ISSN1000-9825 Journal of Software 软件学报 2000,11(6):735~744

# FDBSCAN: A Fast DBSCAN Algorithm*

ZHOU Shui-geng ZHOU Ao-ying JIN Wen FAN Ye QIAN Wei-ning

(Department of Computer Science Fudan University Shanghai 200433)

E-mail: {sgzhou,ayzhou}@fudan.edu.cn

**Abstract** Clustering is an important application area for many fields including data mining, statistical data analysis, pattern recognition, image processing, and other business applications. Up to now, many algorithms for clustering have been developed. Contributed from the database research community, DBSCAN algorithm is an outstanding representative of clustering algorithms for its good performance in clustering spatial data. Relying on a density-based notion of clusters, DBSCAN is designed to discover clusters of arbitrary shape. It requires only one input parameter and supports the user in determining an appropriate value of it. In this paper, a fast DBSCAN algorithm (FDBSCAN) is developed which considerably speeds up the original DBSCAN algorithm. Unlike DBSCAN, FDBSCAN uses only a small number of representative points in a core point's neighborhood as seeds to expand the cluster such that the execution frequency of region query and consequently the I/O cost are reduced. Experimental results show that FDBSCAN is effective and efficient in clustering large-scale databases, and it is faster than the original DBSCAN algorithm by several times.

**Key words** Large-scale database, data mining, clustering, fast DBSCAN algorithm, representative point.

Finding useful patterns in large-scale databases has attracted considerable interest recently[1], and one of the most widely studied problems in this area is clustering, which is the task of grouping the data of a database into meaningful subclasses in such a way that minimizes the intra-differences and maximizes the inter-differences of these subclasses. There are a lot of application areas for clustering techniques, which include statistical data analysis, pattern recognition, image processing, and other business applications, to name a few. Up to now, a lot of clustering algorithms have been proposed, in which famous algorithms contributed from the database community are CLARANS[2], BIRCH[3], DBSCAN[4], CURE[5], and recently, the STING[6], CLIGUE[7] and Wave Cluster[8] algorithms. All these algorithms try to challenge the clustering problems handling huge amount of data in large-scale databases. Based on the DBSCAN algorithm[4], this paper presents a fast DBSCAN algorithm (FDBSCAN) which considerably speeds up the original DBSCAN algorithm. By selecting only a small number

* This research is supported by the National 973 Fundamental Research Program of China (国家重点基础研究计划, No. G1998030414), the National Natural Science Foundation of China (国家自然科学基金, No. 69743001), and the National Doctoral Subject Foundation of China (国家博士后项目基金, No. 1999024521). **ZHOU Shui-geng** was born in September, 1966. He is a Ph. D. candidate at Department of Computer Science, Fudan University. His current research areas include databases, data warehouses, data mining and information retrieval. **ZHOU Ao-ying** was born in May, 1965. He is a professor and doctoral supervisor of Department of Computer Science, Fudan University. His research interests are databases, knowledge bases, data mining and information retrieval. **JIN Wen** was born in August, 1967. He is a Ph. D. candidate at Department of Computer Science, Simon Fraser University, Canada. His current research areas include databases, data warehouses and data mining. **FAN Ye** was born in March, 1976. He is a graduate student of Department of Computer Science, Fudan University. His current research areas are databases and data mining. **QIAN Wei-ning** was born in December, 1976. He is a graduate student at Department of Computer Science, Fudan University. His current research areas are databases and data mining.

Manuscript received 1999-03-19, accepted 1999-06-25.

of representative points in a core point's neighborhood as seeds to expand the cluster, FDBSCAN executes less region queries than DBSCAN does, and thus reduces clustering time and I/O cost.

The rest of this paper is organized as follows. We first give an overview of major related work on clustering research contributed from the database community in Section 1, then present an introduction to the DBSCAN algorithm, and analyze its features and drawbacks while dealing with large-scale databases in Section 2. We introduce our improved DBSCAN algorithm, i.e. FDBSCAN in Section 3. Following that, some experimental results are given to demonstrate the effectiveness and efficiency of FDBSCAN algorithm in Section 4. Finally, Section 5 concludes this paper and outlines some issues for future research.

## 1 Related Work

In recent years, a number of clustering algorithms for large databases or data warehouses have been proposed. Generally, there are two types of clustering algorithms[9]: partitioning and hierarchical algorithms. Partitioning algorithms construct a partition of a database $D$ of $n$ objects into a set of $k$ clusters. The partitioning algorithms typically start with an initial partition of $D$ and then use an iterative control strategy to optimize an objective function. Each cluster is represented either by the gravity center of the cluster ($k$-means algorithms) or by one of the objects of the cluster located near its center ($k$-medoid algorithms). Ng and Han[2] introduced CLARANS which is an improved $k$-medoid method by combining a sampling procedure and the PAM algorithm. This is the first method that introduces clustering techniques into spatial data mining problems.

Hierarchical algorithms create a hierarchical decomposition of database $D$. The hierarchical decomposition is represented by a *dendrogram*, a tree that iteratively splits $D$ into smaller subsets until a termination condition is satisfied. Hierarchical algorithms do not need $k$ as an input parameter, which is obviously advantageous over partitioning algorithms. The disadvantage is that the termination condition has to be specified. BIRCH[3] uses a hierarchical data structure called CF-tree which is a height balanced tree storing the clustering features. BIRCH tries to produce the best clusters with the available resources. CURE algorithm[5] is also a hierarchical method. However, its contribution is representing a cluster with multiple representative points rather than one medoid as in traditional approaches and shrinking them toward the cluster center for alleviating the effect of outliers so that clusters of arbitrary shape can be effectively found.

Ester *et al.*[4] developed a clustering algorithm DBSCAN based on a density-based notion of clusters. It is designed to discover clusters of arbitrary shape. The key idea in DBSCAN is that for each point of a cluster, the neighborhood of a given radius has to contain at least a minimum number of points. DBSCAN can effectively handle the noise points (outliers).

Recently some new algorithms have been introduced. Wang *et al.*[6] proposed a statistical information grid-based method (STING) for spatial data mining. It divides the spatial area into rectangular cells using a hierarchical structure and stores the statistical parameters of all numerical attributes of objects within cells. CLIQUE clustering algorithm[7] identifies dense clusters in subspace of maximum dimensionality. It partitions the data space into cells. To approximate the density of the data points, it counts the number of points in each cell. The clusters are unions of connected high-density cells within a subspace. CLIQUE generates cluster description in the form of DNF expressions. G. Sheikholeslami *et al.*[8], using multi-resolution property of wavelets, proposed the WaveCluster method which partitions the data space into cells and applies wavelet transform on them. Furthermore, WaveCluster can detect arbitrary shape clusters at different degrees of detail.

## 2 About DBSCAN Algorithm

DBSCAN is a clustering algorithm which relies on a density-based notion of clusters. It is designed to dis-

cover clusters of arbitrary shape. The key idea in DBSCAN is that for each object of a cluster, the neighborhood of a given radius has to contain at least a minimum number of objects. The procedure for finding a cluster is based on the fact that a cluster is uniquely determined by any of its core objects[4].

1. Given an arbitrary object $p$ for which the core object condition holds, the set $\{o|o>_D p$ of all objects $o$ density-reachable from $p$ in $D$ forms a complete cluster $C$ and $p\in C$.

2. Given a cluster $C$ and an arbitrary core object $p\in C$, $C$ in turn equals the set $\{o|o>_D p\}$.

To find a cluster, DBSCAN starts with an arbitrary object $p$ in $D$ and retrieves all objects in $D$ density-reachable from $p$ with respect to *Eps* and Min*Pts*. If $p$ is a core object, this procedure yields a cluster with respect to *Eps* and Min*Pts*. If $p$ is a border object, no objects are density-reachable from $p$ and $p$ is assigned to *noise* temporarily. Then DBSCAN handles the next object in database $D$.

The retrieval of density-reachable objects is performed by successive region queries. A region query returns all objects intersecting a specified query region. Such queries are supported efficiently by spatial access methods such as $R^*$-trees for data from a vector space or M-trees for data from a metric space. Before clustering the database, the $R^*$-tree must be built. DBSCAN requires the user to specify the global parameter *Eps*. (The parameter Min*Pts* is fixed to 4 to reduce the computational complexity.) In order to determine *Eps*, DBSCAN has to calculate the distance between an object and its $k$th ($k=4$) nearest neighbors for all objects. Then it sorts all objects according to the previously calculated distances and plots the sorted $k$-dist graph. Then with the $k$-dist graph, the user needs to choose an appropriate $k$-dist value to get better clustering by trial and error.

DBSCAN does not perform any sort of preclustering and operates directly on the entire database. As a result, for large-scale databases, DBSCAN needs large volume of main memory support and could incur substantial I/O costs. Furthermore, during the clustering process, the number of a cluster's seed objects for expansion increases monotonously and its size is unpredictable. So large volume of main memory must be available in order to guarantee DBSCAN to run smoothly.

## 3 FDBSCAN: A Fast DBSCAN Algorithm

The average run time complexity of DBSCAN is $O(n\log n)$ ($n$ is the number of objects in the database). Most of the time for clustering process is spent on region query operations. As a matter of fact, for DBSCAN the process of clustering is an iterative procedure of executing region query. So if we can reduce the number of region query operation, DBSCAN can be speeded up. Here we propose a fast DBSCAN algorithm based on a heuristic method of reducing region query execution frequency.

DBSCAN selects a global $k$-dist value for clustering. For the thinnest clusters, the number of objects contained in their core object's neighborhood with the radius *Eps* equal to $k$-dist is $k$ (the default value of $k$ in DBSCAN is 4). However, for the other clusters, the number of objects contained in the neighborhood with the same radius is greater than $k$. DBSCAN carries out region query operation for every object contained in the core object's neighborhood. For a given core object $p$ in cluster $C$, it's conceivable that the neighborhoods of the objects contained in $p$ are most possibly intersecting with each other. Suppose $q$ is an object in $p$'s neighborhood. If its neighborhood is covered by the neighborhoods of other objects in $p$'s neighborhood, then the region query operation for $q$ can be omitted because all objects in $q$'s neighborhood can be got by the region queries of the other objects in $p$'s neighborhood, which means that $q$ is not necessary to be selected as a seed for cluster expansion. Therefore, the time consumed on region query operation for $q$ can be cut down. In fact, for the dense clusters, quite a lot of objects in a core object's neighborhood can be ignored in choosing seeds. So in order to speed up DBSCAN algorithm, we should take some representatives rather than all of the objects in $p$'s neighborhood as new seeds. We call these selected seeds *representative objects* of the neighborhood where these ob-

jects are located.

Intuitively, the outer objects in $p$'s neighborhood are favorable candidates of representative objects because the neighborhoods of inner objects tend to be covered by the neighborhoods of outer objects. Hence, selecting the representative seeds is in fact a problem of selecting representative objects which can accurately outline the shape of object distribution in a core object's neighborhood. Figure 1 illustrates an example in which $p$ is a core object in cluster $C$, $q_i$ ($i = 1 \sim 4$) are representative objects which are selected as seeds for further cluster expansion.

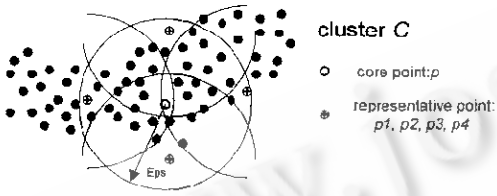


Fig. 1 Neighborhood and representative objects

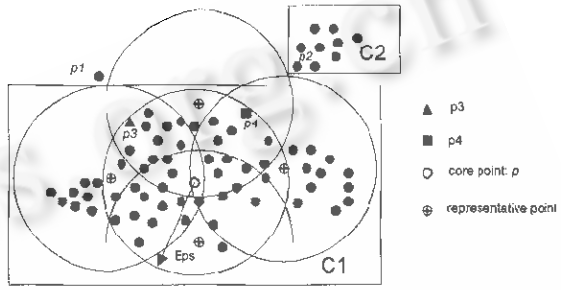


Fig. 2 Lost objects in a cluster

Theoretically, because we select only a limited and fixed number of representative objects in a core object $p$'s neighborhood as seeds for cluster expansion, it's most likely that some core objects in $p$'s neighborhood are ignored. In such a case, objects which are uniquely density-reachable from these ignored core objects will not be included in the cluster when the expansion process is completed. We call these objects *lost objects*. Certainly, they are only temporarily lost if concrete measures are taken to find them out later.

Both *core objects* and *border objects* might be lost. When the ordinary clustering process is finished, the *lost border objects* will be labeled as *noise*, and the *lost core objects* as members of certain clusters different from those clusters to which they should belong by nature. That's to say, the loss of *core objects* possibly will cause clusters to split when two adjacent parts in a cluster are density reachable from each other through *core points* that are lost by chance. Therefore, in order to obtain accurate clustering results, in addition to the ordinary DBSCAN clustering process, FDBSCAN should adopt an extra phase to cope with *lost objects*.

Figure 2 demonstrates such a situation where *lost objects* exist. In Fig. 2, $p_1$ and $p_2$ are uniquely density-reachable from $p_3$ and $p_4$ respectively. However, in the clustering process, if $C_1$ is clustered first, $p_3$ and $p_4$ are unfortunately not selected as representative points, so $p_1$ and $p_2$ are lost when $C_1$ is clustered over. Because $p_2$ is a core point, and $p_1$ is not, consequently, $p_1$ is recognized as *noise* and $p_2$ is assigned to cluster $C_2$. In the lost object handling phase, $p_1$ and $p_2$ should be found back, which means $p_1$ is re-assigned to $C_1$ and $C_2$ is merged into $C_1$.

A key problem is how many representatives should be selected for every core object's neighborhood. This parameter should not be too small and too large. If it is too small, a lot of *lost objects* may be yielded. Otherwise, the advantage of FDBSCAN can not be fully exploited. In our FDBSCAN algorithm, while 2 dimension spatial data are considered, this parameter is set to 4, which is equal to the default value of *MinPts*. The reason is, intuitively, that the neighborhood of a core object can be covered approximately by 4 well scattered representative objects' neighborhoods with the same radius *Eps*. Our experiments also show that by using 4 as the representative number, there are very few *lost objects* left after the ordinary DBSCAN clustering process is finished.

Generally, for a case of $N$-dimensional data space, $2 * N$ representative objects will be selected for cluster expansion. Further discussion about this issue is left to a future paper.

For the convenience of exposition, in the following subsections, we focus mainly on 2-dimensional space.

### 3.1 Algorithm description

FDBSCAN is a fast version of the original DBSCAN algorithm. In FDBSCAN, when the first core point is found in a new cluster, the first batch of representative points is selected as seed points for cluster expansion. And in the subsequent iterations, more representative seeds are added for cluster expansion till no more representative seed can be found, which means the cluster's expansion is finished.

Following is the outline of FDBSCAN algorithm while omitting details of data types and generation of additional information about clusters. FDBSCAN algorithm differs from DBSCAN mainly in two aspects:

— In the main program FDBSCAN(), there is an additional lost objects handling procedure HandleLostPoints();

— In procedure ExpandCluster(), procedure Representative_Seeds_Select() is added to select representative objects for cluster expansion.

```
FDBSCAN (SetofPoints, Eps, MinPts, Representative_MinPts)
// All points in SetofPoints are initialized as UNCLASSIFIED
  ClusterId:=nextId(NOISE);
  for i:=1 to SetofPoints.size do {
    Point:=SetofPoints.get(i)
    if Point.ClId=UNCLASSIFIED then {
      if ExpandCluster (SetofPoints, Point, ClusterId, Eps, MinPts,
                        Representative_MinPts) then
        ClusterId:=nextId(ClusterId)
    }
  }
  HandleLostPoints(SetofPoints, Eps, MinPts, Representative_MinPts).
ExpandCluster(SetofPoints, Point, ClusterId, Eps, MinPts, Representative_MinPts):BOOLEAN;
  candidate_seeds:=SetofPoints.regionquery(Point, Eps);
  if candidate_seeds.size<MinPts {//Point a is a border point
    SetofPoint.changeClId(Point, NOISE);
    return False;
  }
  else {        // Point is a core point
    SetofPoints.changeClId(candidate_seeds, ClId);
    Representative_Seeds_Select(candidate_seeds, representative_seeds,
                                Representative_MinPts, Point);
    while representative_seeds ≠∅ do {
      currentP:=representative_seeds.first();
      result:=SetofPoints_regionquery(currentP, Eps);
      if result.size ≥MinPts do { //currentP is a core point
        Representative_Seeds_Select(result, representative_resultP,
                                    Representative_MinPts, currentP);
        for each point p∈representative_resultP do
          if p.ClId=UNCLASSIFIED then
            representative_seeds.append(p); //add new seeds
```

```
            for each point p∈ result do
              if p.ClId=UNCLASSIFIED or NOISE then
                // label UNCLSAAIFIED or NOISE points
                SetofPoints.changeClId(p, ClId);
          }
        representative_seeds.delete(currentP); // delete the treated seed point
        }
      return True;
    }
```

### 3.2 Representative objects selection

We propose two algorithms for selecting representative seeds from a core point's neighborhood. Algorithm 1 can more accurately depict points distribution in a *core point*'s neighborhood than Algorithm 2, but is more time-consuming than the latter. Algorithm 1 iteratively selects *Representative_Minpts* well-scattered points from a core point's neighborhood. In the first iteration, the point farthest from the core point is chosen as the first representative point. In each subsequent iteration, a point from the core point's neighborhood is chosen that is farthest from the previously chosen representative points. In order to reduce execution time of Algorithm 1, we can use Manhattan distance rather than Euclidean distance. In Algorithm 2, 4 (let *Representative_Minpts* be equal to 4) points are selected as a neighborhood's representative points, which are the *leftmost*, *rightest*, *uppermost*, and *lowest* border points of the neighborhood respectively. Both algorithms are implemented in FDBSCAN. Experimental results show that FDBSCAN with Algorithm 2 is faster than that with Algorithm 1. However, their clustering results of the ordinary clustering phase (i.e. before the lost points are handled) are almost the same. In the following sections, we take Algorithm 2 as the default algorithm for representative seeds selection.

**Algorithm 1.**

```
Representative_Seeds_Select(candidate_seeds, representative_seeds,
                            Representative_Minpts, Point)
  representative_seeds:=∅
  for i:=1 to Representative_Minpts do {
      maxDist:=0;
      for each point p in candidate_seeds do {
        if i=1 then minDist:=dist(p, Point);
        else minDist:=min {dist(p,q)|q∈ representative_seeds}
        if (minDist ≥maxDist)
          maxDist:=minDist;
          maxPoint:=p;
        }
      }
  representative_seeds:= representative_seeds ∪{maxPoint};
```

**Algorithm 2.**

```
Representative Seeds Select(candidate_seeds,representative_seeds,
                            Representative_Minpts,Point)
  representative_seeds:=0;
  for i:=1 to candidate_seeds.size do {
```

```
    currentP:=candidate_seeds.get(i);
    if i=1 then {
      leftmost_point:=currentP;
      rightest_point:=currentP;
      uppermost_point:=currentP;
      lowest_point:=currentP;
    }
    if currentP.x<leftmost_point.x then leftmost_point:=currentP;
    if currentP.x>rightest_point.x then rightest_point:=currentP;
    if currentP.y<lowest_point.y then lowest_point:=currentP;
    if currentP.y>uppermost_point.y then uppermost_point:=currentP;
  }
representative_seeds:=leftmost_point∪rightest_point∪uppermost_point∪lowest_point.
```

### 3.3 About handling the *lost* points

As we have pointed out that the *lost points* are byproduct of fast expansion. When the ordinary clustering phase is over, the *lost border points* are labeled as *noise*, and the *lost core points* form new clusters with other points. The task of handling the *lost points* is, on one hand, to reassign the *lost border points* to the corresponding clusters to which these *lost border points* should belong by nature; and on the other hand, to merge the clusters where the *lost core points* are located with other clusters in which these *lost core points* should have been held.

There is no obvious difference between the *lost border points* and real noise points, so we have to examine all "noise" points to find the *lost border points*. The process is as follows. For a "noise" point, firstly, we get its neighborhood. If all points in its neighborhood are marked as noise, then it is a real noise. Otherwise, if some points are classified, we should further examine whether they are *core points*. If the answer is positive, then we assign the "noise" point to the cluster to which its nearest classified core point belongs. On the contrary, it is still a real *noise* point.

As to handling the *lost core points*, it is in fact an issue of cluster merging which is a time-consuming process. Clearly, *lost points* handling will trade off efficiency of FDBSCAN. As a matter of fact, an extra *lost points* handling procedure is not indispensable. Our strategy is to accept the reality of *lost points*' existence. The underlying reasons are as follows.

(1) That some *border points* are assigned to *noise* will not greatly affect the whole clustering quality. The *border points* are in a status between *noise* and genuine cluster members, so classifying some of them to *noise* is acceptable.

(2) The possibility of splitting a cluster due to the lost *core points* is very low. It is ordinary that two adjacent parts in a cluster are density-reachable from each other side through multiple *core points*. And it is very rare, if not impossible, that all the *core points* connecting the two parts by density are lost altogether.

(3) The selection of the number of representative points is crucial to the occurrences of *lost points*. Through selecting of an appropriate number of representative points in the *core point*'s neighborhood, the *lost points* can be controlled at a very low level.

Our experiments also show that in 2-dimensional space, by selecting 4 as the number of representative points, the ratio of lost points to genuine noise points is less than one percent. Practically and empirically, the lost points handling procedure can be ignored.

## 4 Performance Evaluation

Here we evaluate the performance of FDBSCAN and compare it with the performance of DBSCAN. We implement FDBSCAN algorithm with Borland C++ 5.0 under the software package of the original DBSCAN algorithm. All experiments have been completed on a PC with a P2 CPU (350MHz), 512M memory and a 9.6G secondary storage device. We have used both synthetic databases and the database of the SEQUOIA 2000 benchmark, which was also used in Ref. [4] for testing DBSCAN algorithm. Typical experimental results are given in Tables 1 and 2, Figs. 3 and 4 respectively.

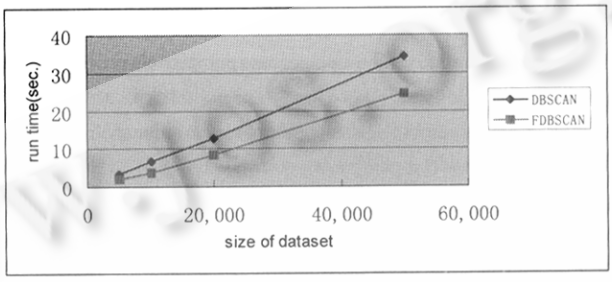


Fig. 3 Scalability with the size of dataset

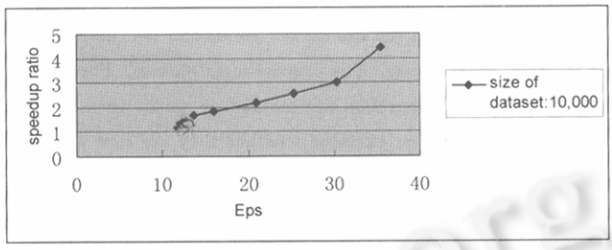


Fig. 4 Speedup ratio($t_{DBSCAN}/t_{FDBSCAN}$) vs. Eps value

Table 1 is the experimental results of the SEQUOIA 2000 benchmark database, which shows that the run time of FDBSCAN is always less than that of DBSCAN. Generally, FDBSCAN is faster than DBSCAN by several times. Table 2 presents the number of *lost points* generated by FDBSCAN with different *Eps* values, which shows that as the *Eps* value increases, the number of *noise points* found by DBSCAN rapidly enlarges, however, the number of *lost points* generated by FDBSCAN augments very slowly and it is so small compared with the total number of *noise points* that can be ignored.

Figure 3 illustrates the results of scale-up experiments with FDBSCAN and DBSCAN. We directly utilized the program in the original DBSCAN software package to build $R^*$-tree. While building $R^*$-tree, this program must load the entire dataset into memory, which restricts us from treating very large dataset. Owing to the limit

**Table 1** Run time in seconds of SEQUIOA 2000 database

| Size of dataset | 625 | 1 252 | 1 955 | 2 607 | 3 128 | 3 910 | 4 469 | 5 213 | 6 256 |
|---|---|---|---|---|---|---|---|---|---|
| DBSCAN | 0.218 | 0.453 | 0.907 | 1.734 | 2.0 | 3.593 | 3.875 | 4.672 | 6.156 |
| FDBSCAN | 0.14 | 0.328 | 0.563 | 0.875 | 1.094 | 1.64 | 1.828 | 2.149 | 2.672 |

**Table 2** Lost points produced by FDBSCAN (dataset size is 20130)

| | $Eps=6.7$ | $Eps=6.1$ | $Eps=5.6$ | $Eps=5.0$ |
|---|---|---|---|---|
| *noise points* found by DBSCAN | 223 | 275 | 401 | 728 |
| *lost points* produced by FDBSCAN | 2 | 10 | 10 | 15 |

of our PC's main memory, we can test only at most 50,000 points at one time. The curves in Fig. 3 show that FDBSCAN's scalability with dataset size is better than that of DBSCAN, which is understandable while considering the fact that the greater is the size of dataset, the more points are ignored for region query, and consequently the less is the run-time of FDBSCAN. However, Fig. 3 does not imply that both DBSCAN and FDBSCAN have linear scalability with dataset size. As the dataset size grows, the run time of both FDBSCAN and DBSCAN will increase non-linearly because they all have a run-time complexity of $O(n\log n)$.

Figure 4 is the results of experiments on a synthetic database with 10,000 points. The aim is to test the relationship between the speedup ratio of FDBSCAN over DBSCAN and *Eps* value. We define speedup ratio of FDBSCAN over DBSCAN as $t_{DBSCAN}/t_{FDBSCAN}$ where $t_{DBSCAN}$ and $t_{FDBSCAN}$ are run-time of FDBSCAN and DBSCAN for the same dataset respectively. The results indicate that speedup ratio of FDBSCAN over DBSCAN increases with Eps value, which coincides with the argument that the larger Eps is, the faster cluster expands in FDBSCAN than in DBSCAN.

## 5 Conclusions

The huge amount of information stored in databases owned by corporations (e.g., retail, financial, telecom) has spurred a tremendous interest in the area of knowledge discovery and data mining. Clustering, in data mining, is a useful technique for discovering interesting data distributions and patterns in the underlying data. As an outstanding representative of clustering algorithms, DBSCAN algorithm shows good performance in clustering spatial data. Based on the original DBSCAN algorithm, this paper presents a fast DBSCAN algorithm (FDBSCAN) which considerably speeds up the original DBSCAN algorithm. By selecting only a small number of representative points in a core point's neighborhood as seeds to expand cluster, FDBSCAN executes less region queries than DBSCAN does, which reduces clustering time and I/O cost. We performed a performance evaluation on synthetic data and real data of the SEQUOIA 2000 benchmark. The experimental results show that FDBSCAN is faster than the original DBSCAN algorithm by several times.

Future research will have to consider the following issues. Firstly, extend the FDBSCAN to high-dimensional data space. Secondly, integrate data sampling, data partitioning and parallel techniques with DBSCAN or FDBSCAN to cluster very large scale databases. Thirdly, establish an adaptive and interactive density-based clustering algorithm, which does not need the user to input any heuristic parameter.

## Acknowledgments

We would like to thank Dr. Martin Ester, the developer of DBSCAN algorithm, for his generosity in providing us with the DBSCAN software package under which we implemented the FDBSCAN algorithm.

## References

1 Chen M S, Han J H, Yu P S. Data mining: an overview from a database perspective. IEEE Transactions on Knowledge and Data Engineering, 1996,8(6):866~883

2 Ng R T, Han J. Efficient and effective clustering methods for spatial data mining. In: Jorge B Bocca, Matthias Jake, Carlo Zaniolo eds. Proceedings of the 20th VLDB Conference. San Francisco: Morgan Kaufmann, 1994. 144~155

3 Zhang T, Ramakrishnan R, Livny M. BIRCH: an efficient data clustering method for very large databases. In: Jagadish H

V, Munlick I S eds. Proceedings of the ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 1996. 103～114

4 Ester M, Kriegel H P, Sander J *et al*. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Simoudis E, Han J, Fayyad U eds. Proceedings of the 2nd International Conference on Knowledge Discovering in Databases and Data Mining (KDD-96). Massachusetts: AAAI Press, 1996. 226～232

5 Guha S, Rastogi R, Shim K. CURE: an efficient clustering algorithm for large databases. In: Haas L, Tiwary A eds. Proceedings of the ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 1998. 73～84

6 Zhang W, Yang J, Muntz R. STING: a statistical information grid approach to spatial data mining. In: Jarke M, Careg M J, Dittrich K R *et al*. eds. Proceedings of the 23rd VLDB Conference. San Francisco: Morgan Kaufmann, 1997. 186～195

7 Agrawal R, Gehrke J, Gunopulos D *et al*. Automatic subspace clustering of high dimensional data for data mining applications. In: Haas L, Tiwary A eds. Proceedings of the ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 1998. 94～105

8 Sheikholeslami G, Chatterjee S, Zhang A. WaveCluster: a multi-resolution clustering approach for very large spatial databases. In: Gupta A, Shmueli O, Widom J eds. Proceedings of the 24th VLDB Conference. San Francisco: Morgan Kaufmann, 1998. 428～439

9 Kaufman L, Rousseeuw P J. Finding groups in data: an introduction to cluster analysis. New York: John Wiley & Sons, 1990

# FDBSCAN:一种快速 DBSCAN 算法

周水庚 周傲英 金文 范晔 钱卫宁

(复旦大学计算机系 上海 200433)

**摘要** 聚类分析是一门重要的技术,在数据挖掘、统计数据分析、模式匹配和图象处理等领域具有广泛的应用前景.目前,人们已经提出了许多聚类算法.其中,DBSCAN 是一种性能优越的基于密度的空间聚类算法.利用基于密度的聚类概念,用户只需输入一个参数,DBSCAN 算法就能够发现任意形状的类,并可以有效地处理噪声.文章提出了一种加快 DBSCAN 算法的方法.新算法以核心对象邻域中所有对象的代表对象为种子对象来扩展类,从而减少区域查询次数,降低 I/O 开销.实验结果表明,FDBSCAN 能够有效地对大规模数据库进行聚类,速度上数倍于 DBSCAN.

**关键词** 大规模数据库,数据挖掘,聚类,快速 DBSCAN 算法,代表点.

**中图法分类号** TP311