a \cup b|} + \frac{|b \cup c| - |b \cap c|}{|b \cup c|} \geq \frac{|a \cup c| - |a \cap c|}{|a \cup c|}$, we first show that it holds if $b = a \cup c$ for any a and c : $\frac{|a \cup b| - |a \cap b|}{|a \cup b|} + \frac{|b \cup c| - |b \cap c|}{|b \cup c|} = \frac{|a \cup c| - |a|}{|a \cup c|} + \frac{|a \cup c| - |c|}{|a \cup c|} = \frac{2|a \cup c| - |a| + |c|}{|a \cup c|} = \frac{2|a \cup c| - (|a \cup c| + |a \cap c|)}{|a \cup c|} = \frac{|a \cup c| - |a \cap c|}{|a \cup c|} \geq \frac{|a \cup c| - |a \cap c|}{|a \cup c|}$. We now

show that for any other set b the left-hand side of the inequality will only be larger. For this purpose, we rewrite the inequality to $1 - \frac{|a \cap b|}{|a \cup b|} + 1 - \frac{|b \cap c|}{|b \cup c|} \geq \frac{|a \cup c| - |a \cap c|}{|a \cup c|}$ and show that the sum $\frac{|a \cap b|}{|a \cup b|} + \frac{|b \cap c|}{|b \cup c|}$ will be decreased for any set b which is not equal to $a \cup c$: Assume a and c are given. Then define $b' = a \cup c$. Now any set b can be constructed by inserting into b' all objects from b which are not already contained in b' (yielding a new set b''), and then subtracting all objects from b'' which do not belong to b (yielding the set b). In the first step, $|a \cap b'|$ and $|b' \cap c|$ do not change; $|a \cup b'|$ and $|b' \cup c|$ can only become larger. Consequently, the sum will decrease. The objects subtracted from b'' in the second step must now be contained either in a or in c . But then again, $|a \cap b''|$ and $|b'' \cap c|$ can only become smaller; $|a \cup b''|$ and $|b'' \cup c|$ do not change because $b'' \supseteq a \cup c$ holds after the insertions. Therefore, the sum will further decrease. \square

number of page accesses of a single region query is the same for DBSCAN and for IncrementalDBSCAN, we only have to compare the number of region queries. Thus, we use the number of region queries as the cost measure for our comparison. Note that we are not interested in the absolute performance of the two algorithms but only in their relative performance, i.e. in the speed-up factor as defined below. To validate this approach, we performed a set of experiments on our test databases and found that the experimental speed-up factor always was slightly larger than the analytically derived speed-up factor (experimental value about 1.6 times the expected value in all experiments).

DBSCAN performs exactly one region query for each of the n objects of the database (see algorithm in figure 32), i.e. the cost of DBSCAN for clustering n objects denoted by $Cost_{DBSCAN}(n)$ is

$$Cost_{DBSCAN}(n) = n$$

The number of region queries performed by IncrementalDBSCAN depends on the application and, therefore, it must be determined experimentally. In general, a deletion affects more objects than an insertion. Thus, we introduce two parameters r_{ins} and r_{del} denoting the average number of region queries for an incremental insertion resp. deletion. Let f_{ins} and f_{del} denote the percentage of insertions resp. deletions in the number of all incremental updates. Then, the cost of the incremental version of DBSCAN for performing m incremental updates denoted by $Cost_{IncrementalDBSCAN}(m)$ is as follows:

$$Cost_{IncrementalDBSCAN}(m) = m \times (f_{ins} \times r_{ins} + f_{del} \times r_{del})$$

Table 4 lists the parameters of our performance evaluation and the values obtained for the 2d spatial database as well as for the WWW-log database. To determine the average values (r_{ins} and r_{del}), the whole databases were incrementally in-

serted and deleted, although f_{del} (percentage of deletions) is equal to zero for the 2-dimensional spatial database.

Parameter	Meaning	Value for 2d spatial database	Value for WWW-log database
n	number of database objects	1,000,000	69,000
m	number of (incremental) updates	varying	varying
r_{ins}	average number of region queries for an incremental insertion	1.58	1.1
r_{del}	average number of region queries for an incremental deletion	6.9	6.6
f_{del}	relative frequency of deletions in the number of all updates	0	0.5
f_{ins}	relative frequency of insertions in the number of all updates ($1-f_{del}$)	1.0	0.5

Table 4: Parameters of the performance evaluation

Now, we can calculate the speed-up factor of IncrementalDBSCAN versus DBSCAN. We define the *speed-up factor* as the ratio of the cost of DBSCAN (applied to the database after all insertions and deletions) and the cost of m calls of IncrementalDBSCAN (once for each of the insertions resp. deletions), i.e.:

$$\begin{aligned} \text{SpeedupFactor} &= \frac{\text{Cost}_{\text{DBSCAN}}(n + f_{ins} \times m - f_{del} \times m)}{\text{Cost}_{\text{IncrementalDBSCAN}}(m)} \\ &= \frac{(n + f_{ins} \times m - f_{del} \times m)}{m \times (f_{ins} \times r_{ins} + f_{del} \times r_{del})} \end{aligned}$$

Figure 65 and figure 66 depict the speed-up factors depending on the size n of the database for several values of updates m . For relatively small numbers of daily updates, e.g. $m = 1,000$ and $n = 1,000,000$, we obtain speed-up factors of 633 for the 2d spatial database and 260 for the WWW-log database. Even for rather large

numbers of daily updates, e.g. $m = 25,000$ and $n = 1,000,000$, IncrementalDBSCAN yields speed-up factors of 26 for the $2d$ spatial database and 10 for the WWW-log database.

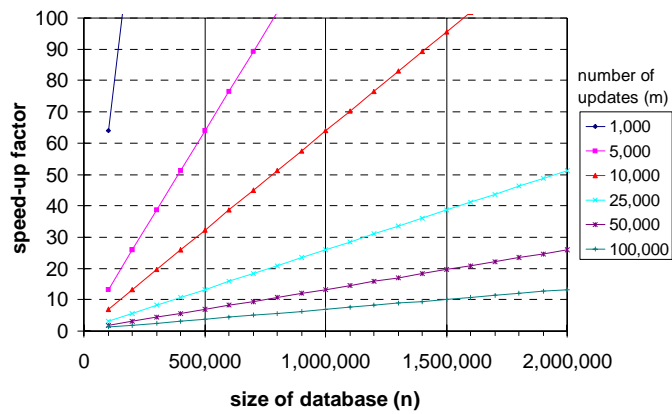


Figure 65: Speed-up factors for $2d$ spatial databases

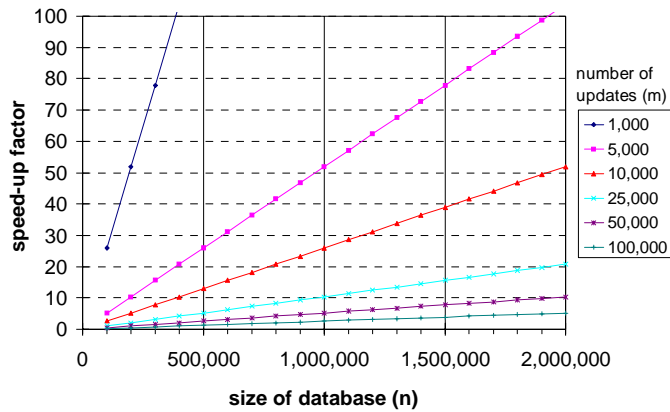


Figure 66: Speed-up factors for the Web-log database

When setting the speed-up factor to 1.0, we obtain the number of updates (denoted by *MaxUpdates*) up to which the multiple application of the incremental version of DBSCAN for each update is more efficient than the single application of DBSCAN to the whole updated database.

Figure 67 depicts the values of *MaxUpdates* depending on n for f_{del} values up to 0.5 which is the maximum value to be expected in most applications. This figure was derived by setting r_{ins} to 1.34 and r_{del} to 6.75. These values are computed as the average over all test databases - 2d and Web-log. Note that - in contrast to the significant differences of other characteristics of the two applications - the differences of the values for r_{ins} and r_{del} are rather small, indicating that the average values are a realistic choice for many applications. The *MaxUpdates* values obtained are much larger than the actual numbers of daily updates in most real databases. For databases without deletions (that is, $f_{del} = 0$), *MaxUpdates* is approximately $3 * n$, i.e. the cost for $3 * n$ updates on a database of n objects using IncrementalDBSCAN is the same as the cost of DBSCAN on the updated database containing $4 * n$ objects. Even in the worst case of $f_{del} = 0.5$, *MaxUpdates* is approximately $0.25 * n$. These results clearly emphasize the relevance of incremental clustering.

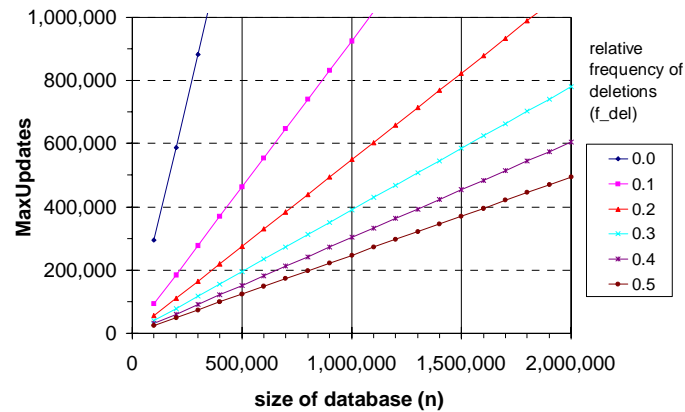


Figure 67: *MaxUpdates* for different relative frequencies of deletions

6.7 Summary

Data warehouses provide a lot of opportunities for performing data mining tasks such as classification and clustering. Typically, updates are collected and applied to the data warehouse periodically in a batch mode, e.g. during the night. Then, all patterns derived from the warehouse by some data mining algorithm have to be updated as well.

In this chapter, we introduced the first incremental clustering algorithm - based on GDBSCAN - for mining in a data warehousing environment. Due to the density-based nature of GDBSCAN, the insertion or deletion of an object affects the current clustering only in a small neighborhood of this object. Thus, efficient algorithms have been presented for incremental insertions and deletions to a clustering, yielding the same result as the application of GDBSCAN to the whole updated database.

A performance evaluation of IncrementalDBSCAN versus DBSCAN using a spatial database as well as a WWW-log database was presented, demonstrating the efficiency of the proposed algorithm (at least for incrementally evaluable *Min-Weight* predicates). For relatively small numbers of daily updates, e.g. 1,000 updates in a database of 1,000,000 objects, IncrementalDBSCAN yields speed-up factors of several hundred. Even for rather large numbers of daily updates, e.g. 25,000 updates in a database of 1,000,000 objects, we obtain speed-up factors of more than 10.

Chapter 7

Hierarchical GDBSCAN

In this chapter, we will introduce the notion of a *hierarchical* density-based decomposition or hierarchical clustering. A hierarchical clustering, formally described as a *nested* density-based decomposition, is simply a hierarchy of “flat” density-based decompositions (section 7.1). We present two different versions of a modified GDBSCAN algorithm to compute nested density-based decompositions. The first is a general version to construct all clustering levels of a nested density-based decomposition with respect to a given sequence of parameters (section 7.2). The second algorithm is a more specialized version for distance-based neighborhood predicates (section 7.1). The advantage of the second algorithm is that we do not produce clustering levels explicitly. We just create an *order* of the database with respect to a maximum distance ϵ , and store very few additional information. This cluster-order can be used for the construction of arbitrary clustering levels with respect to distance values less than ϵ or as a “stand-alone” tool for analyzing the clustering structure of data sets having an arbitrary dimension.

7.1 Nested Density-Based Decompositions

7.1.1 Motivation

So far, we have only considered “flat” density-based decompositions or clusterings. That means that we have partitioned the database into a set of density-connected sets and a set containing noise objects. There are, however, data mining applications where hierarchical clustering information about the data is more useful than a simple partitioning. This is especially the case if an application has one of the following properties:

- The clustering structure of a dataset is best represented by a hierarchical structure, for instance, a dendrogram as produced by hierarchical clustering algorithms (*cf.* chapter 2, section 2.2.1). That means, hierarchical layers of clusters are a “natural” representation of the data set, and therefore are an important property of the data which we may want to detect by a data mining algorithm. Several clustering levels may be considered as being correct. Then, a clustering level can be selected for further investigation depending on the granularity of the analysis.
- Hierarchical clustering information about the data allows us to select the “best” clustering level after the clustering process. That means that a hierarchical clustering procedure is less sensitive to a correct parameter determination. Different layers of clusterings correspond to different parameter settings, and thus the “correct” or “best” clustering parameters can be determined after the clustering process. The additional information which is available for a hierarchical clustering structure allows us also to use other criteria for selecting the appropriate clustering than the density parameters, for instance, the number and the sizes of the clusters.

- It may not be appropriate to use a single parameter setting for the whole data set. To describe the clusters present in different regions of the data space correctly, it may be necessary to select clusters from different levels in the hierarchy. That means, we can choose different parameters for different regions of the data space.

Figure 68 and figure 69 illustrate two different data sets where hierarchical clustering information is more instructive than a simple flat density-based decomposition.

Figure 68 gives an example of a two-dimensional data set with an inherently homogeneous hierarchical clustering structure. One can easily see that there are reasonable clusters present at distinct levels which are defined by different density-parameters. Clusters at different density levels are exemplified on the right side of the figure.

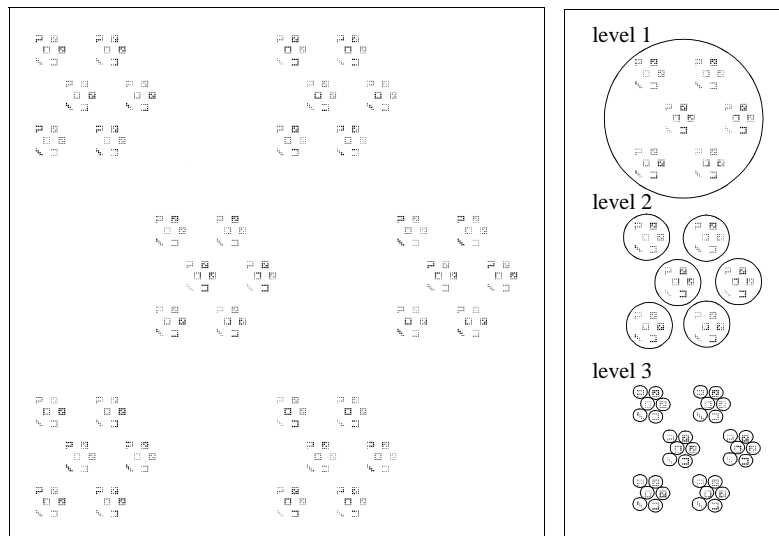


Figure 68: Example of a hierarchical clustering structure

Figure 69 gives an example where different density parameters can be used for different regions of the data space to describe the clustering structure. In this example, it is not possible to detect the clusters A , B , C_1 , C_2 , and C_3 simultaneously using the same density parameters. A flat density-based decomposition could only consist of the clusters A , B , and C , or C_1 , C_2 , and C_3 . In the second case, the objects from A and B would be noise.

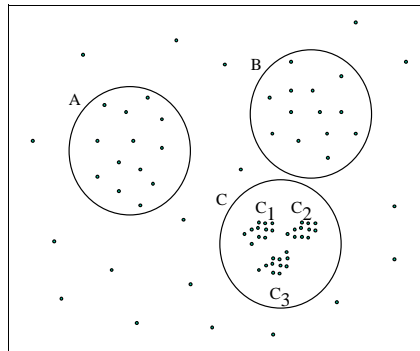


Figure 69: Example of clusters with respect to different density parameters

These properties of the above examples could be easily detected in an appropriate hierarchical representation of the clustering structures, i.e. a *hierarchical density-based decomposition* of the database. A hierarchical density-based decomposition or hierarchical clustering can be described as a *nested* density-based decomposition which is formally introduced in the next section.

7.1.2 Definitions and Properties

A *nested* density-based decomposition is a hierarchy of “*flat*” density-based decompositions as defined in chapter 3, definition 8. However, to construct a hierarchy of density-based decompositions, further assumptions with respect to the den-

sity parameters are required, i.e. we need a sequence of density parameters specifying density-connected sets of increasing density.

For this purpose, we need a list of neighborhood predicates $[NPred_1, \dots, NPred_n]$ which must be ordered in the sense that for all $p \in D$ the condition $N_{NPred_{i+1}}(p) \subseteq N_{NPred_i}(p)$ holds. Furthermore, we have to assume a *monotonous MinWeight* predicate, i.e. if $N_1 \subseteq N_2$ and $MinWeight(N_1)$ then also $MinWeight(N_2)$ (cf. definition 3, chapter 3). Then, if we combine the list of neighborhood predicates with a monotonous *MinWeight* predicate, we obtain a sequence of density parameters $[(MinWeight, NPred_1), \dots, (MinWeight, NPred_n)]$ specifying increasing density values for density-connected sets. For these density parameters, we can prove that density-connected sets with respect to a “higher” density are completely contained in density-connected sets with respect to a “lower” density. This is the content of the following lemma.

Lemma 9: Let D be a database of objects, let *MinWeight* be a monotonous predicate for the minimum weight of sets of objects, and let $NPred_1$ and $NPred_2$ be two neighborhood predicates such that for all $p \in D$: $N_{NPred_2}(p) \subseteq N_{NPred_1}(p)$. If DCS_1 is a density-connected set with respect to $NPred_1$ and *MinWeight*, DCS_2 is a density-connected set with respect to $NPred_2$ and *MinWeight*, and p is a core object in DCS_1 and DCS_2 . Then $DCS_2 \subseteq DCS_1$

Proof: Let $o \in DCS_2$. Then, by lemma 5, o is density-reachable from p with respect to $NPred_2$ and *MinWeight* in D . By definition, there is a chain of objects p_1, \dots, p_n , $p_1=p, p_n=o$ such that for all $i=1, \dots, n-1$: p_{i+1} is directly density-reachable from p_i with respect to $NPred_2$ and *MinWeight* in D . That means that for all $i=1, \dots, n-1$: $p_{i+1} \in N_{NPred_2}(p_i)$ and $MinWeight(N_{NPred_2}(p_i))$ holds. Since, by assumption, $N_{NPred_2}(q) \subseteq N_{NPred_1}(q)$ holds for all $q \in D$, and the *MinWeight* predicate is

monotonous, it follows that $p_{i+1} \in N_{NPred_1}(p_i)$ and $MinWeight(N_{NPred_1}(p_i))$ is also satisfied for all $i=1, \dots, n-1$. That means, o is also density-reachable from p with respect to $NPred_1$ and $MinWeight$ in D . But then, again by lemma 5, it holds that $o \in DCS_1$. \square

Figure 70 illustrates the content of lemma 9 using our DBSCAN specialization and two-dimensional point objects. The neighborhood predicates are defined by using two different distances ε_1 and ε_2 as depicted, and the threshold value $MinPts$ is set to 3. The ε -neighborhoods satisfy the condition that for all $p \in D$: $N_{\varepsilon_2}(p) \subseteq N_{\varepsilon_1}(p)$, and the $MinWeight$ predicate, i.e. comparing the cardinality of a neighborhood to a threshold, is obviously monotonous. In this example, we can easily recognize the set inclusion of density-connected sets satisfying the preconditions of lemma 9, i.e. C_1 and C_2 are density-connected sets with respect to ε_2 and C is a density-connected set with respect to ε_1 completely containing the sets C_1 and C_2 .

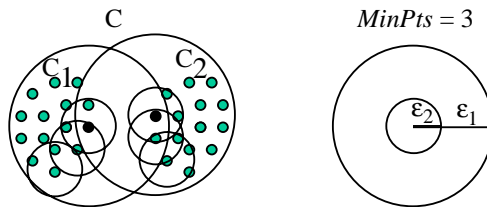


Figure 70: Illustration of lemma 9

We can now define a *nested* density-based decomposition recursively as a tree consisting of “simple” density-based decompositions. The previous lemma guarantees that this tree actually represents the intended *hierarchical* partitioning of a

data set which is induced by the set-inclusion of density-connected sets satisfying the preconditions of lemma 9.

Definition 15: (*nested density-based decomposition*)

Let D be a database of objects, let $MinWeight$ be a monotonous predicate for the minimum weight of sets of objects, and let $[NPred_1, \dots, NPred_n]$, $n \geq 1$, be a sequence of neighborhood predicates such that for all $p \in D$ and for all

$$1 \leq i < n-1: N_{NPred_{i+1}}(p) \subseteq N_{NPred_i}(p).$$

A *nested density-based decomposition* of a database D with respect to $NPred_1, \dots, NPred_n$ and $MinWeight$ denoted as

$$NDBD(D, MinWeight, [NPred_1, \dots, NPred_n])$$

is a rooted tree defined by the following conditions:

The empty tree is defined to be a nested density-based decomposition.

$NDBD(D, MinWeight, [NPred_1, \dots, NPred_n])$ is equal to the empty tree if $[NPred_1, \dots, NPred_n]$ is the empty sequence. Otherwise, the root of the tree is a density-based decomposition $DBD(D, MinWeight, NPred_1) = \{S_1, \dots, S_k; N\}$ of D with respect to $NPred_1$ and $MinWeight$, and the subtrees of this root are the nested density-based decompositions

$$NDBD(S_1, MinWeight, [NPred_2, \dots, NPred_n]),$$

...

$$NDBD(S_k, MinWeight, [NPred_2, \dots, NPred_n]).$$

Figure 71 illustrates definition 15, i.e. the tree structure of a nested density-based decomposition.

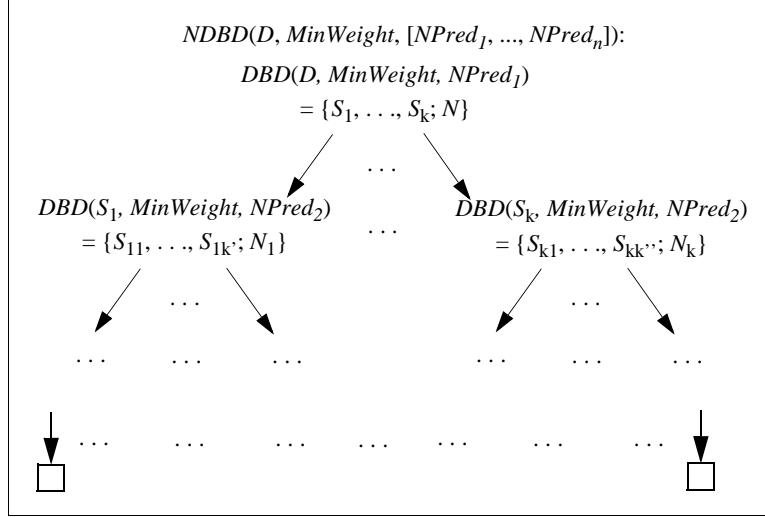


Figure 71: Structure of a nested density-based decomposition

If $NDBD(D, MinWeight, [NPred_1, \dots, NPred_n])$ is a nested density-based decomposition of D , then n is called the height of the decomposition tree. A level i in a nested density-connected set corresponds to a density-based decomposition of the database D with respect to the specific parameters $NPred_i$ and $MinWeight$ used at height i of the tree. Therefore, we define a *clustering-level* of a nested density-based decomposition as the set of all density-connected sets in the tree with respect to the same parameters used at level i ; the noise of a clustering-level is defined to be the union of all noise sets N from the root down to the respective level i .

Definition 16: (*clustering-level of a nested density-based decomposition*)

Let $NC = NDBD(D, MinWeight, [NPred_1, \dots, NPred_n])$ be a nested density-based decomposition of D , let N_1, \dots, N_x be sets of noise contained in the density-based decompositions from the root down to the level $i-1$.

Furthermore, let $DBD(S^1, MinWeight, NPred_i) = \{S^1_1, \dots, S^1_{m1}; N^1\}, \dots, DBD(S^j, MinWeight, NPred_i) = \{S^j_1, \dots, S^j_{mj}; N^j\}$ be the nodes at level i of the tree NC . Then, the *clustering-level* i of NC , is defined as:

$$clustering-level(NC, i) = \{S_1, \dots, S_k; N\}$$

such that

$$\{S_1, \dots, S_k\} = \{S^1_1, \dots, S^1_{m1}\} \cup \dots \cup \{S^j_1, \dots, S^j_{mj}\}, \text{ and}$$

$$N = N_1 \cup \dots \cup N_x \cup N^1 \cup \dots \cup N^j.$$

Lemma 10: Let $NC = NDBD(D, MinWeight, [NPred_1, \dots, NPred_n])$ be a nested density-based decomposition of D . Then, a clustering-level $clustering-level(NC, i) = \{S_1, \dots, S_k; N\}$ of NC is a (“simple”) density-based decomposition of the database D with respect to $NPred_i$ and $MinWeight$.

Proof (sketch): To prove the lemma, we show that the conditions 1) to 5) of definition 8 hold for the clustering level.

1) $clustering-level(NC, i) = \{S_1, \dots, S_k; N\}$, $k \geq 0$: by definition.

2) $S_1 \cup \dots \cup S_k \cup N = D$: Obviously, $S_1 \cup \dots \cup S_k \cup N \subseteq D$. Let $o \in D$. Object o is either a core object at level i , i.e. o is a core object with respect to $NPred_i$ and $MinWeight$ or o is not a core object at level i . If o is core object at level i then o is also a core object at all levels $j \leq i$ because $N_{NPred_{i+1}}(o) \subseteq N_{NPred_i}(o)$ and the predicate $MinWeight$ is monotonous. Then, there must be a path in the tree of the nested density-based decomposition NC from the root down to level i such that o is always contained in one of the density-connected sets of the density-based decompositions associated with a node in the tree. But then, o is contained in one of the sets S_1, \dots, S_k . In the other case, if o is not a core object at level i , there is a smallest level $j \leq i$ such that o is not a core object at level j . If $j < i$, then o is contained in one of the noise sets of level $j+1$, and therefore o is also contained in the set N . If $j = i$ then o is either contained in one of the density-connected sets S_i , $1 \leq i \leq k$,

or o is contained in one of the noise sets of level i depending on whether o is density-reachable or not. Thus, in all cases $o \in S_1 \cup \dots \cup S_k \cup N$, which means that $D \subseteq S_1 \cup \dots \cup S_k \cup N$.

3) For all $i \leq k$: S_i is a density-connected set with respect to $NPred_i$ and $MinWeight$ in D by definition.

4) If there exists S such that S is a density-connected set in D with respect to $NPred_i$ and $MinWeight$ then there also exists an $i \leq k$ and $S = S_i$: Let S be a density-connected set in D with respect to $NPred_i$ and $MinWeight$. Let $o \in S$ be a core object in S . Then, as already shown for condition 2), o is also a core object at all levels $j \leq i$ and thus o must be contained in one of the sets S_j , $1 \leq j \leq k$. However, then by lemma 5 it follows that $S = S_j$.

5) $N = D \setminus (S_1 \cup \dots \cup S_k)$: by condition 2) and the obvious fact that noise objects are not density-reachable, i.e. $N \cap (S_1 \cup \dots \cup S_k) = \emptyset$. \square

For the reasons indicated in section 7.1.1, we may be interested in several clustering levels of a nested density-based decomposition of a database D . In general, however, clustering levels are meaningful only if we have a certain *type* of a neighborhood predicate which can be specialized such that different specializations represent the different clustering levels.

The most important type of neighborhood for this purpose is a distance-based neighborhood, i.e. $NPred(o, o')$ iff $|o - o'| \leq \epsilon$, where different values of ϵ , for instance $\epsilon_1 \geq \dots \geq \epsilon_n$, correspond to different clustering levels. Another type of a neighborhood predicate which may be adapted for the specification of clustering levels is a neighborhood predicate combining a spatial neighborhood with a non-spatial selection condition S , i.e. $NPred(o, o')$ iff $spatial-neighbors(o, o')$ and $S(o)$ and $S(o')$. Then, different clustering levels can be induced by different selection conditions S_1, \dots, S_n becoming more and more restrictive, e.g. $S_1 = A_1(N)$, $S_2 = A_1(N) \wedge A_2(N)$, ..., $S_n = A_1(N) \wedge \dots \wedge A_n(N)$.

7.2 Algorithm H-GDBSCAN

In the following section, we will see that a nested density-based decomposition, and hence all clustering levels, can be computed in a similar way and, most important, in nearly the same time as the computation of a “simple” or “flat” density-based decomposition. This is due to the fact that “smaller” density-connected sets which are contained in a larger density-connected set (*cf.* lemma 9) can be computed without much extra cost while computing the larger density-connected set.

The most expensive part of the construction of a density-connected set is the retrieval of the $NPred$ -neighborhood for all objects from the databases, especially with respect to the number of I/O operations (see chapter 4, section 4.3). However, only a small amount of additional computations is needed to determine a neighborhood $N_{NPred_2}(p)$ for an object p if $N_{NPred_2}(p) \subseteq N_{NPred_1}(p)$ and the neighborhood $N_{NPred_1}(p)$ is already available. This is true because we just have to scan the “larger” neighborhood $N_{NPred_1}(p)$ to find all objects located in the neighborhood $N_{NPred_2}(p)$, instead of consulting the database again.

To make use of this fact, we have to modify our GDBSCAN algorithm in such a way that only a database access for the “largest” neighborhood is necessary when constructing several clustering levels simultaneously. That means, we only want to perform neighborhood queries for the $NPred_1$ -neighborhood from an ordered sequence $[NPred_1, \dots, NPred_n]$ of neighborhood predicates.

We present two different versions of a modified GDBSCAN algorithm. The first is a general version to construct all clustering levels of a nested density-based decomposition with respect to $NPred_1, \dots, NPred_n$ and $MinWeight$. The second is a more specialized version only for distance-based neighborhood predicates. The advantage of the second algorithm is that we do not produce all clustering levels for a set of given distances $\epsilon_1, \dots, \epsilon_n$ explicitly. We just create an *order* of the database

enhanced with additional information using the largest distance ϵ_1 . Then, we can easily extract from this order all clustering levels corresponding to arbitrary distances ϵ , assuming only that $\epsilon \leq \epsilon_1$.

7.2.1 Multiple Clustering Levels

To detect multiple clustering levels in a single pass over the database, it is possible to use an algorithm which is very similar to GDBSCAN. However, our algorithm must process several “parameters” at the same time which forces us to obey a specific order of objects in the function which expands clusters. In the following, we present the algorithm to construct hierarchical layers of clusters in more detail.

The main loop H-GDBSCAN is nearly the same as GDBSCAN (*cf.* figure 31). The difference consists only in passing the additional parameters $NPred_1, \dots, NPred_n$ to the function `MultipleExpandCluster`. Otherwise, it works as if using simple GDBSCAN for the largest $NPred$ -neighborhood $NPred_1$ (see figure 72).

```

H-GDBSCAN (SetOfObjects, [NPred1, ..., NPredn], MinWeight)
// SetOfObjects is UNCLASSIFIED; Object.Processed = FALSE
ClusterIdL1 := nextId(NOISE);
FOR i FROM 1 TO SetOfObjects.size DO
  Object := SetOfObjects.get(i);
  IF NOT Object.Processed THEN
    IF MultipleExpandCluster(SetOfObjects, Object, ClusterIdL1,
                           [NPred1, ..., NPredn], MinWeight)
      THEN ClusterIdL1 := nextId(ClusterIdL1)
    END IF;
  END IF;
END FOR;
END; // H-GDBSCAN

```

Figure 72: Algorithm H-GDBSCAN

Also the structure of the function `MultipleExpandCluster` (see figure 73) does not differ much from the function `ExpandCluster` for non-hierarchical GDBSCAN (cf. figure 32, chapter 4). Only the additional parameters $NPred_1, \dots, NPred_n$ are passed to the class `MultipleSeeds` instead of passing a single neighborhood predicate to the more simple class `Seeds`.

```

MultipleExpandCluster(SetOfObjects, Object,
                      CId, [NPred1, ..., NPredn], MinWeight):Boolean;
neighbors := SetOfObjects.neighborhood(Object, NPred1);
Object.Processed := TRUE;
IF MinWeight(neighbors) THEN // core object at level NPred1
  MultipleSeeds.init(CId, [NPred1, ..., NPredn], MinWeight);
  MultipleSeeds.update(neighbors, Object);
  WHILE NOT MultipleSeeds.empty() DO
    currentObject := MultipleSeeds.next();
    neighbors := SetOfObjects.neighborhood(currentObject, NPred1);
    currentObject.Processed := TRUE;
    IF MinWeight(neighbors) THEN
      MultipleSeeds.update(neighbors, currentObject);
    END IF; // MinWeight(neighbors)
  END WHILE; // NOT MultipleSeeds.empty()
  RETURN True;
ELSE // NOT a core object at level NPred1 (and NPred2, ..., NPredn)
  SetOfObjects.changeCId(Object, [NOISE, ..., NOISE]);
  RETURN False;
END IF; // MinWeight(neighbors)
END; // MultipleExpandCluster

```

Figure 73: Function `MultipleExpandCluster`

The outer IF-condition in the function `MultipleExpandCluster` checks the core object property for `Object`, passed from H-GDBSCAN. If `Object` is not a core object at clustering-level 1 (defined with respect to the “largest” neighborhood pred-

icate NPred_1), then the object can also not be a core object at all the other clustering levels given by $\text{NPred}_2, \dots, \text{NPred}_n$ because the MinWeight predicate is assumed to be monotonous. Then, the control is returned to the main loop H-GDBSCAN which selects the next unprocessed object of the database. Otherwise, if Object is a core object at clustering-level 1, then a density-connected set S at level 1 is expanded in the WHILE-loop of the function $\text{MultipleExpandCluster}$. At the same time, all density-connected sets with respect to smaller neighborhood predicates $\text{NPred}_2, \dots, \text{NPred}_n$ - which are completely contained in S (cf. lemma 9) - are constructed as well. The control of this non-trivial task is completely deferred to the class MultipleSeeds which has to analyze the neighborhood of a current object for all neighborhood predicates $\text{NPred}_1, \dots, \text{NPred}_n$ to determine the correct order in which objects are investigated, and to assign correct cluster-ids for the objects at all clustering-levels. The implementation of the class Seeds for GDBSCAN was very simple, i.e. we could use for instance a *stack* or a *queue*. Now, the structure of the class MultipleSeeds is more complicated. In the following, we will illustrate this structure step by step.

We can think of MultipleSeeds as a list of lists, enhanced with additional information. The structure of the multiple seeds-list is illustrated in figure 74.

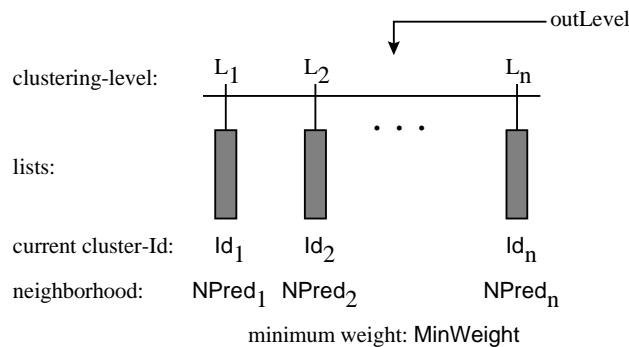


Figure 74: Structure of MultipleSeeds

Each list in `MultipleSeeds` corresponds to one of the neighborhood predicates in the parameter-list of H-GDBSCAN. Roughly speaking, an object o is inserted into the list L_i if i is the largest level such that o is contained in the $NPred_i$ -neighborhood of a `currentObject` and `currentObject` has been a core object with respect to $NPred_i$. A pointer `outLevel` indicates the list, i.e. the clustering-level, from which the last `currentObject` has been taken out by the operation `MultipleSeeds.next()`. Associated with each clustering-level is a current cluster-id for that clustering-level. The cluster-ids of objects are assigned on the basis of these “cluster-level-ids”.

We adopt a hierarchical labeling scheme for cluster-ids of objects matching the tree structure of a nested density-based decomposition. A cluster-id for an object o is now not a simple data type, but an array of cluster-ids storing one id for each clustering-level, i.e. $o.clusterId = [Id_1, \dots, Id_n]$. If Integers are used for the single ids, and we represent noise by the number 0, a complex cluster-id for object o may have for instance the following entries: [2, 1, 4, 0, 0]. The meaning is that o belongs to cluster 2 at level 1, and o also belongs at level 2 to the cluster 1 within cluster 2 of level 1, and so on (o is a noise object at level 4 and level 5 in this example). In general, all objects which belong to the same density-connected set at a certain clustering-level share the same prefix of their cluster-ids, i.e.: if o and o' belong to the same density-connected set at clustering-level i and $o.clusterId = [Id_1, \dots, Id_n]$, and $o'.clusterId = [Id'_1, \dots, Id'_n]$ then it holds that $Id_1 = Id'_1, \dots, Id_i = Id'_i$.

We will now look at the methods of the class `MultipleSeeds` in greater detail. The most simple one is the initialization of a new instance when a new density-connected set at the first clustering-level has to be expanded. A call of the method `MultipleSeeds.init(ClId, [NPred_1, \dots, NPred_n], MinWeight)` results in a multiple seeds-list as illustrated in figure 75. All sub-lists are empty, and all but the current cluster-id for the first clustering-level, i.e. L_1 , are set to NOISE; `outLevel` points to the first list L_1 .

MultipleSeeds after `init(ClId, [NPred1, ..., NPredn], MinWeight)`:

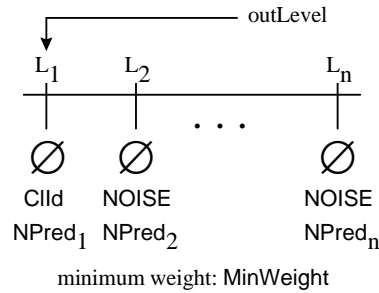


Figure 75: Method `MultipleSeeds::init()`

In the course of execution, objects are inserted into `MultipleSeeds` by the method `MultipleSeeds.update` resulting in a state of the multiple seeds-list in which some of the sub-lists contain objects, and some sub-lists may be empty (see figure 76, left, for an illustration). The intuitive meaning of such a state can be described as follows. An object o which is contained in the list L_i is a candidate for further expansion of all current density-connected sets at all levels $j \leq i$. The reason for this is that o has been inserted into the list L_i only because o has been in the $NPred_i$ -neighborhood of another object p which has been a core object at level i , (which was also the largest i such that p has been a core object). But then, it holds that p has also been a core object at all levels $j \leq i$ and o was contained also in these $NPred_j$ -neighborhoods, because if $j \leq i$ then $N_{NPred_i}(p) \subseteq N_{NPred_j}(p)$.

In such an arbitrary state of the `MultipleSeeds`, the next object from the set of candidates for further expansion has to be selected. We already indicated that because of expanding density-connected sets for all given clustering-levels simultaneously, we always have to obey a specific order for the selection of candidates. Obviously, we must always select the next elements from `MultipleSeeds` such that density-connected sets with respect to smaller neighborhoods are finished first.

More precisely: Assume, there are two current density-connected sets S_i and S_j with respect to neighborhoods $NPred_i$ resp. $NPred_j$, where $j < i$. Then, the cluster S_i must be completely contained in the cluster S_j . Therefore, in such an incomplete state, we must select as candidates for further expansion of the cluster S_j those objects first which are also candidates for the cluster S_i before considering other candidates for S_j . For an arbitrary number of clustering-levels this means that we must select the next object from the set of candidates for the unfinished density-connected sets with respect to the smallest neighborhood. Consequently, we have to select an object from that sub-list of `MultipleSeeds` having the largest or “deepest” clustering-level which is not empty. This is performed by the method `MultipleSeeds.next()`, depicted in figure 76, which also deletes this object from the multiple seeds-list.

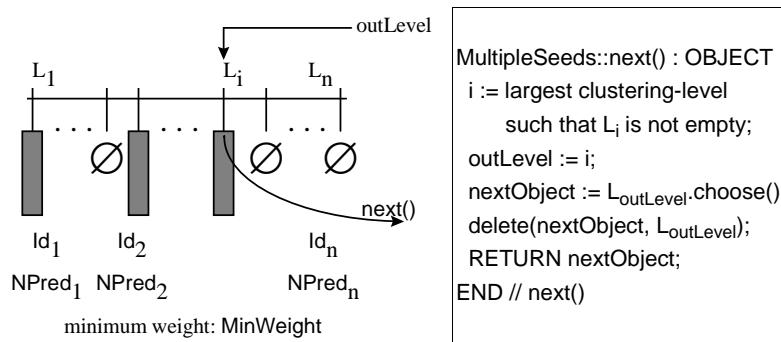


Figure 76: Method `MultipleSeeds::next()`

The method `MultipleSeeds.update(neighbors, currentObject)` has to accomplish three important tasks. First, the objects contained in a current `neighbors` set must be correctly inserted into the sub-lists of `MultipleSeeds`. Second, a cluster-id must be assigned to the objects, and third, the current cluster-level-ids of the class `MultipleSeeds` must be correctly incremented. We will consider these tasks one after another. However, before we can understand the update of a multiple seeds-list,

we have to be familiar with the concepts of a “core-level” and a “reachability-level” for objects.

Definition 17: (*core-level of an object p*)

Let p be an object from a database D , let $[NPred_1, \dots, NPred_n]$ be a list of neighborhood predicates such that $N_{NPred_{i+1}}(p) \subseteq N_{NPred_i}(p)$ for all $i=1, \dots, n-1$, and let $MinWeight$ be a monotonous predicate for the minimum weight of objects. Then, the *core level* of p denoted as $core-level(p)$ is defined as

$$core-level(p) = \begin{cases} \text{UNDEFINED, if } MinWeight(N_{NPred_1}(p)) = \text{FALSE} \\ \max\{1 \leq j \leq n \mid MinWeight(N_{NPred_j}(p))\}, \text{ else} \end{cases}$$

Figure 77 illustrates the core-level of an object p and a $MinWeight$ predicate defined as $MinWeight(N)$ iff $|N| \geq 4$.

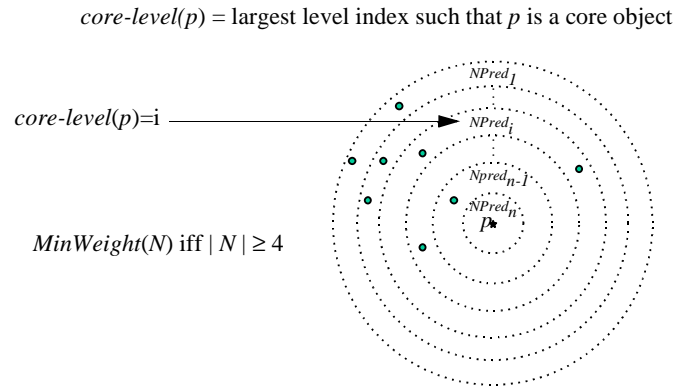


Figure 77: Core-level of an object p

Intuitively, the *core-level* of an object p is the largest level index i in a sequence of levels represented by our list of neighborhood predicates $[NPred_1, \dots, NPred_n]$ at which p is a core object. In other words, if $core-level(p) = i$ then it holds that the condition $MinWeight(N_{NPred_j}(p))$ is true for all $NPred_j$ such that $1 \leq j \leq i$, and the condition is false for $i < j \leq n$. Note that the “largest” level index corresponds to the “smallest” neighborhood around p .

In a call of the method `MultipleSeeds.update(neighbors, currentObject)`, the core-level of the object `currentObject` can be determined easily because the argument `neighbors` contains the $NPred_1$ -neighborhood of `currentObject` which in turn contains all $NPred_j$ -neighborhoods for the larger clustering-levels $j = 2, \dots, n$.

The concept of a *reachability-level* is more closely related to the actual order in which the objects are examined by the algorithm H-GDBSCAN. Therefore, we define the reachability-level of an object p with respect to another object o from which p is density-reachable. There may be more than one object o for which the reachability-level of p is defined.

Definition 18: (*reachability-level of an object p with respect to an object o*)

Let p and o be objects from a database D , let $[NPred_1, \dots, NPred_n]$ be a list of neighborhood predicates such that $N_{NPred_{i+1}}(p) \subseteq N_{NPred_i}(p)$ for all $i=1, \dots, n-1$, and let $MinWeight$ be a monotonous predicate for the minimum weight of objects. Then, the *reachability level* of p with respect to o denoted as *reachability-level*(p, o) is defined as

$$reachability-level(p, o) = \begin{cases} \text{UNDEFINED, if } MinWeight(N_{NPred_1}(o)) = \text{FALSE} \\ \max\{1 \leq j \leq n \mid MinWeight(N_{NPred_j}(o)) \wedge p \in N_{NPred_j}(o)\}, \text{ else} \end{cases}$$

The reachability-level of p with respect to o is the smallest neighborhood at which p is a candidate for further expansion of a density-connected set. More precisely, the reachability-level of object p with respect to o is the largest level index i in a sequence of levels represented by our list of neighborhood predicates $[NPred_1, \dots, NPred_n]$ at which p is directly density-reachable from the core object o . Note that this requires o to be a core object. Therefore, the reachability-level cannot be larger than the core-level of o .

Figure 78 illustrates the reachability-level of objects $p1$, $p2$, and $p3$ with respect to an object o for a sequence of neighborhood predicates $[NPred_1, \dots, NPred_4]$ as depicted and a *MinWeight* predicate defined as $MinWeight(N)$ iff $|N| \geq 4$.

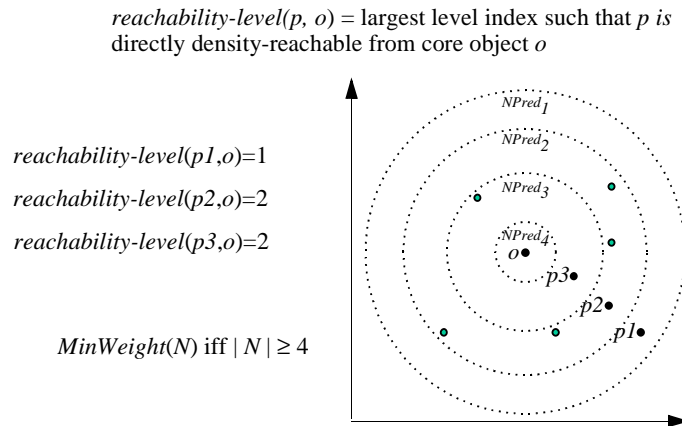


Figure 78: Reachability-level of objects $p1$, $p2$, $p3$ with respect to o

The object o is a core object only in level one and level two, object $p1$ is found in $NPred_1$ -neighborhood of o , $p2$ is found in the $NPred_2$ -neighborhood of o , and $p3$ is found in the $NPred_3$ -neighborhood of o . Then, the reachability-level of $p2$ with respect to o is equal to 2. But also the reachability-level of $p3$ with respect to o is equal to 2 (and not equal to 3) because $p3$ is directly density-reachable from o

only with respect to $NPred_i$ and $MinWeight$ if $i \leq 2$. The reachability-level of $p1$ with respect to o is equal to 1 because $p1$ is not contained in the $NPred_i$ -neighborhood of o for clustering-levels i which are larger than 1.

In our algorithm, we will use an attribute `reachability-level` for each object p . In the beginning, we set `p.reachability-level` to $reachability-level(p, o)$ for an object o which is not a core object at clustering-level one, i.e. we the reachability-level to UNDEFINED. Consequently, the reachability-level of objects is only defined for objects which are contained in the multiple seeds-list because if they are inserted into `MultipleSeeds` they must be at least density-reachable from a core object at level one.

For our algorithm, the reachability-level of an object p determines the position of p in the multiple seeds-list which means that `p.reachability-level` may change as long as p is a member of the multiple seeds-list. The object p has always to be contained in that sub-list of `MultipleSeeds` which is associated with the largest reachability-level of p . As we can easily see, this requirement is a necessary pre-condition for the method `MultipleSeeds.next()` to select the next element correctly. However, this implies that the reachability-level of an object p which is already a member of the multiple-seeds list has to be changed if p becomes again directly density-reachable from another core object at a larger clustering-level.

Figure 79 depicts an example for a changing reachability-level of an object p in the course of execution of the algorithm. Let $o1$ be the first object selected for an expansion of clusters. Then, the reachability-level of p with respect to $o1$ (at time t) is equal to two because $o1$ is a core object at levels one and two and p is contained in the $NPred_2$ -neighborhood of $o1$. Assume that the next object selected by the method `MultipleSeeds.next()` is object $o2$. Now, at time $t+1$, the object p is directly density-reachable from $o2$ at level three and therefore, the reachability-level of p at time $t+1$ is changed to three.

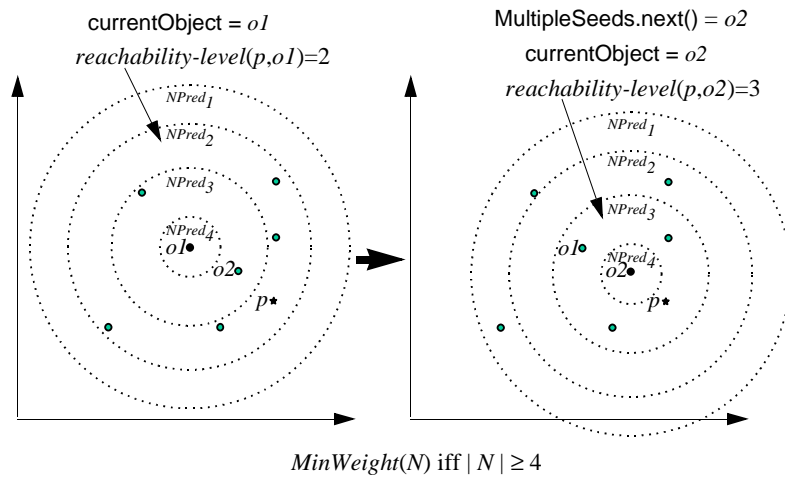


Figure 79: Changing reachability-level of an object p

Now, we can explain the method `MultipleSeeds.update` which is called by the function `MultipleExpandCluster`. The algorithm for the update of a multiple seeds-list is depicted in figure 80. As already indicated, it consists of the three parts “incrementing cluster-level-ids”, “insertion of yet unprocessed objects”, and “assignment of correct cluster-ids to each object”. To see that our algorithm is correct, we have to look at these different parts in more detail.

The *insertion* of the objects from the set `neighbors` into the multiple seed-list is controlled by two variables, `Object.processed` and `Object.reachability_level`. ‘`Object.processed = TRUE`’ holds only if a neighborhood query has already been performed for `Object` - either because it has been the first object considered for expanding a new density-connected set at level one or because it has been selected for further expansion of a cluster by the method `MultipleSeeds.next()`, i.e. it has already been a member of the multiple seeds-list. Obviously, these objects must not be inserted into the multiple seeds-list (again). Therefore, `Object` is considered for

```

MultipleSeeds::update(neighbors, CenterObject);

c_level := core_level(CenterObject);
IF c_level > outLevel THEN
  currentClusterIds[outLevel + 1] = nextId(currentClusterIds[outLevel + 1]);
  FOR i := outLevel + 2 TO max_level DO
    IF i ≤ c_level THEN
      currentClusterIds[i] = nextId(NOISE);
    IF c_level < i ≤ max_level THEN
      currentClusterIds[i] = NOISE;
    END FOR;
  SetOfObjects.changeCId(CenterObject, currentClusterIds);

FORALL Object FROM neighbors DO
  previous_r_level := Object.reachability_level;
  new_r_level := reachability_level(Object, CenterObject, c_level);
  IF NOT Object.Processed THEN
    IF previous_r_level = UNDEFINED THEN
      Object.reachability_level := new_r_level;
      insert(Object, Lnew_r_level);
    ELSE
      IF new_r_level > previous_r_level THEN
        Object.reachability_level := new_r_level;
        delete(Object, Lprevious_r_level);
        insert(Object, Lnew_r_level);

FOR i := previous_r_level+1 TO max_level DO
  IF i ≤ new_r_level THEN
    Object.CIds[i] := currentClusterIds[i];
  IF new_r_level < i ≤ max_level THEN
    Object.CIds[i] := NOISE;

END; // MultipleSeeds::update

```

*Increment
cluster-level-ids*

Insert

*Assign
cluster-ids*

Figure 80: Method MultipleSeeds::update()

insertion only if `Object.processed = FALSE` holds. This condition, however, does not exclude that `Object` may already be a member of `MultipleSeeds` which can be decided by looking at the variable `Object.reachability_level`.

If `Object.reachability_level = UNDEFINED`, it holds that `Object` is not yet a member of the multiple seeds-list, because, in any case, if an object is inserted into `MultipleSeeds`, its reachability-level will be set to a defined value. For those objects, we simply compute the new reachability-level 'new_r_level' with respect to the current core object (`CenterObject`), and insert `Object` into the sub-list of `MultipleSeeds` which is associated with `new_r_level`. In the other case, i.e. if `Object.reachability_level` has a value different from `UNDEFINED`, then `Object` must have been directly density-reachable from a previously selected core object, and hence, has been inserted into the multiple seeds-list. Because it also holds that `Object.processed = FALSE`, `Object` must still be a member of `MultipleSeeds`. In this case, we check if the new reachability-level of `Object` is larger than the previous reachability-level. If yes, the reachability-level of `Object` must be changed to the new reachability-level and `Object` must be moved to the corresponding sub-list for that new reachability-level.

The *assignment of cluster-ids* for objects is made on the basis of the reachability-level, i.e. it is based on the direct density-reachability from the current core object. However, new cluster-ids are only assigned for levels greater than the previous reachability-level of an object (we assume that `UNDEFINED` is the smallest reachability-level). For levels which are smaller than or equal to the previous reachability-level, a cluster-id has already been set or it is `UNDEFINED`. Such cluster-ids which are already set will not change because we are still expanding the same density-connected set at these levels. Only if the reachability-level of an object increases, we have to assign the current cluster-level-ids from the previous reachability-level up to the new reachability-level. That is necessary because `Object` has become directly density-reachable from the core object `CenterObject` at larger clustering-levels, and therefore belongs to the current clusters at these larger

level. For clustering-levels that are even larger than the new reachability-level of *Object*, the cluster-id has to be set to NOISE because *Object* is not yet density-reachable from a core object at these levels. Obviously, this assignment of cluster-ids for objects is only correct if the cluster-level-ids are correct.

The *setting for the cluster-level-ids* is controlled by the core-level of the current core object (*CenterObject*). However, nothing is changed if the core-level of the *CenterObject* is smaller than the current *outLevel*, i.e. smaller than the level from which *CenterObject* was selected. In this case, *CenterObject* has been density-reachable from another core object at level one up to *outLevel* and its cluster-ids are already set correctly. Furthermore, no current cluster has been finished because *CenterObject* has been a candidate for further expansion of clusters at levels one up to *outLevel*. That means, we are just going to add objects which are directly density-reachable from *CenterObject* to the current clusters at these levels. Consequently, the cluster-level-ids do not have to be changed.

In the other case, if the core-level of *CenterObject* is larger than the current *outLevel*, a new cluster must be started at all levels which are larger than the current *outLevel* up to the core-level of *CenterObject*. Because we have an unfinished cluster *C* at *outLevel*, the cluster-id for *outLevel* is not changed. If there had been clusters at larger levels which were completely contained in the cluster *C*, these clusters are now all finished because otherwise *outLevel* would have been larger. That means that the *next* cluster within cluster *C* is started at level *outLevel*+1. Consequently, the cluster-id for this level is incremented. This case corresponds to an empty seeds-list in the execution of the non-hierarchical GDBSCAN algorithm for the *NPred*-parameter which is associated with *outLevel*+1. If *CenterObject* is also a core object at even larger levels, the new clusters which are started at these levels are completely contained in the newly created cluster at level *outLevel*+1. Because we adopted a hierarchical labeling scheme for cluster-level-ids, the cluster-ids for these consecutive levels must be set to the first valid cluster-id in the order of all cluster-ids.

At levels larger than the core-level of `CenterObject`, there exists no current cluster because `CenterObject` is not a core object at levels larger than its core-level. Therefore, we set the cluster-level-ids for these levels equal to NOISE.

The call of the method `SetOfObjects.changeCId(CenterObject, currentClusterIds)` sets the cluster-ids of `CenterObject` to the current cluster-level-ids. Note that this is just an abbreviation. For smaller levels than `outLevel`, we do not have to change the cluster-ids of `CenterObject` because they are already equal to the current cluster-level-ids.

Now it is easy to see that the hierarchical GDBSCAN algorithm actually computes a nested density-based decomposition of a database because we can construct a flat density-based decomposition for each level i and also use a hierarchical labeling scheme for cluster-ids. A density-connected set at level i is given by all objects which have the same prefix of their cluster-ids. If p and q are density-connected at a level i , i.e. they belong to the same density-connected set at level i , then p and q are also density-connected at all smaller levels j ($1 \leq j \leq i$). Therefore, p and q will share the same prefix of their cluster-ids, i.e. $p.clusterIds=[Id_1, \dots, Id_n]$, $q.clusterIds=[Id'_1, \dots, Id'_n]$, and $Id_j = Id'_j \neq \text{NOISE}$, for all $1 \leq j \leq i$. Then, the noise at level i is simply given by the set of all objects o having the cluster-id NOISE at level i , i.e. $o.clusterIds=[Id_1, \dots, Id_n]$ and $Id_i = \text{NOISE}$. As we have seen, our algorithm assigns the cluster-ids in exactly this way.

We performed several experiments to measure the run-time of the algorithm H-GDBSCAN. As expected, the run-time of H-GDBSCAN using the parameters $([NPred_1, \dots, NPred_n], MinWeight)$ is very similar to the run-time of the non-hierarchical GDBSCAN algorithm using the largest neighborhood, i.e. using the parameters $(NPred_1, MinWeight)$. This is due to the fact that the additional cost for managing a multiple seeds-list instead of a regular seeds-list is dominated by the cost of the region queries. Actually, the overall run-time of H-GDBSCAN in all experiments was between 1.3 and 1.5 times the run-time of GDBSCAN.

7.2.2 Ordering the Database with respect to Cluster Structure

A nested density-based decomposition is very useful for cluster analysis. However, there are some problems considering the number of different levels and their corresponding *NPred*-values. For example, the most important neighborhood predicates are distance-based, and for distance-based neighborhood predicates there are an infinite number of possible distance values. In such applications, we do not know the number of different levels needed to reveal the inherent cluster structure of a data set in advance. Even if we would include a very large number of different clustering-levels - which requires a lot of secondary memory to store the different cluster-ids for each point and clustering level - we may miss the “interesting” clustering levels because the “correct” parameters were not included in the parameter list.

In this section we present a technique to overcome the mentioned problems for distance-based *NPred*-neighborhood predicates and a monotonous *MinWeight* predicate. The basic idea is to run an algorithm which is very similar to the hierarchical algorithm presented in the previous section but which simply produces a special order of the database with respect to a so called “generating distance”. Instead of computing and storing cluster-ids for different clustering-levels, this algorithm stores only two additional values for each object: the core-distance and the reachability-distance. These notions are formally introduced below.

The ordering with respect to the cluster structure of a data set can be used for a fast interactive approximation of “correct” or “interesting” clustering-levels because it allows a fast computation of *every* clustering level with respect to a smaller distance than the generating distance. However, we will argue that this is not the best way to use the cluster-order for the purpose of cluster analysis. Since the cluster-order of a data set contains the information about the inherent clustering structure of that data set (up to the generating distance), we can use it as a stand-alone tool which offers good opportunities for an interactive cluster analysis procedure.

In the following, we restrict our considerations to distance-based neighborhood predicates and use the comparison of the cardinality of a neighborhood set to a threshold as *MinWeight* predicate, i.e. we restrict the discussion to the parameter specialization for DBSCAN as introduced in definition 9. For a better understanding of the following algorithm, we use as parameters a distance ϵ and a value *MinPts* instead of *NPred* and *MinWeight*.

The algorithm to create an extended order of a database with respect to the cluster structure is similar to the hierarchical version of GDBSCAN. To explain this algorithm, we need the notion of a *core-distance* and a *reachability-distance* which correspond directly to the core-level and the reachability-level introduced in the previous section for H-GDBSCAN.

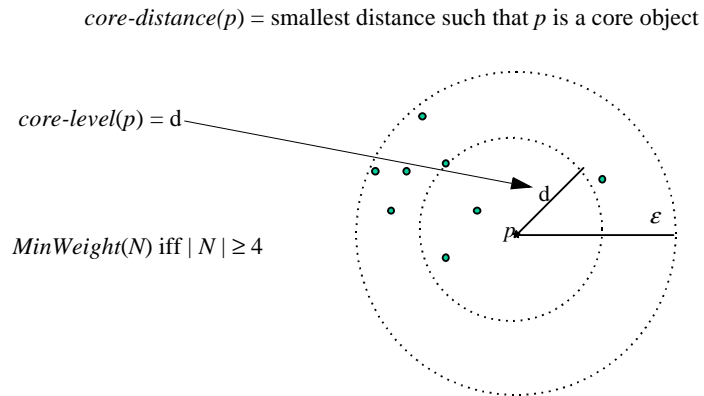
Definition 19: (*core-distance of an object p*)

Let p be an object from a database D , let ϵ be a distance value, let $N_\epsilon(p)$ be the ϵ -neighborhood of p , and let *MinPts* be a threshold value for the minimum weight of $N_\epsilon(p)$. Then, the *core-distance* of p denoted as *core-distance*(p) is defined as

$$\text{core-distance}(p) = \begin{cases} \text{UNDEFINED, if } |N_\epsilon(p)| < \text{MinPts} \\ \text{MinPts-distance}(p), \text{ else} \end{cases}$$

The core-distance of an object p is simply the smallest distance to an object in its ϵ -neighborhood such that p would be a core object if we would use this distance instead of ϵ ; the core-distance is UNDEFINED if this distance would be larger than the value of ϵ .

The intuition behind this notion is the same as for the core-level defined in definition 17. The difference is only that we do not assume different predefined levels. Figure 81 illustrates the notion of a core-distance.



Definition 20: (reachability-distance of an object p with respect to an object o)

Let p and o be objects from a database D , let $N_\epsilon(o)$ be the ϵ -neighborhood of p , and let $MinPts$ be a threshold value for the minimum weight of $N_\epsilon(o)$. Then, the reachability-distance of p with respect to o denoted as $reachability-distance(p, o)$ is defined as

$$reachability-distance(p, o) = \begin{cases} \text{UNDEFINED, if } |N_\epsilon(p)| < MinPts \\ \max(core-distance(o), distance(o, p)), \text{ else} \end{cases}$$

Again, the intuition behind the notion of a reachability-distance is the same as for the reachability-level defined in definition 18 except, that we do not assume different predefined levels. Figure 82 illustrates the notion of a reachability-distance.

$reachability-level(p, o) =$ smallest distance such that p is directly density-reachable from core object o

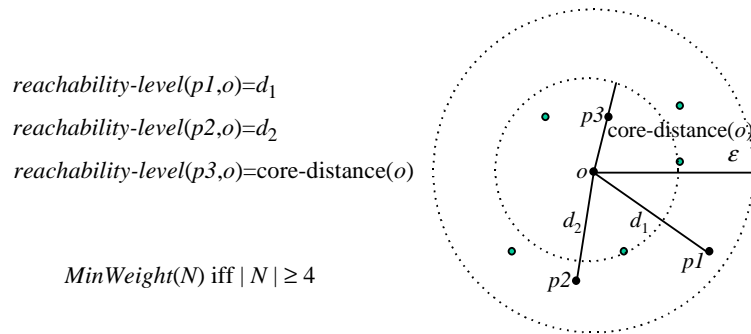


Figure 82: Reachability-distance of objects $p1, p2, p3$ with respect to o

As for the reachability-level defined in the previous section, the reachability-distance of an object p depends on the core object with respect to which it is calculated. A current value is stored for each object in the execution of our algorithm and this value may change over time.

Now we can explain the algorithm, called H-DBSCAN-ORDER, to generate an order of a data set with respect to the clustering structure. In principle, H-DBSCAN-ORDER works like H-GDBSCAN using a “generating distance” ϵ corresponding to the largest neighborhood predicate $NPred_1$. However, we do not assign cluster-level-ids but store the objects in the order in which they are processed. Also stored in this order are the core-distance and the last reachability-distance for each object. We will see that this information is sufficient to extract the clustering level with respect to any distance d which is smaller than the generating distance ϵ from this order. Consequently, we do not need any further neighborhood predicates or distances as input parameter for H-DBSCAN-ORDER.

Figure 83 illustrates the main loop of the algorithm H-DBSCAN-ORDER. It is easy to see the similarity to the main loop of H-GDBSCAN. The only difference - except the more specialized parameter list - is that we open at the beginning a file `OrderedFile` for writing and close this file after ending the loop. Each object from a database `SetOfObjects` is simply handed over to a procedure `ExpandClusterOrder` if the object is not yet processed.

```

H-DBSCAN-ORDER (SetOfObjects,  $\epsilon$ , MinPts, OrderedFile)
OrderedFile.open();
FOR i FROM 1 TO SetOfObjects.size DO
Object := SetOfObjects.get(i);
IF NOT Object.Processed THEN
ExpandClusterOrder(SetOfObjects, Object,  $\epsilon$ , MinPts, OrderedFile)
END IF;
END FOR;
OrderedFile.close();
END; // H-DBSCAN-Order

```

Figure 83: Algorithm H-DBSCAN-ORDER

The structure of the procedure `ExpandClusterOrder` does not differ much from the function `ExpandCluster` for simple GDBSCAN (*cf.* figure 32, chapter 4) or the function `MultipleExpandCluster` for H-GDBSCAN (*cf.* figure 73, above).

First, we perform an ϵ -range query for the object `Object`, passed from the main loop H-GDBSCAN-ORDER, set its reachability-distance to UNDEFINED and determine its core-distance. Then, `Object` is written to `OrderedFile`. In the IF-condition the procedure `ExpandClusterOrder` checks the core object property for `Object`. If `Object` is not a core object at the generating distance ϵ , the control is simply returned to the main loop H-GDBSCAN-ORDER which selects the next unprocessed object of the database. Otherwise, if `Object` is a core object at the generating distance ϵ , we proceed as if expanding a density-connected set S for this distance ϵ in the WHILE-loop of `ExpandClusterOrder`. However, we store each object select-

ed from the seeds-list after its core-distance is determined. The reachability-distance of each object selected from the seeds-list is then already set to a defined value. Managing the reachability-distances for the objects is handled by the class `OrderSeeds`. The pseudo-code for the procedure `ExpandClusterOrder` is depicted in figure 84.

```

ExpandClusterOrder(SetOfObjects, Object, ε, MinPts, OrderedFile);
neighbors := SetOfObjects.neighborhood(Object, ε);
Object.Processed := TRUE;
Object.reachability_distance := UNDEFINED;
Object.setCoreDistance(neighbors, ε, MinPts);
OrderedFile.write(Object);
IF Object.core_distance <> UNDEFINED THEN
  OrderSeeds.init();
  OrderSeeds.update(neighbors, Object);
  WHILE NOT OrderSeeds.empty() DO
    currentObject := OrderSeeds.next();
    neighbors := SetOfObjects.neighborhood(currentObject, ε);
    currentObject.Processed := TRUE;
    currentObject.setCoreDistance(neighbors, ε, MinPts);
    OrderedFile.write(currentObject);
    IF currentObject.core_distance <> UNDEFINED THEN
      OrderSeeds.update(neighbors, currentObject);
    END IF;
    OrderedFile.write(currentObject);
  END WHILE;
END IF;
END; // ExpandClusterOrder

```

Figure 84: Procedure `ExpandClusterOrder`

Again, as for H-GDBSCAN, the structure of the class `OrderSeeds` is crucial for the execution of our algorithm. However, the structure is much simpler than the structure of the class `MultipleSeeds` because we do not need to assign correct clus-

ter-level-ids for the objects. The objects contained in `OrderSeeds` are stored in a priority-queue, sorted by their reachability-distance. Intuitively, this is similar to a `MultipleSeeds`-list in H-GDBSCAN for an infinite number of clustering levels smaller than the largest level determined by $NPred_1$.

The method `OrderSeeds::next()` selects the first object from the priority-queue, i.e. an object having the smallest reachability-distance in the queue. The method `OrderSeeds::update(...)` is depicted in figure 85. The method is easy to understand if we compare it with the method `MultipleSeeds::update(...)` (cf. figure 80, above). The pseudo-code for `OrderSeeds::update` corresponds almost one-to-one to the “insert”-part of the method `MultipleSeeds::update(...)`. The reachability-distance for each object in the set `neighbors` is determined with respect to the center-object. Objects which are not yet in the priority-queue are simply inserted with their reachability-distance. Objects which are already in the queue are moved further to the top of the queue if their new reachability-distance is smaller than their previous reachability-distance. Figure 85 depicts the code for `OrderSeeds::update()`.

```

OrderSeeds::update(neighbors, CenterObject);
c_dist := CenterObject.core_distance;
FORALL Object FROM neighbors DO
  IF NOT Object.Processed THEN
    new_r_dist := max(c_dist, CenterObject.distance(Object));
    IF Object.reachability_distance = UNDEFINED THEN
      Object.reachability_distance := new_r_dist;
      insert(Object, new_r_dist);
    ELSE // Object already in OrderSeeds
      IF new_r_dist < Object.reachability_distance THEN
        Object.reachability_distance := new_r_dist;
        decrease(Object, r_dist);
      END IF;
    END IF;
  END IF;
END FORALL;
END; // OrderSeeds::update

```

Figure 85: Method `OrderSeeds::update()`

Due to its similarity to the algorithm H-GDBSCAN, the run-time of the algorithm H-DBSCAN-ORDER is the same as the run-time for H-GDBSCAN. Extracting specified clustering levels requires only a single scan over the cluster-ordered data set, i.e. it requires $O(n)$ time, and therefore does not change the overall run-time complexity of the algorithm.

We can extract any density-based clustering from this order with respect to *MinPts* and a clustering-distance ϵ which is smaller than the generating distance d by simply “scanning” the cluster-ordered data set and assign cluster-level-ids depending on the reachability-distance and the core-distance of the objects. Figure 86 depicts the algorithm ExtractClustering for constructing a single density-based decomposition with respect to ϵ and *MinPts* from the cluster-order of a database. Modifying this algorithm to extract multiple clustering-levels simultaneously is a trivial task which is not presented here since it offers no additional insights.

```

ExtractClustering (ClusterOrderedObjects,  $\epsilon$ , MinPts)
// Precondition:  $\epsilon \leq$  generating distance for ClusterOrderedObjects
ClusterId := NOISE;
FOR i FROM 1 TO ClusterOrderedObjects.size DO
  Object := ClusterOrderedObjects.get(i);
  IF Object.reachability_distance >  $\epsilon$  THEN      // also UNDEFINED >  $\epsilon$ 
    IF Object.core_distance  $\leq$   $\epsilon$  THEN
      ClusterId := nextId(ClusterId);
      Object.clusterId := ClusterId;
    ELSE
      Object.clusterId := NOISE;
    END IF
  ELSE // Object.reachability_distance  $\leq$   $\epsilon$ 
    Object.clusterId := ClusterId;
  END IF;
END FOR;
END; // ExtractClustering

```

Figure 86: Algorithm ExtractClustering

To extract a clustering, i.e. to assign cluster-ids to the objects, we first have to look at the reachability-distance of the current object *Object*. If the reachability-distance of *Object* is larger than the clustering-distance ϵ , the object is not density-reachable with respect to ϵ and *MinPts* from any of the object which are located before the current object in the cluster-order. This is obvious because if *Object* would have been density-reachable with respect to ϵ and *MinPts* from a preceding object in the order, it would have been inserted into *OrderSeeds* with a reachability-distance of at most ϵ while generating the cluster-order.

Therefore, if the reachability-distance is larger than ϵ , we look at the core-distance of *Object* and start a new cluster if *Object* is a core object with respect to ϵ and *MinPts*; otherwise, *Object* is assigned to NOISE. Note that the reachability-distance of the first object in the cluster-order is always UNDEFINED and that we assume UNDEFINED to be greater than any real distance ϵ .

If the reachability-distance of the current object is smaller than ϵ , we simply assign this object to the current cluster. Obviously the reachability-distance of *Object* can only be smaller than ϵ if *Object* is density-reachable with respect to ϵ and *MinPts* from a preceding core object *o* in the cluster-order. Because the procedure *ExpandClusterOrder* collects and processes density-reachable objects like the hierarchical GDBSCAN algorithm no object between *o* and *Object* in the order can have a larger reachability-distance than ϵ . This is true because if *o* and *Object* are density-reachable with respect to ϵ and *MinPts*, there is a chain of directly density-reachable objects between *o* and *Object* and all objects in the chain have a reachability-distance smaller than ϵ . Starting from *o*, the object in the chain are processed by *ExpandClusterOrder* before any object having a greater reachability-distance than ϵ because *OrderSeeds* iteratively collects directly density-reachable objects and sort them according to increasing reachability-distance. But then, because no reachability-distance has been greater than ϵ , no new cluster has been started since visiting *o* by the algorithm *ExtractClustering*. Consequently, *Object* will be assigned to the same cluster as *o*.

The clustering created from a cluster-ordered data set by `ExtractClustering` is exactly the same as created by DBSCAN if we set $MinPts \leq 3$, i.e. if there are no border objects (*cf.* lemma 4 and the belonging footnote on page 56). Otherwise, if border objects exist, some of them may be assigned to NOISE when extracted by the algorithm `ExtractClustering`. This happens if they were processed by the algorithm `H-DBSCAN-ORDER` before a core object of the corresponding cluster has been found. We already discussed this case for the GDBSCAN algorithm where these border objects are re-assigned to a cluster when they are found again in the neighborhood of some core object (see page 79). When extracting a clustering from a cluster-order of a data set we cannot recognize these cases because we do not use neighborhood queries any more.

To re-assign those objects assigned to NOISE which actually belong to one of the clusters, we could perform a second pass over the database and look for core objects in the neighborhood of noise objects. However, our experiments indicate that such border objects are very rare and that the resulting clustering differs only very slightly from the clustering produced by DBSCAN when using higher $MinPts$ -values. Therefore, we can omit a second pass over the database after the extraction of clustering without much loss of information.

The extraction of a clustering from a cluster-ordered data set can be understood easily if the problem is represented graphically. For a cluster-order it holds that each object having a lower reachability-distance than ϵ belongs to a cluster with respect to ϵ and $MinPts$ and objects belonging to the same cluster are close to each other in the cluster-order. Furthermore, each object, except the “first” (i.e. the left-most) object belonging to a cluster, having a higher reachability-distance than ϵ is in general a noise object. In principle, we can “see” such a clustering if we plot the reachability-distance values for each object in the cluster-order of a data set and then draw a horizontal line at a reachability-distance of ϵ . Figure 87 depicts this visualization of a density-based clustering which can be extracted from a cluster-ordered data set.

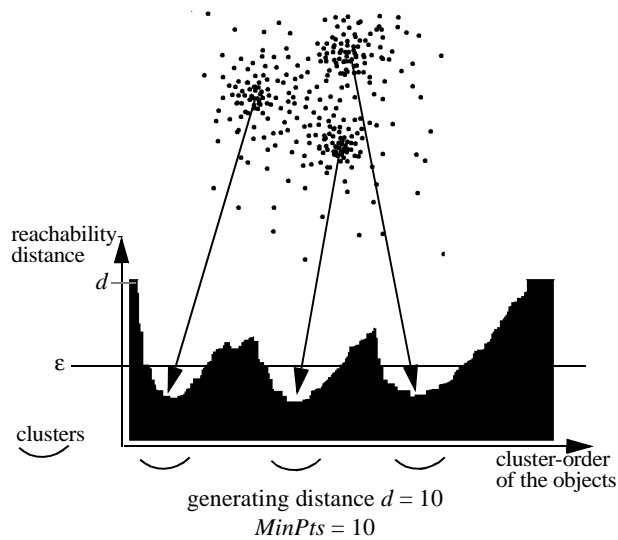


Figure 87: Illustration of clustering level and cluster-order

It is very easy to extract from a cluster-order of a data set not only a “flat” density-based decomposition but also a nested density-based decomposition with respect to an arbitrary number of levels in a single pass over the ordered data set. However, if we look at figure 87 more closely, we can see that there is actually no need to do this for the purpose of data mining. The order of a database with respect to the clustering structure already contains the clustering information in a very comprehensible way. Therefore, we argue that the best way to do cluster analysis in a semi-automatic way is to use the cluster-order of a data set as a stand-alone tool. A user can *see* all clusters of all densities directly in a visualization of the cluster-order - up to a minimum density specified by the generating distance d .

Note that the visualization of the cluster-order is independent of the dimension of the data set. For example, if the objects of a high-dimensional data set would be distributed similar to the distribution of the 2-dimensional data set depicted in

figure 87 (i.e. there are three “Gaussian bumps” in the data set), the “reachability-plot” would also look very similar.

A further advantage of cluster-ordering a data set compared to other clustering methods is that the reachability-plot is rather insensitive to the input parameters of the method, i.e. the generating distance d and the value for $MinPts$. Roughly speaking, the values have just to be “large” enough to yield a good result. The concrete values are not crucial because there is a large range of possible values for which we always can see the clustering structure of a data set when looking at the corresponding reachability-plot. Figure 88 shows the effects of different parameter settings on the reachability-plot for the same data set used in figure 87. In the first plot we used a smaller generating distance d for the second plot we set $MinPts$ to the smallest possible value. Although, these plots look different from the plot depicted in figure 87, the overall clustering structure of the data set can be recognized in these plots as well.

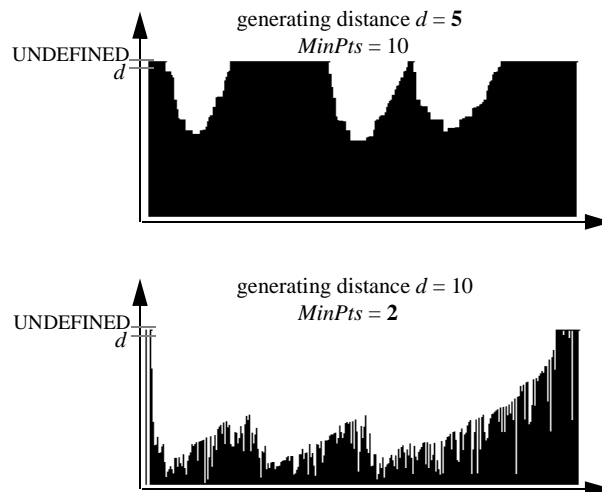


Figure 88: Effects of parameter settings on cluster-order

The influence of the generating distance d on the reachability-plot concerns the number of clustering-levels which we can see in the reachability-plot. The smaller we choose the value of d the more objects may have an UNDEFINED reachability-distance and we may therefore not see clusters of lower density, i.e. clusters where the core objects are core objects only for distances larger than d .

The optimal value for d is the smallest value so that a density-based decomposition of the database with respect to d and $MinPts$ consists of only one cluster containing almost all points of the database. Then, the information of *all* clustering levels will be contained in the reachability-plot. However, there is a large range of values around this optimal value for which the appearance of the reachability-plot will not change significantly. Therefore, we can use rather simple heuristics to determine the value for d which guarantee only that the distance value will not be too small. For example, we can use the expected k -nearest-neighbor distance (for $k = MinPts$) under the assumption that the objects are randomly distributed, i.e. under the assumption that there are no clusters. This value can be determined analytically for a data space DS containing N points. The distance is equal to the radius r of a d -dimensional hyper-sphere S in DS where S contains exactly k points. Under the assumption of a random distribution of the points, the following holds for the volume of S :

$$\frac{Volume_{DS}}{N} \times k = Volume_S$$

The volume of a d -dimensional hyper-sphere S having a radius r is given as

$$Volume_{S(r)} = \frac{\sqrt{\pi^d}}{\Gamma(\frac{d}{2} + 1)} \times r^d$$

Where Γ denotes the Gamma-function. The expression $\Gamma(\frac{d}{2} + 1)$ can be evaluated easily using the following equations: $\Gamma(\frac{1}{2}) = \sqrt{\pi}$ and $x\Gamma(x) = \Gamma(x + 1)$.

The radius r can be now computed as

$$r = d \sqrt{\frac{\text{Volume}_{DS} \times k \times \Gamma(\frac{d}{2} + 1)}{N \times \sqrt{\pi^d}}}$$

This radius may be larger than necessary. However, concerning the efficiency of the algorithm H-DBSCAN-ORDER with respect to large generating distances d , recall that using multiple neighborhood queries yields higher speed-up factors for larger neighborhoods.

The effect of the value for *MinPts* on the visualization of the cluster-order can be seen in figure 88. The overall shape of the reachability-plot is very similar for different *MinPts* values. However, for lower values the reachability-plot will look more jagged and higher values for *MinPts* will significantly smoothen the curve. Our experiments indicate that we will always get good result using any value between 10 and 20 for *MinPts*.

To show that the reachability-plot is very easy to understand, we will finally present some examples. Figure 89 depicts the reachability-plot for a very high-dimensional “real-world” data set. The data set contains 10,000 gray-scale images of 32x32 pixels. Each object is represented by a vector containing the gray-scale value for each pixel. Thus, the dimension of the vectors is equal to 1,024. The Euclidean distance function was used as similarity measure for these vectors.

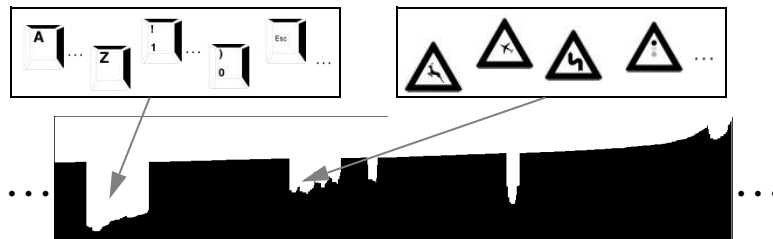
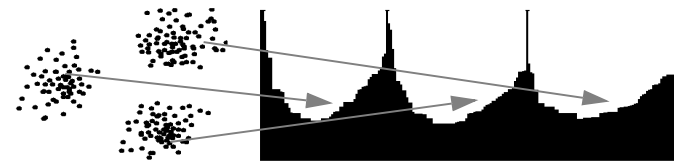
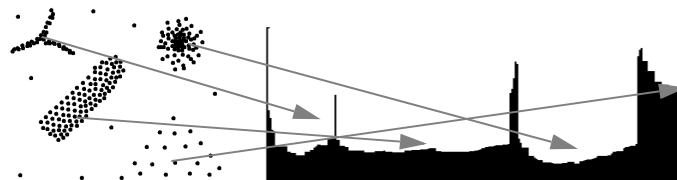


Figure 89: Part of the reachability-plot for a 1,024- d image data set

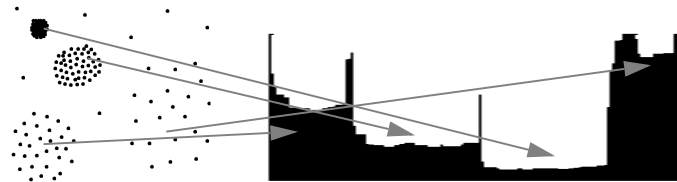
Figure 90 shows further examples of reachability-plots for several data sets having different clustering characteristics. For a better comparison of the real distribution with the cluster-order of the objects, the data sets were synthetically generated in two dimensions.



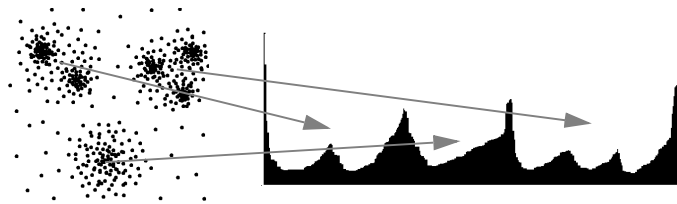
(a) clusters without noise



(b) clusters of different shapes and sizes



(c) clusters of different densities



(d) hierarchical clustering structure

Figure 90: Reachability-plots for data sets having different characteristics

7.3 Summary

In this chapter, the notion of a *nested* density-based decomposition was introduced which is simply a hierarchy of “flat” density-based decompositions.

We presented the algorithm H-GDBSCAN to compute a nested density-based decomposition with respect to a monotonous *MinWeight* predicate and a sequence of neighborhood predicates $NPred_1, \dots, NPred_n$ where for all $i=1, \dots, n-1$ the condition $N_{NPred_{i+1}}(p) \subseteq N_{NPred_i}(p)$ holds. The run-time of the algorithm H-GDBSCAN is nearly the same as the run-time of GDBSCAN for computing a “flat” density-based decomposition with respect to *MinWeight* and $NPred_1$ because density-connected sets with respect to smaller neighborhoods are completely contained in density-connected sets with respect to larger neighborhoods. When computing a nested density-based decomposition H-GDBSCAN proceeds in principle like GDBSCAN. The difference is only that H-GDBSCAN has to obey a certain *order* in which objects are processed.

We developed a more specialized version of a hierarchical clustering algorithm H-DBSCAN-ORDER which is designed for distance-based neighborhood predicates. H-DBSCAN-ORDER does not produce clustering levels explicitly. It just stores the objects in the *order* in which H-GDBSCAN *would* process the objects if we would use all, i.e. an infinite number of distance values (from 0 up to a generating distance d) as input parameters. Furthermore, H-DBSCAN-ORDER does not make any decision with respect to cluster membership (cluster-ids). Instead, it just stores for each object o the information which *would* be used by H-GDBSCAN to assign cluster-ids, i.e. the reachability-distance and the core-distance of o .

From this cluster-order we can extract any clustering level for distance values ϵ if $\epsilon \leq d$. However, the cluster-order is also a powerful “stand-alone” tool for cluster analysis. By visualizing the reachability-distances a user can actually *see* the clustering structure in a database independent of the dimension of the data space.

Chapter 8

Conclusions

In this chapter the achievements of this thesis are shortly summarized and some directions for future research are indicated.

This thesis presented the following *contributions* to the field of spatial data mining:

First, we developed the general framework of density-based decompositions to describe various cluster-like structures, in particular: results produced by different clustering algorithms, patterns recognized by region growing algorithms as well as “connected” groups of objects of - in principle - arbitrary data-type satisfying certain conditions, for instance connected groups of polygons from a geographic information system. To specify a particular density-based decomposition, simply two predicates have to be specified: a neighborhood predicate *NPred* for pairs of objects which has to be symmetric and reflexive, and a predicate *MinWeight* for the minimum weight of sets of objects.

We discussed several instances of a density-based decomposition in detail - especially the results of different clustering algorithms and presented an algorithmic schema GDBSCAN to construct density-based decompositions. We indicated how GDBSCAN is implemented independently from the specific predicates for the neighborhood of objects, and the minimum weight for sets of objects. Furthermore, a performance evaluation showed that GDBSCAN can be efficiently applied to large spatial databases if *NPred*-neighborhood queries are supported by spatial access structures.

We also introduced advanced database techniques such as neighborhood indices and multiple neighborhood queries to further speed-up the run-time of GDBSCAN by large factors. Especially, we showed that an even more general algorithmic schema than GDBSCAN called *ExploreNeighborhoods* exists which can be transformed by purely syntactical means into an semantically equivalent schema called *MultipleExploreNeighborhoods*. This schema can be supported efficiently by the technique of multiple neighborhood queries and it does not only cover GDBSCAN as an instance but also a broader class of different spatial data mining algorithms.

In a separate chapter, some applications of GDBSCAN were presented in greater detail: first, an application to a 5-dimensional spectral space to create land-use

maps; second, an application to 3-dimensional protein data where we extracted concave and convex surface segments on proteins; third, an application to 2-dimensional astronomical images to detect celestial sources; fourth, an application to find interesting regions for trend detection in a geographic information system, i.e. a database of 2-dimensional polygons also having several non-spatial attributes. These applications demonstrated the use of different types of *NPred* and *Min-Weight* predicates to find clusters or cluster-like groups of objects in databases of different types.

To our best knowledge, we introduced the first *incremental* clustering algorithm, based on GDBSCAN, for mining in a dynamic environment, i.e. an environment where insertions and deletions occur. Due to the density-based nature of a density-based clustering, the insertion or deletion of an object affects the current clustering only in a small neighborhood of this object. Thus, efficient algorithms have been developed for incremental insertions and deletions to a clustering, yielding the same result as the application of GDBSCAN to the whole updated database. A cost-model and an experimental performance evaluation using a $2d$ -database as well as a WWW-log database was conducted to determine the speed-up factors, the break-even point, and to demonstrate the efficiency of the proposed algorithm .

In the last chapter, the notion of a *nested* density-based decomposition - which is simply a hierarchy of “flat” density-based decompositions - was defined. To compute a nested density-based decomposition with respect to a sequence of neighborhood predicates $NPred_1, \dots, NPred_n$ we developed the hierarchical algorithm H-GDBSCAN. The run-time of H-GDBSCAN is nearly the same as the run-time of GDBSCAN for computing a “flat” density-based decomposition with respect to the largest neighborhood because density-connected sets with respect to smaller neighborhoods are completely contained in density-connected sets with respect to larger neighborhoods. When computing a nested density-based decomposition H-GDBSCAN proceeds in principle like GDBSCAN. The difference is only that H-GDBSCAN has to obey a certain *order* in which objects are processed.

Additionally we developed a more specialized version of the hierarchical clustering algorithm H-GDBSCAN which is designed for distance-based neighborhood predicates. This algorithm called H-DBSCAN-ORDER does not produce clustering levels explicitly. It just creates an *order* of the database with respect to a maximum distance d , and stores in this order the reachability-distance and the core-distance of each object. From this cluster-order we can then extract any clustering level for distance values $\varepsilon \leq d$. However, in practice we will not extract any clustering-levels because the cluster-order itself is a powerful “stand-alone” tool for cluster analysis. By visualizing the reachability-distances a user can actually *see* the clustering structure in a database independent of the dimension of the data space.

There are several possibilities for *future research*. In our opinion, the most important tasks arise in connection with the cluster-ordering of a database. We think that, compared with other clustering methods, the cluster-ordering of a database produces the most useful information with respect to semi-automatic cluster-analysis in high dimensional spaces. To improve the applicability of the cluster-ordering technique we see the following opportunities for further research:

- For very high-dimensional spaces there exist no index structures to support the range queries needed in the algorithm H-DBSCAN-ORDER. That means that, even if we use multiple neighborhood queries, the run-time of the algorithm H-DBSCAN-ORDER is inherently $O(N^2)$. Therefore, it is impossible to apply it in its current form to a database containing several million object. Consequently, the most interesting question is whether we can modify the algorithm so that we can trade-off a limited amount of accuracy for a large gain in efficiency (e.g. by using intelligent sampling techniques).

- More sophisticated visualization techniques for the reachability-plot may be combined with visualizations of certain attribute values to offer additional insights in the clustering structure of a data set.
- For a more detailed analysis of existing clusters it may be worthwhile to extract automatically from a cluster-order “traditional” clustering information such as representatives of clusters (e.g. the local minima of the reachability-plot), cluster descriptions like the attribute ranges in different dimensions, and so forth.
- Incrementally managing a cluster-order when updates on the database occur is a further interesting challenge. Although, we have developed techniques to incrementally update a “flat” density-based decomposition it is not obvious how these ideas can be extended to a density-based cluster-ordering of a data set.

References

- [AF 96] Allard D. and Fraley C.: "*Non Parametric Maximum Likelihood Estimation of Features in Spatial Point Process Using Voronoi Tessellation*", Journal of the American Statistical Association, to appear in December 1997. [Available at <http://www.stat.washington.edu/tech.reports/tr293R.ps>].
- [AGG+ 98] Agrawal R., Gehrke J., Gunopulos D., Raghavan P.: "Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications", Proc. ACM SIGMOD'98 International Conference on Management of Data, Seattle, Washington, 1998, pp. 94-105
- [And 73] Anderberg M. R.: "*Cluster Analysis for Applications*," Academic Press, 1973.
- [AS 91] Aref W.G., and Samet H.: "Optimization Strategies for Spatial Query Processing", Proc. 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain, 1991, pp. 81-90.
- [AS 94] Agrawal R., Srikant R.: "*Fast Algorithms for Mining Association Rules*", Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile, 1994, pp. 487-499.
- [Atkis 96] ATKIS 500. 1996. Bavarian State Bureau of Topography and Geodasy, CD-Rom.

- [BA 96] Brachmann R. and Anand T., 1996: "*The Process of Knowledge Discovery in Databases: A Human Centered Approach*", in: *Advances in Knowledge Discovery and Data Mining*, AAAI Press, Menlo Park, pp.37-58.
- [BBK 98] Berchtold S., Böhm C., Kriegel H.-P.: "*The Pyramid-Technique: Towards Breaking the Curse of Dimensionality*", Proc. ACM SIGMOD'98 Int. Conf. on Management of Data, Seattle, Washington, 1998, pp. 142-153.
- [BBKK 97] Berchtold S., Böhm C., Keim D., Kriegel H.-P.: "*A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space*", ACM PODS Symposium on Principles of Database Systems, Tucson, Arizona, 1997.
- [BC 96] Berndt D. J., and Clifford J.: "*Finding Patterns in Time Series: A Dynamic Programming Approach*", in Fayyad U., Piatetsky-Shapiro G., Smyth P., Uthurusamy R. (eds.): *Advances in Knowledge Discovery and Data Mining*, AAAI Press / The MIT Press, 1996, pp. 229 - 248.
- [BH 64] Ball G. H., Hall D. J.: "*Some fundamental concepts and synthesis procedures for pattern recognition preprocessors*". International Conference on Microwaves, Circuit Theory, and Information Theory, Tokio, 1964.
- [BKK 96] Berchtold S., Keim D., Kriegel H.-P.: "*The X-Tree: An Index Structure for High-Dimensional Data*", 22nd Conf. on Very Large Databases, Bombay, India, 1996, pp. 28-39.
- [BKSS 90] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: "*The R*-tree: An Efficient and Robust Access Method for Points and Rectangles*", Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, ACM Press, New York, 1990, pp. 322-331.

- [BKSS 94] Brinkhoff, T., Kriegel, H.-P., Schneider, R., Seeger B.: “*Efficient Multi-Step Processing of Spatial Joins*”, Proc. ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, MN, ACM Press, New York, 1994, pp.197-208.
- [BKW+ 77] Bernstein F. C., Koetzle T. F., Williams G. J., Meyer E. F., Brice M. D., Rodgers J. R., Kennard O., Shimanovich T., Tasumi M.: “*The Protein Data Bank: a Computer-based Archival File for Macromolecular Structures*”. Journal of Molecular Biology 112, 1977, pp. 535 - 542.
- [BR 93] Banfield J. D., Raftery A. E.: “*Model-Based Gaussian and Non-Gaussian Clustering*”, Biometrics 49, 1993, pp. 803-821.
- [BR 96] Byers S. and Raftery A. E.: “*Nearest Neighbor Clutter Removal for Estimating Features in Spatial Point Processes*”, Technical Report No. 305, Department of Statistics, University of Washington. [Available at <http://www.stat.washington.edu/tech.reports/tr295.ps>].
- [BWH 95] Becker, R.H., White, R.L., and Helfand, D.J.: “*The FIRST Survey: Faint Images of the Radio Sky at Twenty Centimeters*”, Astrophys. J. 450:559, 1995.
- [CCG+ 95] Condon, J.J., Cotton, W.D., Greisen, E.W., Yin, Q.F., Perley, R.A., and Broderick, J.: “*The NRAO VLA Sky Survey. I. Goals, Methods, and First Results*”, 1995, URL: <http://www.cv.nrao.edu/~jcondon/nvss.html>
- [CHNW 96] Cheung D. W., Han J., Ng V. T., Wong Y.: “*Maintenance of Discovered Association Rules in Large Databases: An Incremental Technique*”, Proc. 12th Int. Conf. on Data Engineering, New Orleans, USA, 1996, pp. 106-114.

- [CHY 96] Chen M.-S., Han J. and Yu P. S.: “*Data Mining: An Overview from Database Perspective*”, IEEE Transactions on Knowledge and Data Engineering, Vol. 8, No. 6, December 1996, IEEE Computer Society Press, Los Alamitos, California, pp. 866-883.
- [CKS 88] Cheeseman P., Kellu J., Self M., Stutz J., Taylor W. and Freeman D.: “*AUTOCLASS: a Bayesian classification System*,” Proc. of the 5th Int. Conf. on Machine Learning, Ann Arbor, 1988, MI: Morgan Kaufmann, pp 54-64.
- [Con 86] Connolly M.L.: “*Measurement of protein surface shape by solid angles*”, Journal of Molecular Graphics, 4(1), 1986, pp. 3 - 6.
- [CPZ 97] Ciaccia P., Patella M., Zezula P.: “*M-tree: An Efficient Access Method for Similarity Search in Metric Spaces*”, Proc. 23rd Int. Conf. on Very Large Data Bases, Athens, Greece, 1997, pp. 426-435.
- [Chr 68] Christaller W.: “*Central Places in Southern Germany*”, (in German), Wissenschaftliche Buchgesellschaft, 1968.
- [EFKS 98] Ester M., Frommelt A., Kriegel H.-P., Sander J.: “*Algorithms for Characterization and Trend Detection in Spatial Databases*”, will appear in: Proc. 4th Int. Conf. on Knowledge Discovery and Data Mining, New York, 1998.
- [EKS 97] Ester M., Kriegel H.-P., Sander J.: “*Spatial Data Mining: A Database Approach*”, Proc. 5th Int. Symp. on Large Spatial Databases, Berlin, Germany, 1997, pp. 47-66.
- [EKS+ 98] Ester M., Kriegel H.-P., Sander J., Wimmer M. Xu X.: “*Incremental Clustering for Mining in a Data Warehousing Environment*”, will appear in: Proc. 24th Int. Conf. on Very Large Databases, New York, 1998.
- [EKSX 98] Ester M., Kriegel H.-P., Sander J., Xu X.: “*Clustering for Mining in Large Spatial Databases*”. Themenheft Data Mining, KI-Zeitschrift, 1/98, ScienTec Publishing, Bad Ems, 1998.

- [EKSX 97] Ester M., Kriegel H.-P., Sander J., Xu X.: "*Density-Connected Sets and their Application for Trend Detection in Spatial Databases*". Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining. Newport Beach, California, AAAI Press, Menlo Park, California, 1997.
- [EKSX 96] Ester M., Kriegel H.-P., Sander J., Xu X.: "*A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise*". Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. Portland, Oregon, AAAI Press, Menlo Park, California, 1996, pp. 226-231.
- [EKX 95a] Ester M., Kriegel H.-P., Xu X.: "*Knowledge Discovery in Large Spatial Databases: Focusing Techniques for Efficient Class Identification*", Proc. 4th Int. Symp. on Large Spatial Databases, Portland, ME, 1995, in: Lecture Notes in Computer Science, Vol. 951, Springer, 1995, pp.67-82.
- [EKX 95b] Ester, M., Kriegel, H.-P., Xu, X.: "*A Database Interface for Clustering in Large Spatial Databases*", Proc. 1st Int. Conf. on Knowledge Discovery and Data Mining, Montreal, Canada, AAAI Press, Menlo Park, California, 1995.
- [Eve 81] Everitt B.: "*Cluster Analysis*," London: Heinemann, 1981.
- [EW 98] Ester M., Wittmann R.: "*Incremental Generalization for Mining in a Data Warehousing Environment*", Proc. 6th Int. Conf. on Extending Database Technology, Valencia, Spain, 1998, in: Lecture Notes in Computer Science, Vol. 1377, Springer, 1998, pp. 135-152.
- [FHS 96] Fayyad, U. M., Haussler D. and Stolorz Z. 1996: "KDD for Science Data Analysis: Issues and Examples", Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. Portland, Oregon, AAAI Press, Menlo Park, California, 1996, pp. 50 -56.

- [FAAM 97] Feldman R., Aumann Y., Amir A., Mannila H.: "*Efficient Algorithms for Discovering Frequent Sets in Incremental Databases*", Proc. ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, Tucson, AZ, 1997, pp. 59-66.
- [Fis 87] Fisher D. H.: "Knowledge Acquisition via Incremental Conceptual Clustering," *Machine Learning* 2(2), 1987, pp 139-172.
- [Fis 95] Fisher D. H.: "*Iterative Optimization And Simplification Of Hierarchical Clusterings*," Proc. 1st Int. Conf. on Knowledge Discovery & Data Mining, Montreal, Canada, 1995, pp.
- [FL 95] Faloutsos C., and Lin K.: "*FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets*". Proc. ACM SIGMOD Int. Conf. on Management of Data. San Jose, CA, 1995, pp. 1 - 25.
- [FPL 91] Fisher D. H., Pazzani M. J. and Langley P.: "*Concept Formation: Knowledge and Experience in Unsupervised Learning*," Morgan Kaufmann Publishers, 1991.
- [FPS 96] Fayyad, U. M., .J., Piatetsky-Shapiro, G., Smyth, P. 1996: "*From Data Mining to Knowledge Discovery: An Overview*", in: *Advances in Knowledge Discovery and Data Mining*, AAAI Press, Menlo Park, pp.1-34.
- [FPS 96a] Fayyad, U. M., .J., Piatetsky-Shapiro, G., Smyth, P. 1996: "*to Knowledge Discovery and Data Mining: Towards a Unifying Framework*", Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. Portland, Oregon, AAAI Press, Menlo Park, California, pp. 82 -88.
- [GHJ+ 95] Gamma E., Helm R., Johnson R., Vlissides J.: "*Design Patterns: elements of reusable object-oriented software*", Addison Wesley Publishing Company, 1995.

- [Gue 94] Gueting R. H.: "*An Introduction to Spatial Database Systems*", in: The VLDB Journal, Vol. 3, No. 4, October 1994, pp.357-399.
- [Gut 84] Guttman A.: "*R-trees: A Dynamic Index Structure for Spatial Searching*", Proc. ACM SIGMOD Int. Conf. on Management of Data, 1984, pp. 47-54.
- [Har 75] Hartigan J. A.: "*Clustering Algorithms*," John Wiley & Sons, 1975.
- [HCC 93] Han J., Cai Y., Cercone N.: "*Data-driven Discovery of Quantitative Rules in Relational Databases*", IEEE Transactions on Knowledge and Data Engineering, Vol. 5, No. 1, 1993, pp. 29-40.
- [HT 93] Hattori K., and Torii Y.: "*Effective algorithms for the nearest neighbor method in the clustering problem*". Pattern Recognition, 1993, Vol 26, No. 5, pp. 741-746.
- [Hua 97] Huang, Z.: "*A Fast Clustering Algorithm to Cluster Very Large Categorical Data Sets in Data Mining*". In Proceedings of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, Tech. Report 97-07, UBC, Dept. of CS, 1997.
- [Huy 97] Huyn N.: "*Multiple-View Self-Maintenance in Data Warehousing Environments*", Proc. 23rd Int. Conf. on Very Large Data Bases, Athens, Greece, 1997, pp. 26-35.
- [JD 88] Jain A. K. and Dubes R. C.: "*Algorithms for Clustering Data*," Prentice-Hall, Inc., 1988.
- [KAH 96] Koperski K., Adhikary J. and Han J.: "*Spatial Data Mining: Progress and Challenges*", SIGMOD'96 Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'96), Montreal, Canada, June 1996, Technical Report 96-08, University of British Columbia, Vancouver, Canada.

- [KH 95] Koperski K., Han J.: “*Discovery of Spatial Association Rules in Geographic Information Databases*”, Proc. 4th Int. Symp. on Large Spatial Databases, Portland, ME, 1995, pp. 47-66.
- [KHA 96] Koperski K., Adhikary J., Han J.: “*Knowledge Discovery in Spatial Databases: Progress and Challenges*”, Proc. SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, Technical Report 96-08, University of British Columbia, Vancouver, Canada, 1996.
- [KN 96] Knorr E.M. and Ng R.T.: “*Finding Aggregate Proximity Relationships and Commonalities in Spatial Data Mining*,” IEEE trans. on knowledge and data engineering, vol. 8, no. 6, Dec. 1996, pp 884-897.
- [KR 90] Kaufman L., Rousseeuw P. J.: “*Finding Groups in Data: An Introduction to Cluster Analysis*”, John Wiley & Sons, 1990.
- [LH 92] Lu W., Han J.: “*Distance-Associated Join Indices for Spatial Range Search*”, Proc. 8th Int. Conf. on Data Engineering, Phoenix, AZ, 1992, pp. 284-292.
- [LHO 93] Lu W., Han J., Ooi B. C.: “*Discovery of General Knowledge in Large Spatial Databases*”, Proc. Far East Workshop on Geographic Information Systems, Singapore, 1993, pp. 275-289.
- [LJF 95] Lin K., Jagadish H. V., Faloutsos C.: “*The TV-Tree: An Index Structure for High-Dimensional Data*”, VLDB Journal, Vol 3, 1995, pp. 517-542.
- [Luo 95] Luotonen A.: “*The common log file format*”, <http://www.w3.org/pub/WWW/>, 1995.
- [Mac 67] MacQueen, J.: “*Some Methods for Classification and Analysis of Multivariate Observations*”, 5th Berkeley Symp. Math. Statist. Prob., Volume 1, pp.281-297.

- [MCP 93] Matheus C. J., Chan P. K., Piatetsky-Shapiro G.: “*Systems for Knowledge Discovery in Databases*”, IEEE Trans, on Knowledge and Data Engineering, Vol. 5, No. 6, 1993, pp. 903-913.
- [MJHS 96] Mombasher B., Jain N., Han E.-H., Srivastava J.: “*Web Mining: Pattern Discovery from World Wide Web Transactions*”, Technical Report 96-050, University of Minnesota, 1996.
- [MQM 97] Mumick I. S., Quass D., Mumick B. S.: “*Maintenance of Data Cubes and Summary Tables in a Warehouse*”, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 100-111.
- [MS 92] Muise R. and Smith C.: “*Nonparametric minefield detection and localization*”, Technical Report CSS-TM-591-91, Naval Surface Warfare Center, Coastal Systems Station.
- [Mur 83] Murtagh F.: “*A Survey of Recent Advances in Hierarchical Clustering Algorithms*”, The Computer Journal 26(4), 1983, pp.354-359.
- [Ng 96] Ng R. T.: “*Spatial Data Mining: Discovering Knowledge of Clusters from Maps*”, Proc. SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, Technical Report 96-08, University of British Columbia, Vancouver, Canada, 1996.
- [NH 94] Ng R. T., Han J.: “*Efficient and Effective Clustering Methods for Spatial Data Mining*”, Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile, Morgan Kaufmann Publishers, San Francisco, California, 1994, pp. 144-155.
- [NHS 84] Nievergelt, J., Hinterberger, H., and Sevcik, K. C. 1984: “*The Grid file: An Adaptable, Symmetric Multikey File Structure*”, ACM Trans. Database Systems 9(1), pp.38-71.
- [Nie 90] Niemann H. 1990: “*Pattern Analysis and Understanding*”. Springer-Verlag, Berlin.

- [PBK+ 96] Piatetsky-Shapiro G., Brachman R., Khabaza T., Kloesgen W. and Simoudis E., 1996: "*An Overview of Issues in Developing Industrial Data Mining and Knowledge Discovery Applications*", Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. Portland, Oregon, AAAI Press, Menlo Park, California, 1996, pp. 89 -95.
- [PDB 94] Protein Data Bank, "Quarterly Newsletter 70". Brookhaven National Laboratory, Upton, NY, 1994.
- [Ric 83] Richards A.J. 1983. "*Remote Sensing Digital Image Analysis. An Introduction*". Berlin: Springer Verlag.
- [Rol 73] Rohlf F. J.: "*Hierarchical clustering using the minimum spanning tree*", The Computer Journal 16, No. 1, 1973, pp. 93-95.
- [Rot 91] Rotem D.: "*Spatial Join Indices*", Proc. 7th Int. Conf. on Data Engineering, Kobe, Japan, 1991, pp. 500-509.
- [Sam 90] Samet H.: "*The Design and Analysis of Spatial Data Structures*", Addison Wesley Publishing Company, 1990.
- [Sch 91] Schreiber T.: "A Voronoi Diagram Based Adaptive K-Means-Type Clustering Algorithm for Multidimensional Weighted Data". In: Bieri H., Noltemeier H. (eds.): "Computational Geometry Methods, Algorithms and Applications", Int. Workshop on Comp. Geometry CG'91, Lecture Notes in Computer Science 553, Springer, 1991, pp. 265-275.
- [Sch 96] Schikuta, E.: "*Grid clustering: An efficient hierarchical clustering method for very large data sets*", In Proc. 13th Int. Conf. on Pattern Recognition, Vol 2, IEEE Computer Society Press, Los Alamitos, California, pp. 101-105.

- [SEKX 98] Sander J., Ester M., Kriegel H.-P., Xu X. 1998: “*Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and its Applications*”, will appear in: *Data Mining and Knowledge Discovery, An International Journal*, Kluwer Academic Publishers, Norwell, MA, 1998.
- [SFGM 93] Stonebraker M., Frew J., Gardels K., and Meredith J.: “*The SEQUOIA 2000 Storage Benchmark*”. Proc. ACM SIGMOD Int. Conf. on Management of Data. Washington, DC, 1993, pp. 2 - 11.
- [Sib 73] Sibson R.: “*SLINK: an optimally efficient algorithm for the single-link cluster method*”. *The Computer Journal* 16(1), 1973, pp.30-34.
- [Sym 81] Symons M.: “*Clustering criteria and multivariate normal mixtures*”, *Biometrics* 37, 1981, pp. 35-43.
- [Uhl 91] Uhlmann J.K.: “*Satisfying general proximity/similarity queries with metric trees*”, *Inf. Proc. Lett.*, Vol. 40, No. 4, 1991, pp. 175-179.
- [WFD 95] Weir, N., Fayyad, U.M., and Djorgovski, S.: “*Automated Star/Galaxy Classification for Digitized POSS-II*”, *Astron. J.* 109: 2401, 1995.
- [WSB 98] Weber R., Schek H.-J., Blott S.: “*A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces*”, Proc. 24th Int. Conf. on Very Large Data Bases, New York, USA, Morgan Kaufmann Publishers, San Francisco, California, 1998.
- [WYM 97] Wang W., Yang J., Muntz R.: “*STING: A Statistical Information Grid Approach to Spatial Data Mining*”, Proc. 23th Int. Conf. on Very Large Data Bases, Athens, Greece, Morgan Kaufmann Publishers, San Francisco, California, 1997, pp. 186-195.

- [XEKS 98] Xu, X., Ester, M., Kriegel, H.-P., and Sander J.: “*A Nonparametric Clustering Algorithm for Knowledge Discovery in Large Spatial Databases*”, will appear in Proc. IEEE Int. Conf. on Data Engineering, Orlando, Florida, 1998, IEEE Computer Society Press, Los Alamitos, California.
- [Zah 71] Zahn C.T.: “*Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters*”, IEEE Transactions on Computers, Vol. C-20, No. 1, 1971.
- [ZCW 94] Zepka, A.F., Cordes, J.M., and Wasserman, I.: “*Signal Detection Amid Noise with Known Statistics*”, Astrophys. J. 427: 438, 1994.
- [ZRL 96] Zhang T., Ramakrishnan R., Linvy M.: “*BIRCH: An Efficient Data Clustering Method for Very Large Databases*”. Proc. ACM SIGMOD Int. Conf. on Management of Data, ACM Press, New York, 1996, pp.103-114.

Index

A

- Absorption into a cluster 140
- Affected objects 137
- Application
 - astronomy 119
 - geography 122
 - protein data 117
 - raster data 114
 - Web- log database 133
- Association rules 131

B

- Bavaria information system 122
- BIRCH 37, 91

C

- Cases for deletions 143
- Cases for insertions 140
- Celestial sources 119
- CF-tree 37
- Changing reachability-level 177
- CLARANS 24, 90
- CLIQUE 29, 63
- Clustering extended objects 65
- Clustering Features 37
- Clustering-level 164
- Cluster-order
 - generating distance 196
 - visualization 197
- Core-distance 184

- Core-level 174
- Cost-Model for IncrementalDBSCAN 152
- Creation of a cluster 140

D

- Data mining
 - defintion 4
 - methods 4
 - spatial data mining 4, 42
- Data warehouse 130
- Database primitives forspatial data mining
95
- DBCLASD 28
- DBSCAN 30, 59, 80
- Density-based clustering 28
- Density-based decomposition 55
- Density-connected set 52
- Density-connectivity 51
- Density-reachable 50
- Directly density-reachable 49
- Distance function for sessions 150
- Distance-based neighborhood 46
- Distribution-based clustering 27

E

- Economic geography 123
- ExpandCluster 77
- ExpandClusterOrder 188
- ExploreNeighborhoods 103
- ExploreNeighborhoodsMultiple 104
- ExtractClustering 190

G

- GDBSCAN
 - algorithmic schema 76
 - implemmentation 80
 - run-time complexity 87
- Graph clustering 25
- Grid clustering 35
- Grid file 13
- Grid-based clustering 63

H

- H-DBSCAN-ORDER 187

- H-GDBSCAN 168
- Hierarchical cluster-ids 171
- Hierarchical clustering 20
 - average link 21
 - complete link 21
 - single link 21
- Hierarchical clustering layers 158

- I**
- Index-based sampling 34

- K**
- KDD
 - defintion 2
 - steps 3
- k-distance plot 68
- k-means clustering 23
- k-medoid clustering 23
- k-modes clustering 23
- k-nearest neighbor queries 12
- Knowledge Discovery in Databases. See KDD

- M**
- Merge of clusters 140
- Metric trees 17
- Minimum spanning tree 26
- MinWeight predicate 46
 - incrementally evaluable 47
 - monotonous 47
- M-tree 18, 151
- Multiple neighborhood queries 102
 - implementation 107
- MultipleExpandCluster 169
- MultipleSeeds 170
 - initialization 171
 - selection of next object 173
 - structure 170
 - update 178
- Multi-step query processing 86

- N**
- Neighborhood graph 95
- Neighborhood index 98
- Neighborhood relations 95
- Neighbors operation 100
- Nested density-based decomposition 163
- NPred-neighborhood 45

- O**
- OrderSeeds 189
 - update 189
- Overlap of clusters 56

- P**
- Partitioning clustering 22
 - density-based 28
 - distribution-based 27
 - k-means 23
 - k-medoid 23
 - k-modes 23
- Potential split of a cluster 143
- Pyramid technique 16

- R**
- R*-tree 15
- Reachability-distance 185
- Reachability-level 175
- Reachability-plot 194
- Region growing 31, 64
- Region queries 12
- Removal of a cluster 143
- R-tree 14

- S**
- Seeds 78
- Seeds for an update 139
- Segmentation of protein surfaces 117
- Simple reduction of a cluster 143
- Single scan clustering 34
- Single-link effect 58
- Single-Link level 57
- Solid angle 117
- Spatial data 12
- Spatial database system 12
- Spatial index structures 13
- Spatial trends 123
 - approximation based method 125

difference-based method	125
influence regions	125
STING	38, 61

T

Thematic maps.....	114
Triangle inequality.....	151

V

VA-file.....	18
--------------	----

W

Web access patterns.....	132
Web session	132
WWW access log databases	132

X

X-tree.....	15
-------------	----

Z

Zahn's clustering method	26
--------------------------------	----

List of Definitions

1	(Notation: neighborhood of an object)	45
2	(Notation: minimum weight of a set of objects)	46
3	(incrementally evaluable MinWeight, monotonous MinWeight)	47
4	(directly density-reachable)	49
5	(density-reachable, $>D$)	50
6	(density-connected)	51
7	(density-connected set)	52
8	(density-based decomposition)	55
9	(DBSCAN)	80
10	(neighborhood graph)	95
11	(neighborhood index)	98
12	(enhanced neighborhood $N_{2N_{Pred}(o)}$)	134
13	(affected objects)	137
14	(seed objects for the update)	139
15	(nested density-based decomposition)	163
16	(clustering-level of a nested density-based decomposition)	164
17	(core-level of an object p)	174
18	(reachability-level of an object p with respect to an object o)	175
19	(core-distance of an object p)	184
20	(reachability-distance of an object p with respect to an object o)	185

List of Figures

1	Illustration of the grid file	13
2	Illustration of the R-tree	14
3	Comparison of R-tree and X-tree structure.....	16
4	Illustration of the pyramid technique	17
5	Illustration of the M-tree	18
6	Single link clustering of $n = 9$ objects	21
7	k-means and k-medoid ($k = 3$) clustering for a sample data set	25
8	MST of a sample data set.....	27
9	Data page structure of an R*-tree for a 2d-point database.....	35
10	Data page structure of a Grid File for a 2d-point database [Sch 96]	36
11	CF-tree structure	38
12	STING structure [WYM 97].....	39
13	Clustering and spatial data mining.....	42
14	Example for a generalized clustering problem	43
15	Sample databases of 2d points	44
16	Generalization of density-based clusters	45
17	Core objects and border objects	48
18	Direct density-reachability	49
19	Density-reachability	50
20	Density-connectivity	52
21	Overlap of two clusters for $\text{MinPts} = 4$	56
22	Illustration of single link level	58
23	Illustration of DBSCAN result.....	60
24	Illustration of density-based clustering using a grid approach	64
25	Illustration of simple region growing	65
26	Illustration of clustering polygonal objects.....	66
27	Sorted 3-distance plot for sample database 3.....	68

28	Determining ϵ using the sorted 3-distance plot.....	70
29	Range for ϵ in sorted 3-distance plot yielding the same clustering	71
30	Examples of sorted 3-distance plots for different data distributions	72
31	Algorithm GDBSCAN.....	76
32	Function ExpandCluster.....	77
33	Method Seeds::update()	78
34	Software Architecture of GDBSCAN.....	81
35	Graphical user interface and parameter setting.....	85
36	Multi-step filter-refinement procedure for spatial query processing	86
37	Clusterings discovered by CLARANS	89
38	Clusterings discovered by DBSCAN.....	89
39	run-time comparison with CLARANS	91
40	Data sets for the performance comparison with BIRCH	92
41	Sample Implementation of a Neighborhood Index	99
42	Algorithm neighbors using neighborhood indices	100
43	Speed-up for the neighbors-operation using a neighborhood index	101
44	Algorithmic scheme ExploreNeighborhoods.....	103
45	Algorithmic scheme ExploreNeighborhoodsMultiple	105
46	Performance of the multiple-neighbors operation	110
47	Relation between 2d image and feature space	114
48	Visualization of the SEQUOIA 2000 raster data.....	115
49	Visualization of the clustering result for SEQUOIA 2000 raster data.	116
50	Visualization of the clustering results for protein 133DA	119
51	Visualization of the astronomy data	120
52	Clustering results for both astronomy images	121
53	Potential sources in astronomy data.....	122
54	Trend analysis in geographic geography	124
55	Comparison of theoretical and observed trends.....	125
56	Influence regions with respect to average income	126
57	Explanation of the influence region of Ingolstadt	127
58	Explanation of the influence region of Munich	127
59	: Sample WWW access log entries	132
60	Changing core object property and connection status	135
61	Affected objects in a sample database	136
62	The different cases of the insertion algorithm	141
63	“Transitive” merging of clusters A, B, C by the insertion algorithm ..	142
64	The different cases of the deletion algorithm	144
65	Speed-up factors for 2d spatial databases	154
66	Speed-up factors for the Web-log database	154
67	MaxUpdates for different relative frequencies of deletions	155

68	Example of a hierarchical clustering structure.....	159
69	Example of clusters with respect to different density parameters	160
70	Illustration of lemma 9.....	162
71	Structure of a nested density-based decomposition	164
72	Algorithm H-GDBSCAN.....	168
73	Function MultipleExpandCluster.....	169
74	Structure of MultipleSeeds	170
75	Method MultipleSeeds::init().....	172
76	Method MultipleSeeds::next()	173
77	Core-level of an object p.....	174
78	Reachability-level of objects p1, p2, p3 with respect to o	176
79	Changing reachability-level of an object p	178
80	Method MultipleSeeds::update()	179
81	Core-distance of an object p	185
82	Reachability-distance of objects p1, p2, p3 with respect to o.....	186
83	Algorithm H-DBSCAN-ORDER.....	187
84	Procedure ExpandClusterOrder	188
85	Method OrderSeeds::update()	189
86	Algorithm ExtractClustering.....	190
87	Illustration of clustering level and cluster-order	193
88	Effects of parameter settings on cluster-order	194
89	Part of the reachability-plot for a 1,024-d image data set.....	196
90	Reachability-plots for data sets having different characteristics	197

