ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/220451774

A Fast Parallel Clustering Algorithm for Large Spatial Databases

Article *in* Data Mining and Knowledge Discovery · September 1999

DOI: 10.1007/0-306-47011-X_3 · Source: DBLP

CITATIONS
240

READS
241

3 authors, including:

Xiaowei Xu
University of Arkansas at Little Rock
136 PUBLICATIONS **24,493** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Causal AI View project

All content following this page was uploaded by Xiaowei Xu on 10 June 2020.

The user has requested enhancement of the downloaded file.

Data Mining and Knowledge Discovery, 3, 263–290 (1999)
© 1999 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Fast Parallel Clustering Algorithm for Large Spatial Databases

XIAOWEI XU* Xiaowei.Xu@mchp.siemens.de
Corporate Technology, Siemens AG, Otto-Hahn-Ring 6, D-81730 München, Germany

JOCHEN JÄGER jaeger@informatik.uni-muenchen.de
HANS-PETER KRIEGEL kriegel@informatik.uni-muenchen.de
Institute for Computer Science, University of Munich, Oettingenstr. 67, D-80538 München, Germany

Editors: Yike Guo and Robert Grossman

Abstract. The clustering algorithm DBSCAN relies on a density-based notion of clusters and is designed to discover clusters of arbitrary shape as well as to distinguish noise. In this paper, we present PDBSCAN, a parallel version of this algorithm. We use the 'shared-nothing' architecture with multiple computers interconnected through a network. A fundamental component of a shared-nothing system is its distributed data structure. We introduce the dR*-tree, a distributed spatial index structure in which the data is spread among multiple computers and the indexes of the data are replicated on every computer. We implemented our method using a number of workstations connected via Ethernet (10 Mbit). A performance evaluation shows that PDBSCAN offers nearly linear speedup and has excellent scaleup and sizeup behavior.

Keywords: clustering algorithms, parallel algorithms, distributed algorithms, scalable data mining, distributed index structures, spatial databases

1. Introduction

Spatial Database Systems (SDBS) (Gueting, 1994) are database systems for the management of spatial data, i.e. point objects or spatially extended objects in a 2D or 3D space or in some high-dimensional feature space. Knowledge discovery becomes more and more important in spatial databases since increasingly large amounts of data obtained from satellite images, X-ray crystal-lography or other automatic equipment are stored in spatial databases.

Data mining is a step in the KDD process consisting of the application of data analysis and discovery algorithms that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the data (Fayyad et al., 1996). Clustering, i.e. grouping the objects of a database into meaningful subclasses, is one of the major data mining methods (Matheus et al., 1993). There has been a lot of research on clustering algorithms for decades but the application to large spatial databases introduces the following new conditions:

*This work was performed while the author was still working at the Institute for Computer Science, University of Munich.

(1) Minimal requirements of domain knowledge to determine the input parameters, because appropriate values are often not known in advance when dealing with large databases.
(2) Discovery of clusters with arbitrary shape, because the shape of clusters in spatial databases may be non-convex, spherical, drawn-out, linear, elongated, etc.
(3) Good efficiency on very large databases, i.e. on databases of significantly more than just a few thousand objects.

Ester et al. (1996) present the density-based clustering algorithm DBSCAN. For each point of a cluster, its *Eps*-neighborhood (for some given $Eps > 0$) has to contain at least a minimum number of points ($MinPts > 0$). DBSCAN meets the above requirements in the following sense: first, DBSCAN requires only two input parameters (*Eps*, *MinPts*) and supports the user in determining an appropriate value for it. Second, it discovers clusters of arbitrary shape and can distinguish noise. Third, using spatial access methods, DBSCAN is efficient even for very large spatial databases. In addition, a generalized version of DBSCAN can cluster point objects as well as spatially extended objects (Sander et al., 1998).

In this paper, we present a parallel clustering algorithm PDBSCAN which is based on DBSCAN for knowledge discovery in very large spatial databases. We use the ‘shared-nothing’ architecture which has the main advantage that it can be scaled up to hundreds and probably thousands of computers. As a data structure, we introduce the dR*-tree, a distributed spatial index structure. The main program of PDBSCAN, the master, starts a clustering slave on each available computer in the network and distributes the whole data set onto the slaves. Every slave clusters only its local data. The replicated index provides an efficient access of data, and the interference between computers is also minimized through the local access of the data. The slave-to-slave and master-to-slaves communication is implemented by message passing. The master manages the task of dynamic load balancing and merges the results produced by the slaves.

We implemented our method on a number of workstations connected via Ethernet (10 Mbit). A performance evaluation shows that PDBSCAN scales up very well and has excellent speedup and sizeup behavior. The results from this study, besides being of interest in themselves, provide a guidance for the design of parallel algorithms for other spatial data mining tasks, e.g. classification and trend detection.

This paper is organized as follows. Section 2 surveys previous efforts to parallelize other clustering algorithms. Section 3 briefly describes the algorithm DBSCAN and Section 4 presents our parallel clustering algorithm PDBSCAN. Section 5 shows experimental results and evaluates our parallel algorithm with respect to speedup, scalability, and sizeup. Section 6 lists the conclusions and highlights directions for future work.

2. Related work on parallel clustering algorithms

Several authors have previously proposed some parallel clustering algorithms. Rasmussen and Willett (1989) discuss parallel implementations of the single link clustering method on an SIMD array processor. Their parallel implementation of the SLINK algorithm does not decrease the $O(n^2)$ time required by the serial implementation, but a significant constant speedup factor is obtained. Li and Fang (1989) describe parallel partitioning clustering (the

k-means clustering algorithm) and parallel hierarchical clustering (single link clustering algorithm) on an n-node hypercube and an n-node butterfly. Their algorithms run in $O(n \log n)$ time on the hypercube and $O(n \log^2 n)$ on the butterfly. Olson (1995) has described several implementations of hierarchical clustering algorithms. His implementation of hierarchical clustering algorithm achieves $O(n)$ time on a n-node CRCW PRAM and $O(n \log n)$ time on $\frac{n}{\log n}$ node butterfly networks or trees. All these parallel clustering algorithms have the following drawbacks:

1. They assume that all objects can reside in main memory at the same time.
2. They need a large number of processors (about the size of the data set) to achieve a reasonable performance.

Both assumptions are prohibitive for very large databases with millions of objects. Therefore, database oriented parallel clustering algorithms should be developed.

Recently, (Pfitzner et al., 1998) present a parallel clustering algorithm for finding halos in N-body cosmology simulations. While overcoming the above two drawbacks, their method relies on some problem-specific knowledge and may be inappropriate for other disciplines.

In the literature, several parallel algorithms for mining association rules have been proposed recently (cf. Agrawal and Shafer, 1996; Cheung et al., 1996; Park et al., 1995). However, for many applications, especially for mining in large spatial databases, scalable parallel clustering algorithms are still in great demand.

In this section, we present a parallel clustering algorithm PDBSCAN which is based on DBSCAN for knowledge discovery in very large spatial databases. We use the shared-nothing architecture, with multiple computers interconnected through a network (Stonebraker, 1986).

3. The algorithm DBSCAN

The key idea of density-based clustering is that for each point of a cluster the neighborhood of a given radius (*Eps*) has to contain at least a minimum number of points (*MinPts*), i.e. the cardinality of the neighborhood has to exceed some threshold.

We will first give a short introduction of DBSCAN including the definitions which are required for parallel clustering. For a detailed presentation of DBSCAN see Ester et al. (1996).

Definition 1. Directly density-reachable: A point p is *directly density-reachable* from a point q w.r.t. *Eps* and *MinPts* in the set of points D if

1. $p \in N_{Eps}(q)$ ($N_{Eps}(q)$ is the subset of D contained in the *Eps*-neighborhood of q.)
2. $Card(N_{Eps}(q)) \geq MinPts$.

Definition 2. Density-reachable: A point p is *density-reachable* from a point q w.r.t. *Eps* and *MinPts* in the set of points D, denoted as $p >_D q$, if there is a chain of points $p_1, \ldots, p_n$,

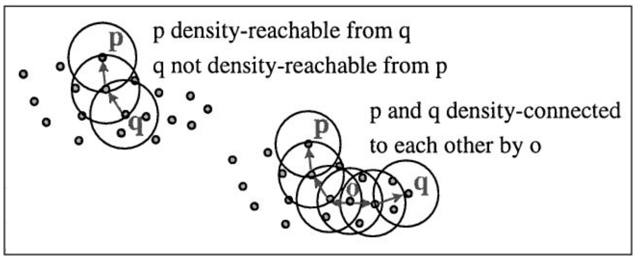

p density-reachable from q
q not density-reachable from p
p and q density-connected
to each other by o
p
q
p
o
q

Figure 1. Density-reachability and density-connectivity.

$p_1 = q$, $p_n = p$ such that $p_i \in D$ and p_{i+1} is directly density-reachable from p_i w.r.t. *Eps* and *MinPts*.

Density-reachability is a canonical extension of direct density-reachability. This relation is transitive, but not symmetric. Although it is not symmetric in general, it is obvious that the density-reachability is symmetric for points o with $Card(N_{Eps}(o)) \geq MinPts$. Two "border points" of a cluster are possibly not density-reachable from each other because there are not enough points in their *Eps*-neighborhoods. However, there must be a third point in the cluster from which both "border points" are density-reachable. Therefore, we introduce the notion of density-connectivity.

Definition 3. Density-connected: A point p is *density-connected* to a point q w.r.t. *Eps* and *MinPts* in the set of points D if there is a point $o \in D$ such that both p and q are density-reachable from o w.r.t. *Eps* and *MinPts* in D.

Density-connectivity is a symmetric relation. Figure 1 illustrates the definitions on a sample database of points from a 2-dimensional *vector* space. Note however, that the above definitions only require a distance measure and will also apply to data from a metric space.

A *cluster* is defined as a set of density-connected points which is maximal w.r.t. the density-reachability and the *noise* is the set of points not contained in any cluster.

Definition 4. Cluster: Let D be a set of points. A *cluster* C w.r.t. *Eps* and *MinPts* in D is a non-empty subset of D satisfying the following conditions:

1. Maximality: $\forall p, q \in D$: if $p \in C$ and $q >_D p$ w.r.t. *Eps* and *MinPts*, then also $q \in C$.
2. Connectivity: $\forall p, q \in C$: p is density-connected to q w.r.t. *Eps* and *MinPts* in D.

Definition 5. Noise: Let $C_1, \ldots, C_k$ be the clusters w.r.t. *Eps* and *MinPts* in D. Then, we define the *noise* as the set of points in the database D not belonging to any cluster C_i, i.e. $noise = \{p \in D \mid \forall i: p \notin C_i\}$.

We omit the term "w.r.t. *Eps* and *MinPts*" in the following whenever it is clear from the context. There are two different kinds of points in a clustering: *core points* (satisfying condition 2 of Definition 1) and *non-core points* (otherwise). In the following, we will refer to this characteristic of a point as the *core point property* of the point. The non-core points in turn are either *border points* (not a core point but density-reachable from another core point) or *noise points* (not a core point and not density-reachable from other points).

The algorithm DBSCAN was designed to discover the clusters efficiently and the noise in a database according to the above definitions. The procedure for finding a cluster is based on the fact that a cluster is uniquely determined by any of its core points:

- First, given an arbitrary point p for which the core point condition holds, the set $\{o \mid o >_D p\}$ of all points o density-reachable from p in D forms a complete cluster C and $p \in C$.
- Second, given a cluster C and an arbitrary core point $p \in C$, C in turn equals the set $\{o \mid o >_D p\}$ (cf. Lemmata 1 and 2 in Ester et al., 1996).

To find a cluster, DBSCAN starts with an arbitrary point p in D and retrieves all points of D which are density-reachable from p with respect to *Eps* and *MinPts*. If p is a core point, this procedure yields a cluster with respect to *Eps* and *MinPts*. If p is a border point, no points are density-reachable from p, and p is assigned to the noise. Then, DBSCAN visits the next point of the database D.

The retrieval of density-reachable points is performed by successive *region queries*. A *region query* returns all points intersecting a specified query region. Such queries are supported efficiently by spatial access methods such as the R*-trees (Beckmann et al., 1990).

The algorithm DBSCAN is sketched in figure 2.

```
Algorithm DBSCAN (D, Eps, MinPts)
// Precondition: All objects in D are unclassified.
 FORALL objects o in D DO:
   IF o is unclassified
     call function expand_cluster to construct a cluster wrt. Eps and MinPts containing o.

FUNCTION expand_cluster (o, D, Eps, MinPts):
 retrieve the Eps-neighborhood $N_{Eps}(o)$ of o;
 IF | $N_{Eps}(o)$ | < MinPts   // i.e. o is not a core object
     mark o as noise and RETURN;
 ELSE // i.e. o is a core object
     select a new cluster-id and mark all objects in $N_{Eps}(o)$ with this current cluster-id;
     push all objects from $N_{Eps}(o) \setminus \{o\}$ onto the stack seeds;
     WHILE NOT seeds.empty() DO
         currentObject := seeds.top();
         seeds.pop();
         retrieve the Eps-neighborhood $N_{Eps}$(currentObject) of currentObject;
         IF | $N_{Eps}$(currentObject) | ≥ MinPts
             select all objects in $N_{Eps}$(currentObject) not yet classified or marked as noise,
             push the unclassified objects onto seeds and mark all of these objects with current cluster-id;
 RETURN
```

Figure 2. Algorithm DBSCAN.

4. PDBSCAN: A fast parallel clustering algorithm

In this section, we present the parallel clustering algorithm PDBSCAN for mining in large spatial databases. We outline the proposed method in Section 4.1. The data placement strategy is crucial for the performance of the parallel algorithm. In Section 4.2, we propose an R*-tree based data placement strategy and the distributed spatial access method dR*-tree. The implementation of PDBSCAN is described in Section 4.3.

4.1. Proposed method

In this section, we outline the basic idea of our parallel clustering algorithm. We focus on the parallelization of DBSCAN for the following reasons:

1. DBSCAN is a clustering algorithm designed for knowledge discovery in spatial databases and it satisfies the requirements of discovering clusters of arbitrary shape from noisy databases as well as good efficiency on large databases.
2. The experience in the implementation of parallel DBSCAN may be directly used in other parallel clustering algorithms, e.g. DBCLASD (Xu et al., 1998), because they have the same algorithmic schema.

An overview of the hardware architecture is presented in figure 3. It consists of a number of computers (e.g. workstations or PCs) connected via a network (e.g. Ethernet). The problem is defined as follows:

Problem. Given a set of d-dimensional points $DB = \{p_1, p_2, \ldots, p_n\}$, a minimal density of clusters defined by *Eps* and *MinPts*, and a set of computers $CP = \{C_1, C_2, \ldots, C_N\}$ connected by a message passing network; find the density-based clusters with respect to the given *Eps* and *MinPts* values.

We use a partitioning strategy (Jaja, 1992) to implement parallel DBSCAN. Our method consists of three main steps. The first step is to divide the input into several partitions, and to distribute these partitions to the available computers. The second step is to cluster partitions concurrently using DBSCAN. The third step is to combine or merge the clusterings of the partitions into a clustering of the whole database. We describe this method more formally in the following:

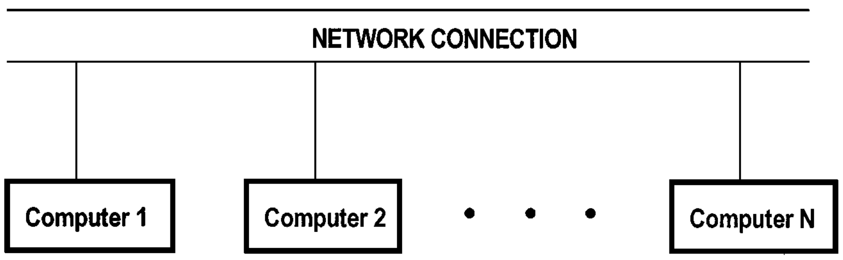


Figure 3. Proposed architecture (shared-nothing).

1. divide the input data set *DB* into N partitions $S_1, S_2, \ldots, S_N$ such that $DB = \cup_{i=1}^{N} S_i$ and $S_i \cap S_j = \emptyset$, for $i \neq j$. The partition S_i is distributed on C_i where $i = 1, 2, \ldots, N$.
2. process the N partitions concurrently using DBSCAN on the available computers C_1, $C_2, \ldots, C_N$., i.e. call algorithm DBSCAN(S_i, *Eps*, *MinPts*) concurrently on C_i for $i = 1, 2, \ldots, N$.
3. merge the clustering results obtained from the partitions $S_i, i = 1, 2, \ldots, N$, into a clustering result for *DB*.

The first step is called *data placement* in the literature (Mehta and DeWitt, 1997). In a shared-nothing environment, a proper data placement is not only crucial for the performance and scalability of the parallel algorithm, but also for its load balancing. An ideal data placement strategy for our parallel clustering algorithm should satisfy the following requirements:

1. *Load balancing*: The data should be placed such that in the second step all concurrent parallel DBSCAN(S_i, *Eps*, *MinPts*), $i = 1, 2, \ldots, N$, will be finished at the same time. Since the run-time of DBSCAN only depends on the size of the input data, the partitions should be almost of equal size if we assume that all computers have the same processing (computing and I/O) performance. If the computers have different processing performance, then we can distribute the input data on computers according to their processing performance. To simplify the description, we assume that all computers have the same processing performance in the following.
2. *Minimized communication cost*: The data should be placed such that the communication cost is minimized. To achieve this goal, each concurrent process of DBSCAN(S_i, *Eps*, *MinPts*), $i = 1, 2, \ldots, N$, should avoid accessing those data located on any of the other computers, because the access of the remote data requires some form of communication. Nearby objects should be organized on the same computer.
3. *Distributed data access*: The data should be placed such that both local and remote data can be efficiently accessed. Without any spatial access method to support efficient access to local data, the run-time of the concurrent DBSCAN in step 2 is $O(|S_i|^2)$, where $|S_i|$ is the number of objects contained in the input data set S_i, $i = 1, 2, \ldots, N$. This is also the run-time complexity of parallel DBSCAN which does not scale well for large databases. Therefore, we need a spatial access method such as the R*-tree to support efficient access to the local data in order to reduce the run-time to $O(|S_i| \log |S_i|)$. Figure 4

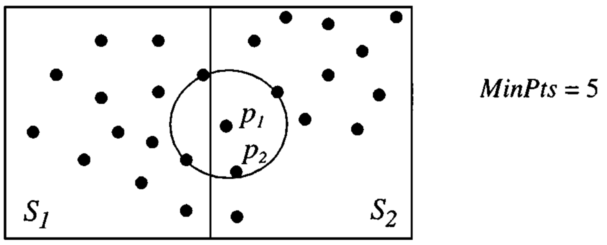


Figure 4. Illustration of the necessity of the access to remote data.

illustrates the necessity of the access to remote data: For a given *Eps*, and *MinPts* $= 5$, if there is no support of accessing remote data, then the neighborhood of object p_1 would contain only 3 points which is less than *MinPts*, and therefore p_1 would not be a core point. In this case, p_2 would not be density-reachable from any point in the partition S_2. According to the cluster definition (cf. Definition 4), p_2 would not be assigned to the cluster. Therefore, to obtain correct clustering results, a "view" over the border of partitions is necessary, i.e. the access to remote data should be supported. Of course, we have to pay communication cost for every access to remote data. This communication cost, however, can be minimized by the replication of indices which we will introduce in the following Section 4.2. On the other hand, access to remote data takes place only for the objects located on the border of two neighboring partitions. Another pay-off of remote data access is that we can efficiently merge the clustering results. We will discuss the merging issues in Section 4.3.

In the following section, we will present a new data placement method which satisfies the three requirements above. Our method, based on the R*-tree, provides not only a *spatial data placement* strategy for clustering, but also efficient access to spatial data in a shared-nothing architecture through the replication of indices. The new data placement strategy is not only useful for the parallelization of clustering, but may also be directly applied to other spatial data mining algorithms such as trend detection and classification.

4.2. Data placement and a distributed spatial access method

Data placement is an important resource management issue in the shared-nothing parallel and distributed database system. Much excellent research has been conducted on both relational databases and spatial databases. All previous work used the *declustering strategy* to place data among available computers.

Declustering exploits I/O parallelism but it also leads to higher communication cost. Declustering minimizes the query time for a single query. DBSCAN needs one range query for every object in the database, and thus we have to maximize the throughput of range queries. If we use a declustering strategy, the network may became the bottleneck. Therefore, declustering is not an optimal data placement strategy for efficient parallel clustering according to the requirements stated in Section 4.1.

According to the requirement of minimized communication cost in section 4.1, the objects that are close together in space and therefore likely to belong to the same cluster should be stored on the same computer. The goal is to reduce the communication cost and the interference between concurrent clustering operations. We call this strategy a *clustering data placement strategy.*

Given a spatial database, our first reaction would be to divide the data space into equi-sized grid cells and distribute buckets over available computers such that adjacent buckets are placed on the same computer. While this method satisfies the minimized communication cost requirement, it does not provide an efficient method for distributed data access (requirement 3 in Section 4.1).

Due to its good performance and its robustness, we use the R*-tree as our database interface to spatial data mining, as mentioned in (Ester et al., 1995). Let M be the number of directory entries that fit into a node and let m be a parameter specifying the minimum number of entries in a non-leaf node, $2 \leq m \leq \lceil M/2 \rceil$. An R*-tree satisfies the following properties:

- The root has at least two children unless it is a leaf.
- Every non-leaf node contains between m and M entries unless it is the root.
- The tree is balanced, i.e. every leaf node has the same distance from the root.
- Non-leaf nodes contain entries of the form (*ptr*, R), where *ptr* is a pointer to a child node in the R*-tree; R is the MBR (minimal bounding rectangle) that covers all rectangles in the child node.
- Leaf nodes contain entries of the form (*obj_id*, R) where *obj_id* is a pointer to the object description, and R is the MBR of the object.

These facts lead to the idea of grouping the MBRs of leaf nodes of the R*-tree into N partitions such that the nearby MBRs should be assigned to the same partition and the partitions should be almost of equal size with respect to the number of MBRs. We assign the partitions to the computers. To achieve efficient access to distributed data, the index will be replicated on all computers. We have the following three design decisions:

1. How to partition the MBRs of the leaf nodes such that nearby rectangles are in the same partition, and the size of each partition is almost the same?
2. How to distribute the partitions of rectangles onto the computers?
3. How to replicate the index among N computers?

For the first question, we propose to use space filling Hilbert curves to achieve good clustering. In a k-dimensional space, a space-filling curve starts with a path on a k-dimensional grid of side 2. The path visits every point in the grid exactly once without crossing itself. This basic curve is said to be of order 1. To derive a curve of order n, each vertex of the basic curve is replaced by the curve of order $n - 1$ which may be appropriately rotated and/or reflected. Figure 5 shows the Hilbert curves of order 1, 2 and 3 in the 2-dimensional space. The space filling curve can be generalized for higher dimensionality. An algorithm for higher dimensionality is presented by Bially (1969). The path of a space filling curve imposes a linear ordering which may be calculated by starting at one end of the curve and following the path to the other end. This ordering assigns a unique value, the *Hilbert value*, to each grid point. Figure 5 shows such an ordering. It was shown experimentally that the Hilbert curve achieves better clustering than other comparable methods (Faloutsos and Roseman, 1989). For a given R*-tree, this method works as follows. Every data page of the R*-tree is assigned to a Hilbert value according to its center of gravity. Thanks to the good clustering properties of the Hilbert curve, successive data pages will be close in space. We sort the list of pairs (Hilbert value/data page) by ascending Hilbert values. If the R*-tree has d data pages and we have n slaves, every slave obtains d/n data pages of the sorted list.

The solution to the second question is obvious: we distribute the partitions of the data pages on all available computers.

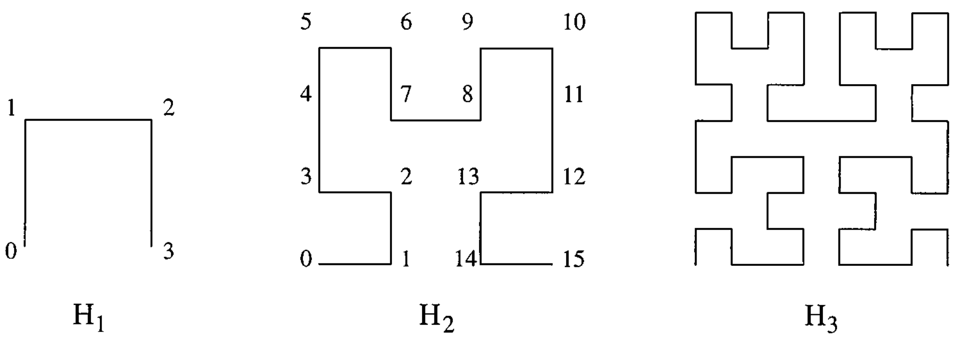


Figure 5. Hilbert curves of order 1, 2 and 3.

We propose to replicate the directory of the R*-tree on all available computers to enable efficient access to the distributed data. This replicated R*-tree is called dR*-tree which stands for distributed R*-tree. The goal is to increase the concurrency of the access. The dR*-tree has only the following structural differences from a traditional centralized R*-tree:

- the data pages are distributed on different computers
- the indices are replicated on all computers
- the pointer to the child node consists of a computer identifier *cptr* and a page identifier *ptr*, i.e. (*cptr*, *ptr*, *R*)

An example of a dR*-tree is given in figure 6. The original R*-tree has 7 data pages. These data pages are distributed onto two computers with 3 data pages on computer 1 and 4 data pages on computer 2.

The query can be started on every available computer. The query processing on a dR*-tree is very similar to the query processing on an R*-tree: a query is performed by starting at the root and computing all entries whose rectangle qualifies. For these entries, the corresponding child nodes are read into the main memory and the query process is repeated, unless the

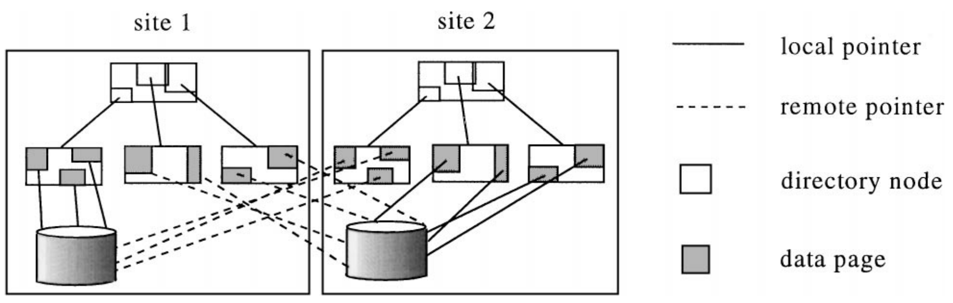


Figure 6. Distributed spatial index.

node in question is a leaf node. If the qualifying node is a local node, then this node can be read into the main memory. Otherwise, a “point to point” message must be sent to the computer where the node is stored. The remote node will be sent back by a “point to point” message.

In a central system, the run-time of a query is often measured by the number of pages accessed. In a distributed system, the run-time of a query is usually measured by the number of pages accessed and the number of messages. The following lemmata describe the performance of the dR*-tree with respect to the query processing.

Lemma 1. *A dR*-tree is a balanced tree with a height of $O(\log n)$.*

Proof: Since a dR*-tree has the same structure as the corresponding R*-tree except that the leaf nodes are distributed on different computers, a dR*-tree is balanced and the height of a dR*-tree is equal to the height of the corresponding R*-tree, i.e. $O(\log n)$. □

According to Lemma 1, a dR*-tree has the same performance as an R*-tree with respect to the number of accessed pages. In addition to the I/O cost, a dR*-tree has also communication cost but the following lemma shows:

Lemma 2. *The run-time of a query on a dR*-tree is of the same order of complexity as the corresponding R*-tree with respect to the number of accessed pages and messages.*

Proof: See Appendix. □

Although it makes no significant difference whether a query is processed by using a dR*-tree or an R*-tree according to Lemma 2, the dR*-tree enables a batch of queries to be concurrently processed on a distributed environment without interference. This advantage of the dR*-tree makes it very attractive for data mining algorithms if the goal is maximizing the throughput.

To summarize, the proposed dR*-tree meets the requirements for parallel clustering for the following reasons:

1. *Load balancing*: The number of objects (workload) on every computer is almost the same, because the number of data pages on every computer is almost the same. If the space utilization of the R*-tree is high (near 100%), the number of objects on every data page will be almost the same. 100% space utilization can be achieved by using index packing techniques (cf. Kamel and Faloutsos, 1993).
2. *Minimized communication cost*: Nearby objects are assigned to the same computer by partitioning data pages using Hilbert curves.
3. *Distributed data access*: Local and remote data can be efficiently accessed (cf. Lemma 2).

We can also use the same idea to extend other spatial access methods of the R-tree family, such as the X-tree (Berchtold et al., 1996), to distributed spatial index structures onto several computers.

4.3. Algorithm PDBSCAN

After introducing the data placement strategy and the distributed spatial access method dR*-tree, we will present the algorithm PDBSCAN in this section.

We implemented PDBSCAN using the *master-slave model* (Geist et al., 1996) in which a separate "control" program termed *master* is responsible for process spawning, initialization, collection, displaying of results, and the timing of functions. The *slave* programs perform the actual computations involved.

In our implementation, the master program of PDBSCAN spawns one slave on each available computer (site). Every slave is responsible for clustering the data stored locally on its computer and reports the result to the master. The workload is a partition of the database which is obtained by using the data placement strategy proposed in Section 4.2. Later it is also called *S*.

Figure 7 illustrates the master-slave model. Obviously, the run-time of PDBSCAN is determined by the slowest slave. Therefore, in order to maximize the throughput, load balancing between all slaves is required. We achieve good load balancing with our data placement strategy which gives each slave nearly equal workload.

SLAVE is implemented to cluster points which are stored locally by using the modified DBSCAN algorithm PartDBSCAN which we will discuss later in this section. Once the initial workload is processed, SLAVE sends the clustering result as one packet to the master.

The goal of PartDBSCAN is, to find clusters in a partition *S* (workload) of the database *DB*. PartDBSCAN uses the same density-based notion of clusters as DBSCAN. Unlike the algorithm DBSCAN, PartDBSCAN handles only the partition S instead of the whole database *DB*. Therefore, we have to adapt DBSCAN to the *space constraint*. This leads to the adaptation of the related definitions adding a space constraint.

First, we introduce the adaptation of direct reachability. In the adapted definition, point q (core point) is restricted to the partition S, because PartDBSCAN is only responsible for finding clusters in the partition S. On the other hand, to achieve a correct clustering result, PartDB-SCAN has to know for every point q in S the number of objects contained in the *Eps*-neighborhood $N_{Eps}(q)$. If q is near the border of S, the *Eps*-neighborhood $N_{Eps}(q)$ may contain objects located in adjacent partitions of S. This case is illustrated in figure 8

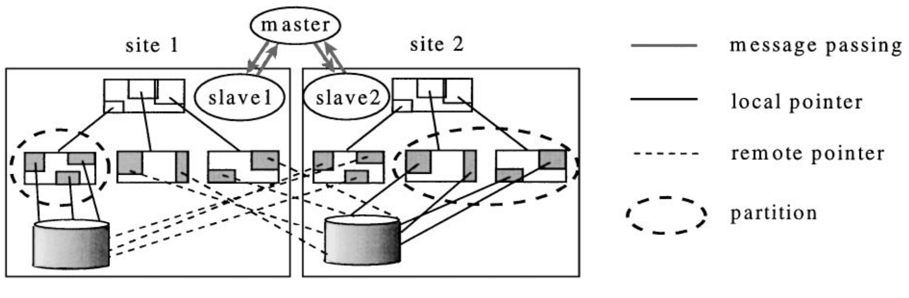


Figure 7. Master-slave model for PDBSCAN.

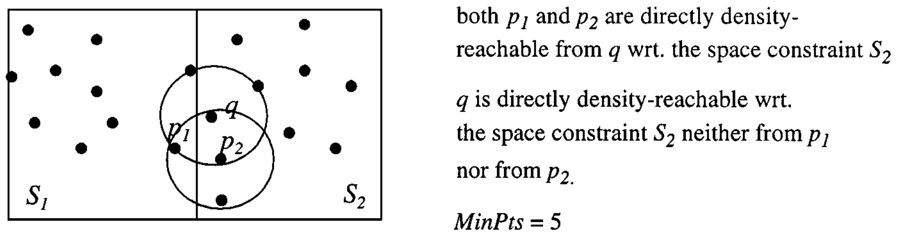

S_1
S_2
p_1
p_2
q
both p_1 and p_2 are directly density-reachable from q wrt. the space constraint S_2
q is directly density-reachable wrt. the space constraint S_2 neither from p_1 nor from p_2.
$MinPts = 5$

Figure 8. Illustration of the direct density-reachability with respect to a space constraint.

where our core point q is located in partition S_2. However, p_1 contained in $N_{Eps}(q)$ is located outside of S_2. Therefore, we adapt the definition of the direct density-reachability as follows:

Definition 6. Directly density-reachable with respect to the space constraint S:
A point p is directly density-reachable from a point q w.r.t. the space constraint S, *Eps* and *MinPts* if

1. $q \in S$,
2. $p \in N_{Eps}(q)$ and
3. $Card(N_{Eps}(q)) \geq MinPts$ (core point condition).

From Definition 6, core point q must be located in the partition S, however, point p does not necessarily reside in S; and if S is equal to the whole data space *DB*, then being directly density-reachable w.r.t. the space constraint S is equivalent to being directly density-reachable. In general, it is obvious that if a point p is directly density-reachable from a point q w.r.t. the space constraint S, *Eps* and *MinPts* where $S \subseteq T$, then p is also directly density-reachable from q w.r.t. the space constraint T, *Eps* and *MinPts*. Obviously, this direct density-reachability is symmetric for pairs of core points. In general, however, it is not symmetric if one core point and one border point are involved. Figure 8 illustrates the definition and also shows the asymmetric case.

Definition 7. Density-reachable with respect to the space constraint S: A point p is density-reachable from a point q w.r.t. the space constraint S, *Eps* and *MinPts*, denoted by $p >_S q$, if there is a chain of points $p_1, \ldots, p_n$, $p_1 = q$, $p_n = p$ such that p_{i+1} is directly density-reachable from p_i w.r.t. the space constraint S, *Eps* and *MinPts*.

In Definition 7, points $p_1 = q, \ldots, p_{n-1}$ (core points) must be located in the partition S. However, point p does not necessarily reside in S. If S is equal to the whole data space *DB*, then being density-reachable w.r.t. the space constraint S is equivalent to being density-reachable. In general, it is obvious that if a point p is density-reachable from a point q w.r.t. the space constraint S, *Eps* and *MinPts* where $S \subseteq T$, then p is also density-reachable from q w.r.t. the space constraint T, *Eps* and *MinPts*. As density-reachability is a canonical extension of direct density-reachability, the density-reachability defined here is also a canonical extension of the direct density-reachability. This relation is transitive, but it is

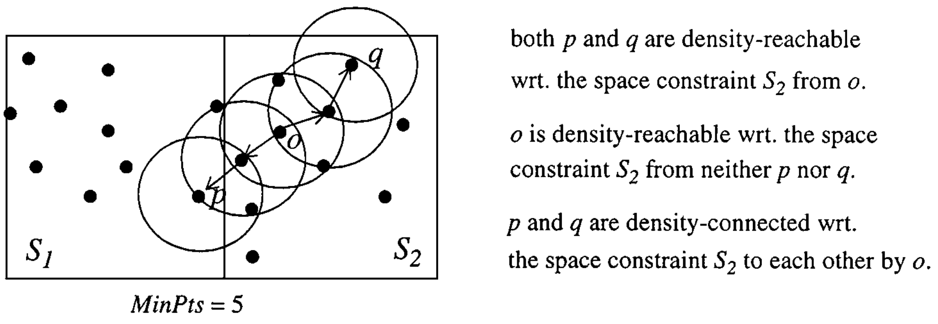

q
o
p
S_1
S_2
$MinPts = 5$
both p and q are density-reachable wrt. the space constraint S_2 from o.
o is density-reachable wrt. the space constraint S_2 from neither p nor q.
p and q are density-connected wrt. the space constraint S_2 to each other by o.

Figure 9. Density-reachability and density-connectivity w.r.t. a space constraint.

not symmetric. Figure 9 depicts the relations of some sample points and, in particular, the asymmetric case. Although it is not symmetric in general, it is obvious that the density-reachability w.r.t. the space constraints is symmetric for core points because a chain from q to p can be reversed if p is also a core point.

Similar to the density-connectivity for a cluster, we introduce the notion of density-connectivity with respect to a space constraint.

Definition 8. Density-connected with respect to the space constraint S: A point p is density-connected to a point q w.r.t. the space constraint S, *Eps* and *MinPts* if there is a point o such that both, p and q are density-reachable from o w.r.t. the space constraint S, *Eps* and *MinPts*.

In Definition 8, point o (core point) must be located in the partition S. However, the points p and q do not necessarily reside in S. If S is equal to the whole data space *DB*, then being density-connected w.r.t. the space constraint S is equivalent to being density-connected. In general, it is obvious that if a point p is density-connected to a point q w.r.t. the space constraint S_1, *Eps* and *MinPts*, where $S_1 \subseteq S_2$, then p is also density-connected to q w.r.t. the space constraint S_2, *Eps* and *MinPts*. This density-connectivity is symmetric and reflexive (cf. figure 9).

Now, we can define a density-based cluster found w.r.t. a space constraint. Similar to our previous density-based cluster definition, a density-based cluster found w.r.t. the space constraint S is defined to be the maximum set of density-connected points which are density-reachable w.r.t. the space constraint S.

Definition 9. Cluster found with respect to the space constraint S: Let *DB* be a database of points and $S \subseteq DB$. A cluster C found w.r.t. the space constraint S, *Eps* and *MinPts* in *DB* is a non-empty subset of *DB* satisfying the following conditions:

1. Maximality: $\forall p, q$: if $p \in C$ and $p >_S q$, then also $q \in C$.
2. Connectivity: $\forall p, q \in C$: p is density-connected to q w.r.t. the space constraint S, *Eps* and *MinPts*.

Note that a cluster C found w.r.t. the space constraint S, *Eps* and *MinPts* contains at least one core point for the following reasons. Since C contains at least one point p, p must be density-connected w.r.t. the space constraint S to itself via some point o (which may be equal to p). Thus, at least o has to satisfy the core point condition. Consequently, the *Eps*-neighborhood of o has at least *MinPts*. This leads to the statement that a cluster found w.r.t. the space constraint S contains at least *MinPts* points. According to the definition of a cluster (cf. Definition 4), a cluster is obviously a cluster found with respect to the space constraint *DB* (cf. Definition 9). But a cluster found with respect to the space constraint S, where $S \subset DB$, is not necessarily a cluster. (Note: Lemma 6 shows when a cluster found w.r.t. the space constraint S, where $S \subset DB$, is also a cluster in the whole data space, i.e. a cluster found wrt. the space constraint *DB*.).

The following lemmata are important for validating the correctness of our clustering algorithm PartDBSCAN, executed by the slaves. Intuitively, they state the following. Given the parameters *Eps* and *MinPts*, we can discover a cluster w.r.t. the space constraint S in a two-step approach. First, choose an arbitrary point from S satisfying the core point condition as a seed. Second, retrieve all points that are density-reachable w.r.t. the space constraint S from the seed to obtain the 'constrained' cluster containing the seed. We omit the proof, because it is an analogue to the proof of Lemma 1 in (Ester et al., 1996).

Lemma 3. *Let p be a point in S and $Card(N_{Eps}(p)) \geq MinPts$. Then the set $O = \{o \in DB \mid o >_S p\}$ is a cluster found w.r.t. the space constraint S, Eps and MinPts.*

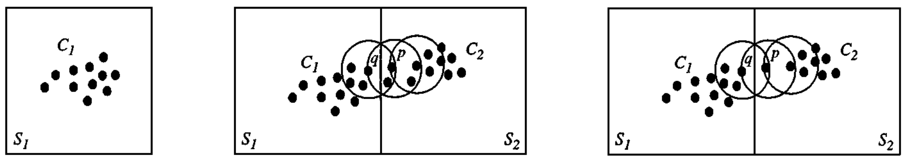
Let C be a cluster found w.r.t. the space constraint S, *Eps* and *MinPts*. C is uniquely determined by any of the core points in $C \cap S$ and, therefore, C contains exactly the points which are density-reachable w.r.t. the space constraint S from an arbitrary core point in $C \cap S$. The following lemma states the fact. We omit the proof again, because it is analogous to the proof of Lemma 2 in (Ester et al., 1996).

Lemma 4. *Let C be a cluster found w.r.t. the space constraint S, Eps and MinPts, where $S \subseteq DB$. Let p be any point in $C \cap S$ with a $Card(N_{Eps}(p)) \geq MinPts$. Then C is equal to the set $O = \{o \in DB \mid o >_S p\}$.*

We want to show that clusters (found w.r.t. the space constraint *DB*), i.e. clusters found by DBSCAN, can be obtained by merging clusters with respect to adjacent space constraints found by PartDBSCAN. Obviously, if all members of a cluster C_1 found w.r.t. the space constraint S_1 are contained in its partition S_1, i.e. $C_1 \subseteq S_1$, then C_1 is also a cluster w.r.t. the space constraint *DB*. However, according to Definition 9, C_1 may contain a point p outside of S_1, i.e. $p \in C_1 \backslash S_1$. This could take place in two cases:

1. p is a core point, i.e. $Card(N_{Eps}(p)) \geq MinPts$.
2. p is a border point, i.e. $Card(N_{Eps}(p)) < MinPts$.

If p is a core point, a cluster C_2 w.r.t. the space constraint S_2 will be generated from p according to Lemma 3, where S_2 is adjacent to S_1. In this case, C_1 and C_2 should be merged in order to achieve the clustering result for DBSCAN. If p is a border point, there will be no cluster w.r.t. the space constraint S_2 generated from p, and in this case p is kept as a member of C_1. Figure 10 illustrates our discussion using $MinPts = 4$. The following lemmata are important to validate the algorithm of merging two clusters found w.r.t. their space constraints.



(a) $C_1 \subseteq S_1$, C_1 is a cluster found wrt. the space constraint *DB*

(b) $p \in C_1 \backslash S_1$ is a core point; C_1 should be merged with C_2

(c) $p \in C_1 \backslash S_1$ is a border point; $p \in C_1$, C_1 and C_2 are NOT merged!

MinPts = 4

Figure 10. Illustration of the relationship between clusters found w.r.t. adjacent space constraints.

Lemma 5. *Let C_1 be a cluster found w.r.t. the space constraint S_1, and C_2 be a cluster found w.r.t. the space constraint S_2. If $\exists p \in C_1 \cap C_2 \cap (S_1 \cup S_2)$ such that $Card(N_{Eps}(p)) \geq MinPts$, then $C_1 \cup C_2$ is a cluster w.r.t. the space constraint $S_1 \cup S_2$, Eps and MinPts.*

Proof: (1) Maximality: let $q_1 \in C_1 \cup C_2$ and $q_2 >_{S_1 \cup S_2} q_1$. Since $q_1 >_{S_1 \cup S_2} p$ and the density-reachability w.r.t. a space constraint is transitive, it follows that $q_2 >_{S_1 \cup S_2} p$. Hence, $q_2 \in C_1 \cup C_2$. (2) Connectivity: All points in $C_1 \cup C_2$ are density-connected w.r.t. the space constraint $S_1 \cup S_2$ via point p. □

The following lemma tells us when the fusion of clusters found w.r.t. adjacent space constraints should be terminated. So for a cluster C (found w.r.t. the space constraint S) if there is no core point p belonging to C, located outside of S, then C is also a cluster w.r.t. the space constraint *DB*. This lemma is also important to validate the algorithm of PDBSCAN.

Lemma 6. *Let C be a cluster found w.r.t. the space constraint S. If $\forall p \in C \backslash S$: $Card(N_{Eps}(p)) < MinPts$, then C is a cluster w.r.t. the space constraint DB.*

Proof: See Appendix. □

PartDBSCAN is a modified DBSCAN algorithm for finding clusters w.r.t. the given space constraint S. To find such a cluster, PartDBSCAN starts with an arbitrary point p within S and retrieves all points which are density-reachable from p w.r.t. the space constraint S, *Eps* and *MinPts*. If p is a core point, this procedure yields a cluster w.r.t. the space constraint S, *Eps* and *MinPts* (see Lemmata 3 and 4). If p is not a core point, no points are density-reachable from p w.r.t. the space constraint S and PartDBSCAN visits the next point in partition S. If all members of C are contained in S, C is also a cluster (w.r.t. the space constraint *DB*) according to Lemma 6. If there are members of C outside of S, C may need to be merged with another cluster found w.r.t. an adjacent space constraint according to Lemma 5. In this case, C should be sent to the master. We call C a *merging candidate*. To achieve an efficient implementation, we apply two techniques: (1) merging candidates will be gathered during the clustering procedure and sent to the master as one packet at the end

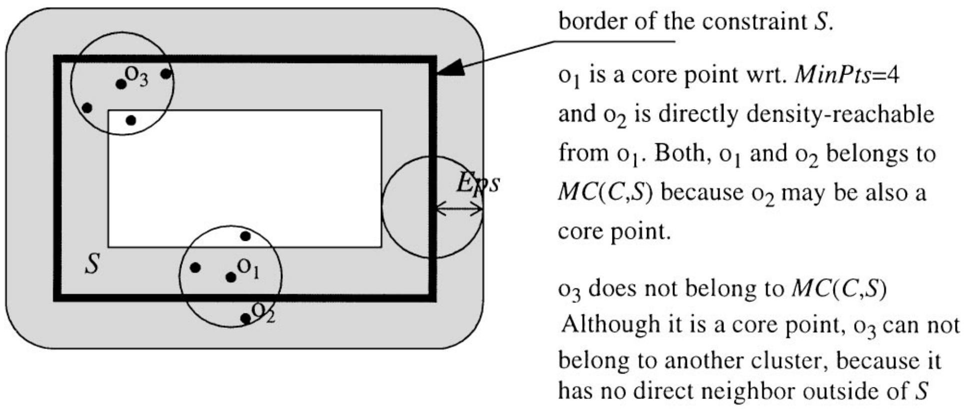

border of the constraint S.
o_1 is a core point wrt. MinPts=4 and o_2 is directly density-reachable from o_1. Both, o_1 and o_2 belongs to MC(C,S) because o_2 may be also a core point.
o_3 does not belong to MC(C,S) Although it is a core point, o_3 can not belong to another cluster, because it has no direct neighbor outside of S
o_3
o_1
o_2
S
Eps

Figure 11. Illustration of the intersecting area of two clusters.

of the clustering procedure. (2) the size of each merging candidate is reduced to a subset which consists of only points near the border of the partition S.

The gathering of merging candidates is very simple: we use a list L to store all merging candidates. If a cluster extends to a point outside of its space constraint, this cluster will be appended to the list L. At the end of the clustering procedure, list L will be sent to the master.

The reduction in size of a merging candidate C is a little more complex: according to Lemma 5, the master PDBSCAN needs to merge C with another cluster found w.r.t. an adjacent space constraint if these two clusters have a core point located in their intersection. According to the definition of a cluster found w.r.t. a space constraint (see Definition 9), each point must be density-reachable from a core point in the space constraint. Therefore, outside of S (the space constraint of C), there are no points of C having a distance greater than *Eps* from the border of S. For the same reason, inside of S, there are no points of another cluster found w.r.t. an adjacent space constraint having a distance larger than *Eps* from the border of S. Hence, the area where two clusters may intersect (*intersecting area*) is an area around the border of the constraint S with a distance to either side of the border of no more than *Eps*. Figure 11 depicts the intersecting area for C, shown as a hollow shaded rounded rectangle, and the constraint S, represented as the thick rectangle. Therefore, the merging candidate can be reduced from C to a subset of C, $MC(C, S)$.

Obviously, $MC(C, S)$ should be contained in the intersecting area and consists of all points o of C which satisfy the following condition: If o is inside of S, then o must be a core point and its *Eps*-neighborhood must contain at least one point p outside of S. If this is the case, o may also be directly density-reachable from p w.r.t. an adjacent space constraint, and may be a member of another cluster found w.r.t. an adjacent space constraint according to Definition 9. In this case, C may intersect with another cluster found w.r.t. an adjacent space constraint in o or p. Therefore, both, o and p should be contained in $MC(C, S)$. If o is outside of S, then o must be directly density-reachable from a core point q inside of S according to the cluster definition (cf. Definition 9). In this case, C may intersect with another cluster found w.r.t. an adjacent space constraint in o or q. Therefore, both, q and o should be included in $MC(C, S)$. In figure 11, points o_1 and o_2 meet the condition for $MC(C, S)$, and point o_3 does not meet the condition. To summarize, if a core point q of C which is located inside of S and its *Eps*-neighborhood contains at least one point outside of S, then both point q and all points which are located outside of S but are still directly density-reachable from q w.r.t. the space constraint S are contained in $MC(C, S)$:

$$\begin{aligned} MC(C, S) = \{o \in \{q\} \cup (N_{Eps}(q)\backslash S) \mid q \in C \cap S \wedge Card(N_{Eps}(q)) \\ \geq MinPts \wedge N_{Eps}(q)\backslash S \neq \emptyset\} \end{aligned}$$

The following lemma validates MC as reduced merging candidate:

Lemma 7. *Let C_1 and C_2 be clusters found w.r.t. the space constraint S_1 and S_2, then*

$$MC(C_1, S_1) \supseteq \{o \in C_1 \cap C_2 \cap (S_1 \cup S_2) \mid Card(N_{Eps}(o)) \geq MinPts\}$$

Proof: See Appendix. □

Lemma 7 means that $MC(C_1, S_1)$ contains all core points belonging to $C_1 \cap C_2 \cap (S_1 \cup S_2)$ for any C_2 and S_2. Therefore, according to Lemmas 5 and 7, we can use $MC(C_1, S_1)$ and $MC(C_2, S_2)$ instead of C_1 and C_2 to test whether C_1 and C_2 should be merged or not. The advantage is that the size of the merging candidates is reduced, and this makes the size of the packet to send to the master smaller and the merging algorithm more efficient.

The test when two clusters found with respect to adjacent space constraints should be merged can be further simplified: we just have to calculate the intersection $MC(C_1, S_1) \cap MC(C_2, S_2)$ and do not have to check whether there is any core point in this intersection or not. The advantage is that we can avoid the core point test and this makes the fusion of clusters even more efficient. The following lemma validates this idea.

Lemma 8. *Let C_1 be a cluster found w.r.t. the space constraint S_1 and C_2 be a cluster found w.r.t. the space constraint S_2. If $MC(C_1, S_1) \cap MC(C_2, S_2) \neq \emptyset$, then $C_1 \cup C_2$ is a cluster w.r.t. the space constraint $S_1 \cup S_2$, Eps and MinPts.*

Proof: See Appendix. □

The algorithm PartDBSCAN is sketched in figure 12. S is the partition (workload). The dR*-tree is a distributed R*-tree which provides efficient access to local and remote data. L is a list of merging candidates. The merging candidates are the subset of clusters (found w.r.t. some space constraint) that may need to be merged in order to achieve the same clustering result as DBSCAN. The merging candidates are collected and appended to L. L will be sent to PDBSCAN at the end of the clustering procedure. ExpandCluster, which is the most important function used by PartDBSCAN, is presented in figure 13.

During the expansion of the current cluster, whenever a point outside of the partition S is reached, the point and its core point are inserted into a set MC which defines the reduced

```
Algorithm: PartDBSCAN ($S$, $dR^*$-tree, $Eps$, $MinPts$)
  // S is the workload and UNCLASSIFIED
  // L is the list of merging candidates
  initialize $L$ to be empty;
  FORALL objects $o$ in $S$ DO
      IF $o$ is UNCLASSIFIED
          //construct a cluster wrt. $S$, $Eps$ and $MinPts$ containing $o$.
          IF ExpandCluster($S$, $dR^*$-tree, $o$, $ClusterId$, $Eps$, $MinPts$, $L$)
              increase $ClusterId$;
  IF $L$ NOT empty
      send $L$ to the master;
```

Figure 12. Algorithm PartDBSCAN.

```
FUNCTION: ExpandCluster($S$, $dR^*$-tree, $o$, $ClusterId$, $Eps$, $MinPts$, $L$): Boolean;
  MC.init(); // initialize the merging candidate set;
  retrieve $Eps$-neighborhood $N_{Eps}(o)$
  IF $|N_{Eps}(o)| < MinPts$        // i.e. $o$ is not a core point;
      mark $o$ as noise and RETURN false;
  ELSE    // i.e. $o$ is a core point;
      select a new $ClusterId$ and mark all objects in $N_{Eps}(o)$ with $ClusterId$;
      push all objects from $N_{Eps}(o) \backslash \{o\}$ onto the stack seeds;
      WHILE NOT seeds.empty() DO
          $currentObject$ := seeds.top();
          seeds.pop();
          IF $currentObject \in S$
              retrieve $Eps$-neighborhood $N_{Eps}(currentObject)$;
              IF $|N_{Eps}(currentObject)| \geq MinPts$
                  select all objects in $N_{Eps}(currentObject)$ not yet classified or marked as noise;
                  push the unclassified objects onto seeds and mark them with $ClusterId$;
                  IF $N_{Eps}(currentObject) \setminus S$ NOT Empty
                      insert $\{currentObject\} \cup N_{Eps}(currentObject) \setminus S$ into the set $MC$
          ELSE    // $currentObject$ is not element of $S$;
              insert $o$ and $currentObject$ into set $MC$;
      IF $MC \neq$ Empty THEN
          L.append(MC);
      RETURN True;
```

Figure 13. ExpandCluster function.

merging candidate of the current cluster (see Lemma 7). If *MC* is empty, this means that the current cluster (found w.r.t. the space constraint *S*) is also a cluster w.r.t. the space constraint *DB* (see Lemma 6). Otherwise, the current cluster may need to be merged with another cluster (see Lemmas 5 and 7). At the end of the expansion, *MC* will be appended to L. The retrieval of the *Eps*-neighborhood for a given object is performed by using a region query. In Euclidean space, e.g., this region is a circle. The center of this circle is the query object and the radius equals *Eps*. Obviously, the run-time of PartDBSCAN is $O(|S| *$ run-time of a region query): for every point in the partition *S* a region query is performed by using the dR*-tree. Since the partition *S* consists of only local points and the height of the dR*-tree is $O(\log n)$, for small *Eps* the run-time of a region query is $O(\log n)$ on the average. Hence, the run-time of PartDBSCAN is $O(|S| * \log n)$ in the average case. In general, the run-time complexity of PartDBSCAN is equivalent to DBSCAN on partition *S* w.r.t. the number of accessed pages and passed messages.

Lemma 9. *For a given partition S, PartDBSCAN based on a dR*-tree has the same order of run-time complexity as DBSCAN based on the corresponding R*-tree w.r.t. the number of accessed pages and passed messages.*

Proof: Since the run-time complexity of PartDBSCAN is $O(|S| *$ run-time of a region query on the dR*-tree) and the run-time complexity of DBSCAN is $O(|S| *$ run-time of a region query on the R*-tree), according to Lemma 2, we have proven the lemma. □

The master PDBSCAN receives a list of merging candidates from every SLAVE who has at least one merging candidate. Each merging candidate represents a cluster found with respect to a space constraint which may now need to be merged with another cluster found with respect to an adjacent space constraint. PDBSCAN collects all the lists L it receives and assigns them to a list *LL*. If *LL* is non-empty, a merging function Merging (shown in figure 14) will be executed.

The function Merging goes through the list *LL* and merges all clusters (found with respect to their different constraints) if their intersections are non-empty. Therefore, we have the following lemma.

Lemma 10. *The clustering result obtained by PDBSCAN is the same as the clustering result obtained by DBSCAN.*

Proof: It is obvious according to Lemmas 5–8. □

5. Performance evaluation

In this section, we evaluate the performance of PDBSCAN with respect to its scaleup, speedup and sizeup. For this purpose, we implemented PDBSCAN on a cluster of HP/UX workstations consisting of 8 HP C160 workstations (with a 150 MHz PA8000 CPU) interconnected by a 10 MB LAN using C++ and PVM (Parallel Virtual Machine, Geist et al., 1996). In this section, we will only evaluate the efficiency of PDBSCAN and not its accuracy,

```
Function: Merging ($LL$)
 FOR $i$ FROM 1 TO $LL.size$ DO
   $L1$ := $LL$.get($i$);
   FOR $j$ FROM 1 TO $L1.size$ DO
     $C1$ := $L1$.get($j$);
     FOR $k$ FROM $i$+1 TO $LL.size$ DO
       $L2$ := $LL$.get($k$);
       FOR $m$ FROM 1 TO $L2.size$ DO
         $C2$ := $L2$.get($m$);
         IF $C1 \cap C2 \neq \varnothing$ THEN
           $C1 := C1 \cup C2$;
           $L2$.remove($C2$);
```

Figure 14. Merging function.

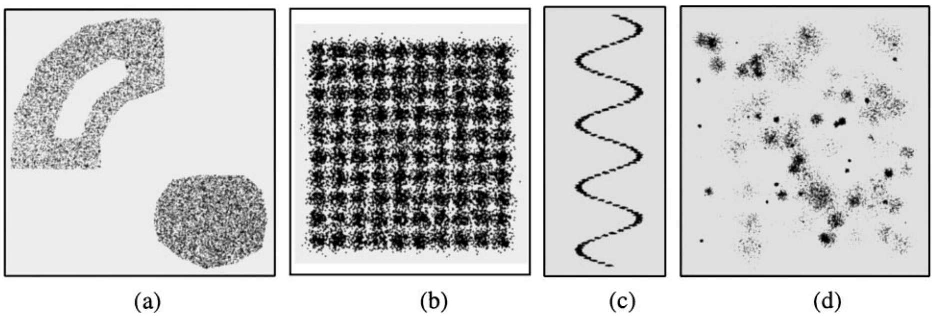


Figure 15. Synthetic data sets.

because PDBSCAN produces always the same results as DBSCAN (cf. Lemma 10). The parameters (*Eps* and *MinPts*) were chosen by using the heuristic presented in (Sander et al., 1998). A more detailed report on experimental and theoretical evaluation of PDBSCAN can also be found in (Xu, 1999).

For the following evaluations, we used both synthetic and real databases. The first synthetic data set we used is depicted in figure 15(a), which represents two clusters, each containing 10,000 points. For scaleup and sizeup experiments, we generated a series of these synthetic databases of varying size from 10,000 points with one cluster to 1 million points with 100 clusters by concatenating the original data file depicted in figure 15(a). The 1000k data set (1 million points) simply contains 50 of these two cluster sets. The databases are named according to the number of points contained. So 100k is a database containing 100,000 points.

We also used three synthetic data sets which were used for the evaluation of the BIRCH algorithm (see Zhang et al., 1998). The data sets birch1, birch2 and birch3 contain 100,000

Table 1. Synthetic and real databases.

Name	Number of points	Dimensions	Number of actual clusters	Size (in Kbytes)	Runtime for one processor (sec)
1000k	1,000,000	2	100	21,632	4199
birch1	100,000	2	100	2248	607
birch2	100,000	2	100	2248	551
birch3	100,000	2	100	2248	369
seq_534k	534,363	5	9	18,976	8916

2-dimensional points which were divided into 100 actual clusters. The structure of these data sets can be seen in figures 15 (b)–(d). For scaleup and sizeup experiments a series of data sets, beginning with 100,000 points and ending with 800,000 points, was generated. They were named birchx_y, where x denotes the data set type and y denotes the size. So birch3_500k would be the data set birch3 concatenated five times and containing 500,000 points.

As real-world data sets we used the raster data of the SEQUOIA 2000 benchmark database (Stonebraker et al., 1993), which is representative for Earth Science tasks. The raster data contains 5-dimensional points obtained from several satellite images of a region on the surface of the earth covering California. Every 1000 by 1000 meter area on the surface corresponds to a 5-dimensional vector, containing information for 5 different spectral channels. Finding clusters in such feature spaces is a common task in remote sensing digital image analysis (Richards, 1983) for the creation of thematic maps in geographic information systems. (Sander et al., 1998) describe the application of DBSCAN to the SEQUOIA data set. We used the same data set where identical points were removed (seq_534k). For scaleup and sizeup experiments, we used a series of data sets, beginning with 100,000 points and ending with 800,000 points. They were named seq_y, where y denotes the size. seq_100k corresponds to the data subset of 100,000 points of the data set sequoia. seq_500k is five times the concatenation of this subset, so it contains 500,000 points. The characteristics of the databases we used are summarized in Table 1.

We report on the results of a series of experiments described in Section 5.1. In Section 5.2, we study the I/O and communication cost factors in our experiments. The purpose was to explore further possibilities to improve the efficiency of the PDBSCAN algorithm.

5.1. *Experimental performance evaluation*

Below, we examine the speedup, scaleup and sizeup characteristics of the PDBSCAN algorithm.

To measure the speedup, we keep the database constant and increase the number of computers in the system. More formally, we apply our clustering algorithm to a given database in a system consisting of one computer (server). Then we increase the number of computers in the system from 1 to m, and cluster the database in the system with m computers. The speedup given by the larger system with m computers is measured as:

$$\textit{Speedup}(m) = \text{run-time on one computer/run-time on } m \text{ computers}$$

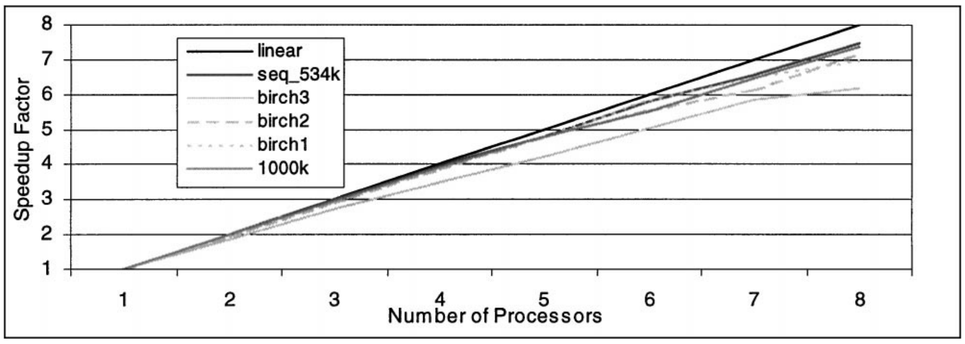


Figure 16. Speedup.

The ideal parallel algorithm demonstrates linear speedup: a system with *m* times the number of computers yields a speedup of *m*. However, linear speedup is difficult to achieve because of the communication cost and the skew of the slaves, i.e. the problem that the slowest slave determines the total time needed. If not every slave needs the same time, we have this skew problem.

We have performed the speedup evaluation on databases with quite different sizes and structures. The number of computers varied from 1 to 8. Figure 16 shows the speedup for these databases. As the graphs show, PDBSCAN has a very good speedup performance. This performance is almost the same for databases with very different sizes and shapes. Birch3 has a slightly lower speedup curve, because there are many clusters with varying density which sometimes even overlap. It is very hard to distribute the R-tree data pages to the slaves in a manner that one slave has only neighboring points. Therefore, the skew in this case is higher than in the other data sets and the total speedup is lower.

Speedup analysis holds the database size constant and grows the system. Scaleup measures the ability to grow both the system and the database size. Scaleup is defined as the ability of an *m*-times larger system to perform an *m*-times larger job in the same run-time as the original system. The scaleup metric is:

$$\begin{aligned}\mathit{Scaleup}(DB, m) = {} & \text{run-time for clustering } DB \text{ on 1 computer/run-time for}\\ & \text{clustering } m * DB \text{ on } m \text{ computer}\end{aligned}$$

To demonstrate how well the PDBSCAN algorithm handles larger databases when more computers are available, we have performed scaleup experiments where we have increased the size of the databases in direct proportion to the number of computers in the system. For the data set birch1, e.g., 100,000 points are clustered on 1 computer and 800,000 points are clustered on 8 computers. Figure 17 shows the performance results of the databases. Clearly, the PDBSCAN algorithm scales very well.

Sizeup analysis holds the number of computers in the system constant, and grows the size of the databases by the factor *m*. Sizeup measures how much longer it takes on a given system, when the database size is *m*-times larger than the original database. The sizeup

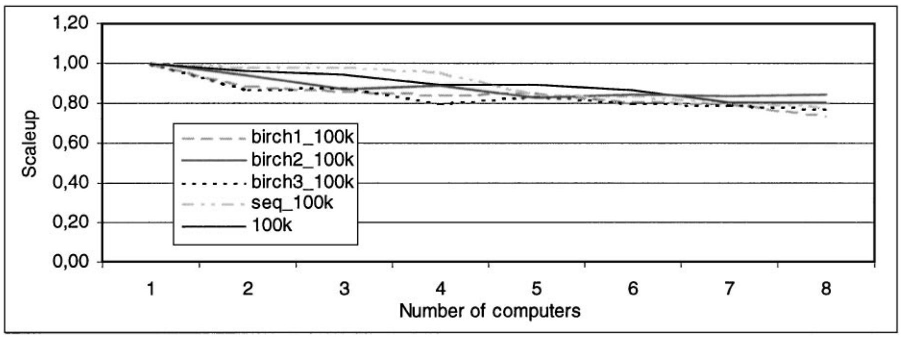


Figure 17. Scaleup.

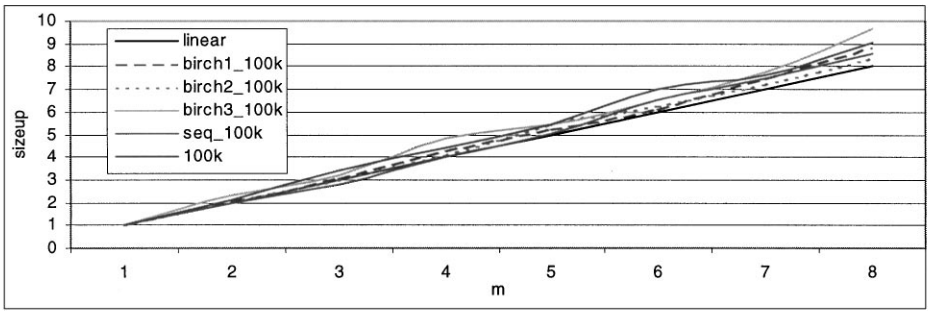


Figure 18. Sizeup.

metric is defined as follows:

$$Sizeup(DB, m) = \text{run-time for clustering } m * DB/\text{run-time for clustering } DB$$

To measure the performance of sizeup, we have fixed the number of computers to 1, 2, 4, and 8 respectively. Figure 18 shows the results on 8 computers. The graphs show that PDBSCAN has a very good sizeup performance. An 8 times larger problem needs about 8 to 10 times more time.

5.2. A Study on I/O and communication cost

In this section, we study the two main cost factors of PDBSCAN, i.e. I/O cost and communication cost. Typically, the I/O cost and the communication cost are measured by the number of pages accessed and the number of messages passed, respectively.

To analyze the impact of I/O cost on the performance of PDBSCAN, we fix the database size and grow the system by increasing its number of computers. Figure 19 shows the I/O cost as a function of the number of computers. The vertical axis is scaled logarithmically. The graphs show that PDBSCAN has a very good I/O cost performance. The I/O cost is

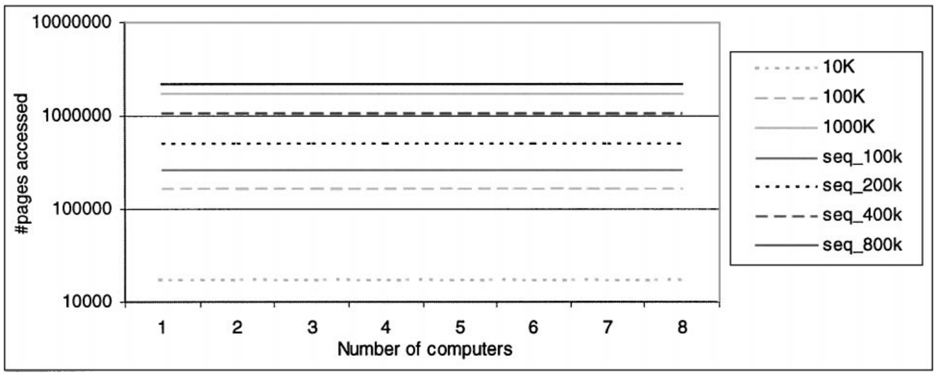


Figure 19. *I*/*O* cost.

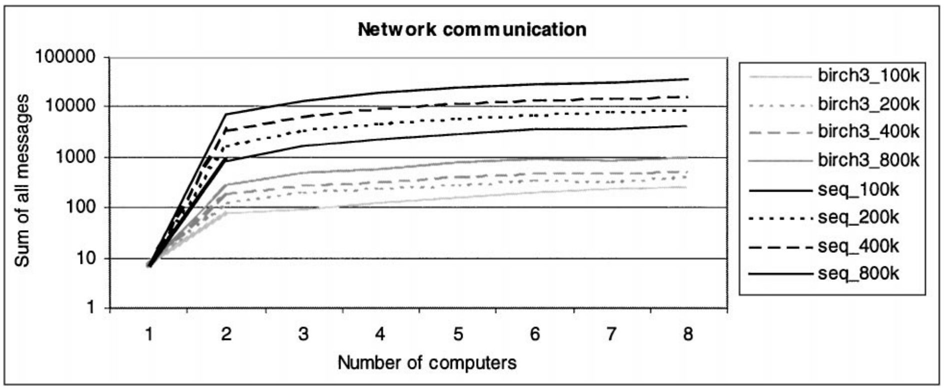


Figure 20. Communication cost.

almost constant as the number of computers in the system increases. This shows also that the dR*-tree has a good scalability w.r.t. the number of computers used.

To see the impact of the communication cost on the performance of PDBSCAN, again we fixed the database size and increased the number of computers in the system. Figure 20 shows the communication cost, i.e. the number of messages passed, as a function of the number of computers for the databases birch and SEQUOIA. The graph plots indicate that the communication cost increases linearly w.r.t. the number of computers used.

Therefore, the performance of PDBSCAN can be further improved if we use a high speed network, e.g. FDDI (100 Mbit/sec) or HiPPI (800 Mbit/sec to 1.6 Gbit/sec) instead of Ethernet (10 Mbit/sec) in our shared-nothing system.

6. Conclusions

In this paper, we proposed the parallel clustering algorithm PDBSCAN for mining in large distributed spatial databases. The main characteristics are:

1. an R*-tree based data placement strategy which minimizes the interference between slaves.
2. a distributed spatial access method dR*-tree which meets the requirements for parallel clustering with respect to load balancing, and minimizes communication cost as well as distributed data access.
3. an efficient parallel clustering algorithm PDBSCAN which is based on DBSCAN. Our theoretical analysis and the experimental evaluation show that PDBSCAN has very good performance w.r.t. speedup, scaleup and sizeup. We have also proven that the clustering result of PDBSCAN is the same as that of DBSCAN.

There are also some interesting issues for future research. This paper has shown that the distributed access method is a very powerful structure for data mining in the distributed environment. We can also use the same idea to extend other spatial access methods of the R-tree family, such as the X-tree, to distribute spatial index structures for high-dimensional data.

Other data placement strategies should also be considered and compared with our method. Furthermore, the parallelization of other spatial data mining methods should be considered in the future.

Appendix

Proof of Lemma 2: Since a dR*-tree has the same structure as the corresponding R*-tree except that the leaf nodes are distributed on different severs, a query processed using a dR*-tree accesses the same number of pages as a query using its corresponding R*-tree. We denote the number of accessed pages by *#accessed_pages*. We use *#accessed_data_pages* to denote the number of accessed data pages. It is obvious that *#accessed_data_pages* < *#accessed_pages*. Two messages are passed for each accessed data page. Therefore, the run-time *QT* of a query on a dR*-tree can be expressed as follows:

$$\begin{aligned} QT &= \text{time for } I/O + \text{ time for communication cost} \\ &= O(\#accessed\_pages) + O(2 \times \#accessed\_data\_pages) \\ &= O(\#accessed\_pages) \end{aligned}$$

Hence, the run-time of a query on a dR*-tree is of the same order of complexity as on an R*-tree with respect to the number of accessed pages and the number of messages passed. □

Proof of Lemma 6: (1) Maximality: Let $p \in C$ and let be q density-reachable from p w.r.t. the space constraint *DB*, *Eps* and *MinPts*, i.e. $q >_{DB} p$,. According to Definition 7, there is a chain of points $p_1, \ldots, p_n$, $p_1 = p$, $p_n = q$ such that p_{i+1} is directly density-reachable from p_i w.r.t. the space constraint *DB*, *Eps* and *MinPts*, and $p_i \in DB$. Since $p_i, i = 1, \ldots, n-1$ are core points, $p_i \in C$. Hence, also $q \in C$. (2) Connectivity: for $\forall p, q \in C$, since p is density-connected to q w.r.t. the space constraint S, *Eps* and *MinPts*, where $S \subseteq DB$, then p is also density-connected to q w.r.t. the space constraint *DB*, *Eps* and *MinPts*. □

Proof of Lemma 7: Let $p \in \{o \in C_1 \cap C_2 \cap (S_1 \cup S_2) \mid Card(N_{Eps}(o)) \geq MinPts\}$. It follows that $p \in (C_1 \cap C_2 \cap S_1) \cup (C_1 \cap C_2 \cap S_2)$ and $Card(N_{Eps}(p)) \geq MinPts$. Since $S_1 \cap S_2 = \emptyset$ there are only two possibilities:

(1) $p \in C_1 \cap C_2 \cap S_1$. In this case, we have $p \in C_1 \cap S_1$ and $p \in C_2$. This implies that p must be directly density-reachable from a point q w.r.t. the space constraint S_2 in C_2. It follows that $dist(p, q) \leq Eps$. Therefore, $q \in N_{Eps}(p) \backslash S_1$. This means that $N_{Eps}(p) \backslash S_1 \neq \emptyset$. To summarize, p satisfies the conditions for $MC(C_1, S_1)$: $p \in C_1 \cap S_1$, $Card(N_{Eps}(p)) \geq MinPts$ and $N_{Eps}(p) \backslash S_1 \neq \emptyset$.

(2) $p \in C_1 \cap C_2 \cap S_2$. In this case, we have $p \notin S_1$ and $p \in C_1$. It follows that p must be directly density-reachable from a point q w.r.t. the space constraint S_1 in C_1 such that $Card(N_{Eps}(q)) \geq MinPts$ and $p \in N_{Eps}(q) \backslash S_1$. To summarize, p satisfies the conditions for $MC(C_1, S_1)$: $\exists q \in C_1 \cap S_1$ such that $Card(N_{Eps}(q)) \geq MinPts$ and $p \in N_{Eps}(q) \backslash S_1$.
Therefore, $p \in MC(C_1, S_1)$. □

Proof of Lemma 8: Let $p \in MC(C_1, S_1) \cap MC(C_2, S_2)$. It follows that $p \in MC(C_1, S_1)$. According to the definition of $MC(C_1, S_1)$, p is either a core point in S_1 or a point outside of S_1. If p is a core point in S_1, then according to Lemma 5 $C_1 \cup C_2$ is a cluster w.r.t. the space constraint $S_1 \cup S_2$, *Eps* and *MinPts*. If p is outside of S_1, then p must be in S_2 and $p \in MC(C_2, S_2)$. Therefore, p will be a core point in S_2. According to Lemma 5, we have proven the lemma. □

References

Agrawal, R. and Shafer, J.C. 1996. Parallel mining of association rules: design, implementation, and experience. IBM Research Report.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. 1990. 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, pp. 322–331.

Berchtold S., Keim D.A., and Kriegel, H.-P. 1996. The X-tree: an index structure for high-dimensional data. Proc. 22nd Int. Conf. on Very Large Data Bases, Bombay, India, Morgan Kaufmann, pp. 28–39.

Bially, T. 1969. Space-filling curves: their generation and their application to bandwidth reduction. IEEE Trans. on Information Theory, IT-15(6):658–664.

Cheung, D.W., Han, J., Ng, V.T., Fu, A.W., and Fu, Y. 1996. A fast distributed algorithm for mining association rules. Proc. Int. Conf. on Parallel and Distributed Information System (PDIS'96). Miami Beach, FL, USA.

Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. Portland, OR, pp. 226–231.

Ester, M., Kriegel, H.-P., and Xu, X. 1995. A database interface for clustering in large spatial databases. Proc. 1st Int. Conf. on Knowledge Discovery and Data Mining. Montreal, Canada, 1995, pp. 94–99.

Faloutsos, C. and Roseman, S. 1989. Fractals for secondary key retrieval. Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pp. 247–252.

Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. 1996. Knowledge discovery and data mining: towards a unifying framework. Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. Portland, OR, pp. 82–88.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. 1996. PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Networked Parallel Computing. Cambridge, MA, London, England: The MIT Press, 3rd printing.

Gueting, R.H. 1994. An introduction to spatial database systems. The VLDB Journal, 3(4):357–399.

Jaja, J. 1992. An Introduction to Parallel Algorithms. Addison-Wesley Publishing Company, pp. 61–65.

Kamel, I. and Faloutsos, C. 1993. On packing R-trees. Proc. 2nd Int. Conf. on Information and Knowledge Management (CIKM).

Li, X. and Fang, Z. 1989. Parallel clustering algorithms. Parallel Computing, 11:275–290.

Matheus, C.J., Chan, P.K., and Piatetsky-Shapiro, G. 1993. Systems for knowledge discovery in databases. IEEE Transactions on Knowledge and Data Engineering, 5(6):903–913.

Mehta, M. and DeWitt, D.J. 1997. Data placement in shared-nothing parallel database systems. VLDB Journal, 6:53–72.

Olson, C.F. 1995. Parallel algorithms for hierarchical clustering. Parallel Computing, 21(8):1313–1325.

Park, J.-S., Chen, M.-S., and Yu, P.S. 1995. An effective hash based algorithm for mining association rules. Proc. ACM SIGMOD Int. Conf. on Management of Data. San Jose, CA, pp.175–186.

Pfitzner, D.W., Salmon, J.K., and Sterling, T. 1998. Halo World: Tools for Parallel Cluster Finding in Astrophysical N-body Simulations. Data Mining and Knowledge Discovery. Kluwer Academic Publishers, Vol. 2, No. 2, pp. 419–438.

Rasmussen, E.M. and Willett, P. 1989. Efficiency of hierarchical agglomerative clustering using the ICL distributed array processor. Journal of Documentation, 45(1):1–24.

Richards, A.J. 1983. Remote Sensing Digital Image Analysis. An Introduction, Berlin: Springer.

Sander, J., Ester, M., Kriegel, H.-P., and Xu, X. 1998. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. Data Mining and Knowledge Discovery, Kluwer Academic Publishers, vol. 2, pp. 1–27.

Stonebraker, M. 1986. The case for shared nothing. Database Eng., 9(1).

Stonebraker, M., Frew, J., Gardels, K., and Meredith, J. 1993. The SEQUOIA 2000 storage benchmark. Proc. ACM SIGMOD Int. Conf. on Management of Data. Washington, DC, pp. 2–11.

Xu, X. 1999. Efficient Clustering for Knowledge Discovery in Spatial Databases. Shaker, Germany: Aachen.

Xu, X., Ester, M., Kriegel, H.-P., and Sander, J. 1998. A distribution-based clustering algorithm for mining in large spatial databases. 14th Int. Conf. on Data Engineering (ICDE'98). Orlando, FL.

Zhang, T., Ramakrishnan, R., and Livny, M. 1998. BIRCH: A New Data Clustering Algorithm and Its Applications, Kluwer Academic Publishers, pp. 1–40.

Xiaowei Xu is a research scientist in the Siemens AG, Corporate Technology. His research interests are in data mining and knowledge discovery in databases, particularly in scalable data mining algorithms, parallel and distributed computing, and efficient data and index structures. He received his M.S. in 1987 from Shenyang Institute for Computing Technology, Chinese Academy of Sciences and his Ph.D. in 1998 from the University of Munich, Germany.

Jochen Jäger is a graduate student with the Institute for Computer Science at the University of Munich. His research interests include data mining, especially in biological data, parallel computing and efficient data and index structures.

Hans-Peter Kriegel is a full professor for database systems in the Institute for Computer Science at the University of Munich. His research interests are in spatial databases, particularly in query processing, performance issues, similarity search, high-dimensional indexing, and in parallel systems. Data Exploration using visualization led him to the area of knowledge discovery and data mining. Kriegel received his M.S. and Ph.D. in 1973 and 1976, respectively, from the University of Karlsruhe, Germany. Hans-Peter Kriegel has been chairman and program committee member in many international database conference. He has published over 150 refereed conference and journal papers.

View publication stats