AnyDBC: An Efficient Anytime Density-based Clustering Algorithm for Very Large Complex Datasets

Son T. Mai
Aarhus University
mtson@cs.au.dk

Ira Assent
Aarhus University
ira@cs.au.dk

Martin Storgaard
Aarhus University
martin@cs.au.dk

ABSTRACT

The density-based clustering algorithm DBSCAN is a state-of-the-art data clustering technique with numerous applications in many fields. However, its $O(n^2)$ time complexity still remains a severe weakness. In this paper, we propose a novel *anytime* approach to cope with this problem by reducing both the range query and the label propagation time of DBSCAN. Our algorithm, called AnyDBC, compresses the data into smaller density-connected subsets called primitive clusters and labels objects based on connected components of these primitive clusters for reducing the label propagation time. Moreover, instead of *passively* performing the range query for all objects like existing techniques, AnyDBC *iteratively* and *actively* learns the current cluster structure of the data and selects a few most promising objects for refining clusters at each iteration. Thus, in the end, it performs substantially fewer range queries compared to DBSCAN while still guaranteeing the *exact* final result of DBSCAN. Experiments show speedup factors of orders of magnitude compared to DBSCAN and its fastest variants on very large real and synthetic complex datasets.

Keywords

Density-based clustering, anytime clustering, active learning

1. INTRODUCTION

The density-based clustering algorithm DBSCAN [5] is a fundamental data clustering technique for finding arbitrary shape clusters as well as for detecting outliers. In DBSCAN, an object is dense if it has more than μ objects inside an ϵ radius. A cluster is formed as a connected set of dense objects separated from other clusters by low density regions. Since its invention, the density-based paradigm of DBSCAN has seen numerous extensions, e.g. [2, 6], as well as applications in, e.g. neuroscience [10], and astronomy [16]. However, many challenges remain, especially when facing massive complex datasets, as described below.

First, for grouping data, DBSCAN [5] performs ϵ-range queries for all objects and determines their neighbors during the cluster expansion process. The overall complexity of DBSCAN thus comes from two sources (1) the range query process with exactly n range queries, resulting in $O(\theta n^2)$ worst case complexity in general, where n is the number of objects and θ is the complexity of the distance measure and (2) the label propagation process with $O(n^2)$ time complexity for assigning labels to objects. When the volume increases, these sources quickly become a severe computational bottleneck and thus have been targets for numerous works for enhancing DBSCAN. Besides techniques that approximate the result of DBSCAN for acquiring speedups like [4], most previous exact techniques for enhancing DBSCAN generally aim at either (1) or (2) in *passive* ways, i.e., knowledge of the data is not learned and exploited to enhance the performance. For example, [2] uses fast lower-bound distances as a filter for reducing slow true distance calculations with the cost of increasing the label propagation time for maintaining the order of its seed list. Grid-based techniques, e.g. [9], divide the data space by grids, perform clustering in each cell locally and merge the results thereby saving runtime. For all these techniques, since knowledge of data is fully not utilized, they incur many redundant distance calculations thus limiting their performance. In contrast, our algorithm, called AnyDBC, introduces a *unique* approach to cope with both problems above in an *active* way. Generally, it consists of an *active learning* [15] scheme to *iteratively* study the current cluster structure of the data and to *actively* select only a small subset of objects for refining clusters at each iteration. As a result, it performs substantially fewer range queries and distance calculations compared to existing techniques, while still guaranteeing to have the *exact* final result of DBSCAN at the end. Moreover, in AnyDBC, objects are assigned into clusters via the labels of their representatives instead of directly propagating labels among them. Thus, the label propagation time is significantly reduced.

Second, most existing methods work in a *batch* scheme, i.e., they run till the end and do not allow user interaction during their execution, e.g. [2, 6]. In contrast, *anytime* algorithms [20] quickly produce an approximate result and continuously refine it during their runtime. Moreover, they allow users to suspend them for examining the result and to resume them for finding better results until a satisfactory result is reached. Due to this scheme, *anytime* algorithms have become a useful approach and are widely employed in many fields [20]. However, there are only few density-based clustering algorithms that have this useful property, e.g. [11]. Unfortunately, all of them are mainly designed to cope with small datasets due to their high time and space complexity. In contrast, AnyDBC is an *anytime* algorithm aiming at providing efficient interactive clustering scheme for very large datasets with millions of objects or more.

KDD '16, August 13-17, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-4232-2/16/08...$15.00
DOI: http://dx.doi.org/10.1145/2939672.2939750

Third, most efficient techniques for DBSCAN are built in a grid scheme that limits them to vector data and the L_p norm, e.g. [6, 7]. There are only few algorithms that are capable of handling complex data such as images, graphs or trajectories, e.g. [2, 11]. However, they all suffer from scalability problems on very large datasets. AnyDBC, on the other hand, does not follow a grid scheme. Thus, it can be applied to such complex data. Moreover, it does not require high time and space complexity data structures like [2] or special conditions such as a sequence of lower-bounding functions [12], thus, it scales well with very large complex data and arbitrary metric distance measures for these data.

Contributions. In this paper, we introduce a novel approach for clustering large datasets that addresses all the problems described above. Concretely, our algorithm, called AnyDBC, has the following advantages:

- AnyDBC *actively* learns knowledge of data and uses it to reduce both the total number of range queries as well as the label propagation time. This scheme significantly speeds up the runtime of up to orders of magnitude compared to DBSCAN and its variants.
- AnyDBC is an *anytime* algorithm which requires very small initial runtime for acquiring similar results as DBSCAN. Thus, it not only allows user interaction but also can be used to obtain good approximations under arbitrary time constraints.
- AnyDBC can be applied for any complex dataset and any metric distance measure, e.g., Edit Distance [3].

The rest of this paper is organized as follows. In Section 2, we briefly introduce the algorithm DBSCAN and some characteristics of anytime algorithms. The algorithm AnyDBC is presented in Section 3. We introduce distance measures used in our experiments in Section 4. Section 5 analyzes the performance of our algorithm. Related works are discussed in Section 6. Finally, Section 7 concludes the paper.

2. BACKGROUND

2.1 Density-based Clustering Algorithm

We assume a set of objects O with n objects, a distance function $d : O \times O \to \mathbb{R}$, and two parameters $\epsilon \in \mathbb{R}^+$ (radius) and $\mu \in \mathbb{N}^+$ (density threshold).

Definition 1. *(ϵ-neighborhood) The ϵ-neighborhood of an object $p \in O$, denoted as $N_\epsilon(p)$, is the set of objects inside an ϵ radius around p.*

$$N_\epsilon(p) = \{q \in O | d(p, q) \leq \epsilon\}$$

Definition 2. *(Core property) An object p is called a:*

1. *Core object, denoted as core(p), iff $|N_\epsilon(p)| \geq \mu$*
2. *Border object, denoted as border(p) iff $|N_\epsilon(p)| < \mu \wedge \exists q \in N_\epsilon(p) : |N_\epsilon(q)| \geq \mu$*
3. *Noise object, denoted as noise(p), else*

Definition 3. *(Density-reachability) An object q is directly density-reachable from object p, denoted as $p \triangleright q$, iff core(p) and $d(q, p) \leq \epsilon$.*

Two objects p and q are density-connected if they are density-reachable through a chain of connected core objects.

Definition 4. *(Density-connectedness) Two objects p and q are density-connected, denoted as $p \bowtie q$, iff there is a sequence of objects $(x_1, \ldots, x_m)$, where $\forall x_i$: core(x_i) and $p \triangleleft x_1 \triangleleft \cdots \triangleright x_m \triangleright q$.*

A cluster is a maximal set of density-connected objects [5].

Definition 5. *(Cluster) A subset $C \subseteq O$, which contains core and border objects, is a density-based cluster iff it satisfies two conditions:*

1. *Maximality: $\forall p \in C, \forall q \in O \setminus C : \neg p \bowtie q$*
2. *Connectivity: $\forall p, q \in C : p \bowtie q$*

For constructing clusters, DBSCAN randomly draws an unlabeled object p and performs the ϵ-range query on p. If p is a core object, the ϵ-range query is executed for all $q \in N_\epsilon(p)$ to expand the cluster until there is no core object found. Then, p and all of its density-connected objects are assigned a cluster label. And, the next unlabeled objects are processed for expanding new clusters.

2.2 Anytime Algorithms

Anytime algorithms provide an effective way to cope with time consuming problems in many fields, e.g. robotics [20], and object recognition [8]. Their general idea is quickly producing an approximate result and continuously refining it until a satisfactory solution is reached or it is terminated by the user. While running, they can be interrupted for examining intermediate results and resumed for finding better solutions at any time. Figure 1 shows the progress of different anytime algorithms. Among three algorithms A, B, and C, C obviously has the worst performance, while it is harder to evaluate the performances of A and B. However, A is preferred by many other works, e.g. [20], since it has better quality improvement in the beginning.

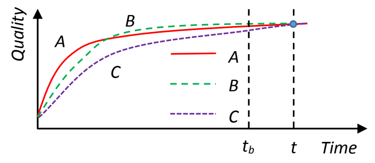


Figure 1: Common progress of anytime algorithms. Generally, the quality of results increases over time. At time $t > t_b$ the results of anytime algorithms are equal to those of the batch ones, where t_b is the runtime of the batch algorithm

3. OUR PROPOSED ALGORITHM

We illustrate the general idea of our algorithm in Section 3.1. In Section 3.2, we present the algorithms and theories behind AnyDBC. Section 3.3 summarizes the algorithm AnyDBC and its characteristics.

3.1 General idea

Figure 2 illustrates the general idea of our algorithm. Assume that we have already performed the range query on objects a, b, c, d, and e. As we can see, their neighborhoods almost determine the final clustering result of DBSCAN with two clusters C_1 and C_2. Here, a, b, and their neighbors form two small clusters inside the cluster C_1, d, e, and their neighbors form two other small clusters inside the cluster C_2, and c is an outlier. Now, if we select the next two objects, say f and g, for querying and updating the current cluster structure, a and b are thus density-connected together since f is a core object. At this time, cluster C_1 is completely determined. Though g is a border object, it allows to determine that h is a core object without having to perform a range query on h since h has at least μ neighbors. Similarly, cluster C_2 is now determined due to the density-connectedness

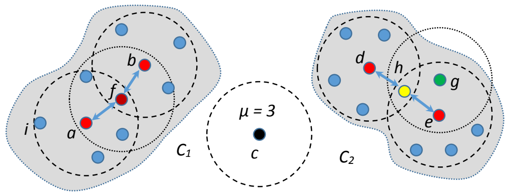


Figure 2: Basic idea of our algorithm

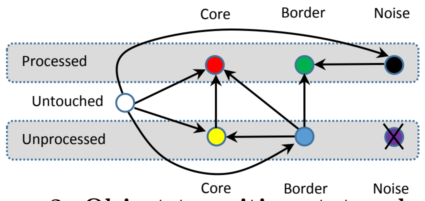


Figure 3: Object transition state schema

between d and e. The algorithm can be stopped and produces *exactly the same result as DBSCAN* without having to execute all range queries like DBSCAN. Obviously, using fewer range queries means that the clustering time is reduced. This leads to a question: (1) *how do we build an initial cluster structure and select objects for updating?* For example, if we choose object i instead of f, it clearly does not help to identify the cluster C_1. For approximating DBSCAN's result, we can simply stop the algorithm at any given time in the update step. However, producing the exact result of DBSCAN is a non-trivial task. Here another question needs to be solved: (2) *when can we stop the algorithm while guaranteeing that the final result is exactly that of DBSCAN?* For example, all additional queries after f and h are redundant since the cluster structure is identified.

3.2 The major steps of AnyDBC

Since AnyDBC is an anytime algorithm, we use T for denoting an arbitrary time point during its execution. We ommit T whenever it is clear from the context. As illustrated in Section 3.1, at a given time T, the core properties of objects are only partly determined since some range queries may not have been executed. Consequently, we start by formally defining the states of objects as follows.

Definition 6. *(Object state) The state of an object p, denoted as $state(p)$, represents the knowledge about p at a given time T. Starting with the* untouched *state at the beginning, if a range query is performed on an object p, it is marked as* processed. *Otherwise it is* unprocessed. *Depending on its current core property, p is additionally labeled as* core, border, *or* noise.

Basically, Definition 6 is an extension of Definition 2, which takes into account that a range query may not have been performed on an object p at T. At $T+1$, p may change its state via interacting with new performed range queries. For example, object h becomes a core object after the range query on object g in Figure 2. Figure 3 summarizes the transition states of objects in our algorithm. Note that the *unprocessed-noise* state does not exist since there is no way to confirm that p is noise without examining its neighbors.

Theorem 1. *At any given time T, an object p may only change its state following the transition schema in Figure 3.*

Proof. (Sketch) Let's take an *unprocessed-border* object p. If a range query is carried out on p confirming that p is a core, its state is changed to *processed-core*. Otherwise p is marked as *processed-border* following Definition 2 since it already lies inside the neighborhood of a core object, i.e., there is no way p moves to the *processed-noise* state. The other cases can be proven similarly. □

At any given time T, AnyDBC produces clusters following the density-based notions of DBSCAN described in Section 2.1 and the current states of objects described above. Generally, AnyDBC consists of the following major steps.

Step 1: Building an initial structure. In this step, some range queries are performed to build a preliminary cluster structure of the data. AnyDBC starts by randomly choosing an object p with *untouched* state for querying. If object p has fewer than μ neighbors, it is marked as *processed-noise* and is stored in a noise list L for a post processing procedure at the last step. If p is a core object, p is marked as *processed-core*, and all *unprocessed* objects $q \in N_\epsilon(p)$ are marked as *unprocessed-border*. If q is *processed-noise*, its state will be changed to *processed-border*. If q is *processed-core* or *unprocessed-core*, its status remains unchanged. If an *unprocessed-border* object $r \in N_\epsilon(p)$ has more than μ neighbors, it surely is a core object regardless of further range queries, thus it is marked as *unprocessed-core*. The whole process is repeated until there is no *untouched* object left, i.e., all objects are either in *processed* or *unprocessed* state.

Since queries are performed only for *untouched* objects, the total number of range queries in this step is much smaller than the total number of range queries in DBSCAN (see Section 5 for a comprehensive study). At the end of step 1, all objects $p \in O$ are grouped into small, potentially overlapping sets of objects called *primitive clusters*.

Definition 7. *(Primitive cluster) At a given time T, a* processed-core *object $p \in O$ together with its known density-connected neighbors form a so-called* primitive cluster, *denoted as $pclu(p)$, where p is the cluster representative. If a primitive cluster consists of only p and its ϵ-neighborhood $N_\epsilon(p)$, it is called a* primitive circle *and is denoted as $pcir(p)$.*

Corollary 1. *At any given time T, all objects $q \in pclu(p)$ belong to the same cluster.*

Proof. Straightforward from Definitions 5. □

Note that, at the end of Step 1 and 2, we only have *primitive circles*. At the next steps, *primitive circles* will be merged to form the more general *primitive clusters*.

Step 2: Creating a graph. In this step, we use a graph G, called the *cluster graph*, for capturing the relationships among primitive clusters and representing the current cluster structure. Two primitive clusters are directly density-connected at a given time T iff there is a chain of objects inside them that makes their representatives density-connected.

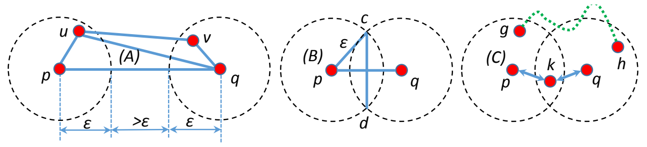
Definition 8. *(Direct cluster connectivity) Two primitive clusters $pclu(p)$ and $pclu(q)$ are directly density-connected at a given time T, denoted as $pclu(p) \bowtie pclu(q)$, iff $\exists X = \{x_1, \cdots, x_m\} \in pclu(p) \cup pclu(q)$ so that $p \lhd x_1 \lhd \cdots \rhd x_m \rhd q$.*

Corollary 2. *At any given time T, if $pclu(p) \bowtie pclu(q)$, all objects $x \in pclu(p) \cup pclu(q)$ belong to the same cluster.*

Proof. Deriving from Definition 5 and 7. □

Lemma 1. *Given two primitive circles $pcir(p)$ and $pcir(q)$, we have:*

- *Case A: $d(p,q) > 3\epsilon \Rightarrow \forall T : \neg pcir(p) \bowtie pcir(q)$*
- *Case B: $d(p,q) > \sqrt{3}\epsilon \ \wedge \ |pcir(p) \cap pcir(q)| \geq \mu \Rightarrow pcir(p) \bowtie pcir(q)$*
- *Case C: $\exists k \in pcir(p) \cap pcir(q) : state(k) =$ processed-core $\vee\, state(k) =$ unprocessed-core $\Rightarrow pcir(p) \bowtie pcir(q)$.*



Proof. The above picture illustrates the three cases A, B, and C of Lemma 1. For simplicity, we use uv for denoting the distance $d(u, v)$ between two objects u and v here.

Case A: Let u and v be arbitrary objects in $pcir(p)$ and $pcir(q)$, respectively. We have $pu + uq \geq pq$ and $uv + vq \geq uq$ (triangle inequality). Thus $uv + vq + pu \geq pq$ or $uv \geq pq - pu - vq$. Since $pq > 3\epsilon$ and $pu, vq \leq \epsilon$, we have $uv > \epsilon \Rightarrow u \notin N_\epsilon(v) \Rightarrow \neg u \bowtie v$ (due to Definition 4) $\Rightarrow \forall T$: $\neg pcir(p) \bowtie pcir(q)$ (following Definition 8).

Case B: We have $pq^2 + cd^2 = 4\epsilon^2 \Rightarrow cd \leq \epsilon$ (since $pq > \sqrt{3}\epsilon$). Therefore, $\forall x \in pcir(p) \cap pcir(q) \Rightarrow pcir(p) \cap pcir(q) \subseteq N_\epsilon(x) \Rightarrow core(x)$ (since $|pcir(p) \cap pcir(q)| \geq \mu$) $\Rightarrow p \lhd x \rhd q$ (Definition 3) $\Rightarrow pcir(p) \bowtie pcir(q)$ (Definition 8).

Case C: We have, $core(k) \Rightarrow p \lhd k \rhd q$ (due to Definition 3) $\Rightarrow pcir(p) \bowtie pcir(q)$ (following Definition 8). □

Definition 9. *(Cluster graph) A cluster graph is a graph $G = (V, E)$, where each vertex $v \in V$ corresponds to a primitive cluster $pclu(v)$, and each edge (u, v) is assigned a state, denoted as $state(u, v)$, that represents the connectivity status of the two primitive clusters $pclu(u)$ and $pclu(v)$.*

- *if $\forall T : \neg pclu(u) \bowtie pclu(v)$, $(u, v) \nexists E$ ($state(u, v) = no$)*
- *else if $pclu(u) \bowtie pclu(v)$, $state(u, v) = yes$*
- *else if $pclu(u) \cap pclu(v) \neq \emptyset$, $state(u, v) = weak$*
- *else $state(u, v) = unknown$*

Generally, each edge of G connects two primitive clusters that may belong to the same cluster. And its state reflects how strong the connection is. For example, at the time T, if $pclu(u)$ and $pclu(v)$ share an object p and they are currently not directly density-connected, $state(u, v)$ is thus *weak*. Obviously, they have more chance to be in the same cluster than the *unknown* case. The state of an edge also changes overtime. Assume that at time $T+1$, some additional range queries reveal that p is a core object, then $state(u, v)$ thus becomes *yes* at time $T+1$ since $pclu(u)$ and $pclu(v)$ are now directly density-connected following Lemma 1. If $state(u, v)$ is *no*, it means that $pclu(u)$ and $pclu(v)$ will never be directly connected even when all range queries have been performed. Thus, (u, v) will not belong to the edge set E of G. To build the graph G, AnyDBC first puts all the primitive clusters found in step 1 into V. Then the states of all edges are determined following Lemma 1 and Definition 9 above.

Lemma 2. *Given two nodes $pcir(u)$ and $pcir(v)$ of G, if u and v are truly density-connected, there must exist a path that connects $pcir(u)$ and $pcir(v)$ in G.*

Proof. (Sketch) Assume that for all chains of nodes $\{u, \cdots, x, y, \cdots, v\}$ of G, there always exist two adjacent nodes x and y so that $state(x, y) = no$. According Lemma 1 Case A, $\forall p \in x\ \forall q \in y : d(p, q) > \epsilon$. Thus, there is no chain of objects that connect u and v (following Definition 4). This lead to a contradiction. □

Lemma 2 directly implies that the connectivity of u and v can be determined via examining all the edges of G.

Step 3: Finding connected components. AnyDBC determines chains of directly density-connected primitive clusters by finding connected components of the graph G'.

Definition 10. *(Cluster connection graph) A cluster connection graph $G' = (V, E')$ is the subgraph of cluster graph G, where $E' = \{(u, v) | (u, v) \in E \ \wedge \ state(u, v) = yes\}$.*

Corollary 3. *Given two nodes u and v in a connected component C of G', $pclu(u)$ and $pclu(v)$ belong to the same cluster at a given time T.*

Proof. Deriving from Definition 9 and Corollary 2. □

Following Corollary 3, at any given time T, an intermediate clustering result of AnyDBC can be produced by labeling all nodes of G' according to their connected components. Objects are then labeled according to the labels of their representative nodes. This is an advantage of AnyDBC compared to other techniques. Since the size of G is very small, the label propagation time is significantly reduced. Thus, we obtain high quality approximations efficiently (Section 5).

Step 4: Merging connected components. All primitive clusters on a connected component will be merged together to reduce the number of nodes in G, thus enhancing the performance, e.g., reducing the time for finding connected components in subsequence steps. For each component, we randomly select the representative of a node inside it as a representative for the whole group. In experiments, the total number of graph nodes decreases extremely fast at the first few iterations due to this merging scheme (see Section 5).

Theorem 2. *Each connected component $C \in G'$ represented by an object p is a primitive cluster after merging.*

Proof. (Sketch) $\forall q \in C$ and $\forall x \in pclu(q)$, we have $x \bowtie q$ (Definition 7). Since p and q belong C, there is a path of nodes $X = \{p = x_1, \cdots, x_m = q\}$, where $state(x_i, x_{i+1}) = yes$, i.e., $x_i \bowtie x_{i+1}$ (Definition 8 and 9). Thus, $x \bowtie p$. Therefore, $pclu(p)$ is a primitive cluster (Definition 7). □

The states of edges of the new graph still follow Definition 9, and can be directly inferred from G following Lemma 3.

Lemma 3. *Given two connected components $C = \{c_1, \cdots, c_2\}$ and $D = \{d_1, \cdots, d_2\}$ of G' at a given time T. We have:*

- *Case A: $\forall c_i \in C, d_j \in D : state(c_i, c_j) = no \Rightarrow state(C, D) = no$*
- *Case B: $\exists c_i \in C, d_j \in D : state(c_i, c_j) = weak \Rightarrow state(C, D) = weak$*
- *Case C: otherwise $state(C, D) = unknown$*

Proof. Since we only merge nodes in each connected component of G', we have for all $c_i \in C$ and $d_j \in D \Rightarrow \neg pclu(c_i) \bowtie pclu(d_j)$ ($state(c_i, d_j) \neq yes$) $\Rightarrow \neg \cup pclu(c_i) \bowtie \cup pclu(d_j) \Rightarrow \neg pclu(C) \bowtie pclu(D)$.

Case A: $\forall c_i, d_j : state(c_i, d_j) = no \Rightarrow \forall T : \neg pclu(c_i) \bowtie pclu(d_j)$ (Definition 9) $\Rightarrow \forall T : \neg pclu(C) \bowtie pclu(D) \Rightarrow state(C, D) = no$ (Definition 9).

Case B: $state(c_i, d_j) = weak \Rightarrow pclu(c_i) \cap pclu(d_j) \neq \emptyset$ (Definition 9) $\Rightarrow pclu(C) \cap pclu(D) \neq \emptyset \Rightarrow state(C, D) = weak$ (Definition 9).

Case C: directly inferred from Case A and B. □

Step 5: Checking the stopping condition. As presented in Section 3.1, knowing when to stop the algorithm is crucial for improving the performance of AnyDBC.

Theorem 3. *At a given time T, if $\forall (u, v) \in E : state(u, v) = yes$, then $\forall T' > T\ \forall (u, v) \in E : state_T(u, v) = state_{T'}(u, v)$, where $state_T(u, v)$ and $state_{T'}(u, v)$ are the states of the edge (u, v) at time T and T'.*

PROOF. If $state_T(u, v) = yes$, we have $pclu(u) \bowtie pclu(v)$ (Definition 9). Due to Theorem 1, the state of an object p only changes from noise to border to core, i.e., not from core to border to noise. Thus, there is no way that a chain of object $u \lhd x_1 \lhd \cdots \rhd x_m \rhd v$ that connects u and v (Definition 8) be broken at any time $T' > T$, i.e., $state_{T'}(u, v) = yes$. $\square$

Following Theorem 3, if all edges of the graph G have states *yes*, AnyDBC can stop without examining all range queries since there will be no change in the graph G and thus in the final clustering result following Corollary 3. As a result, its performance is significantly enhanced. In the next following Section, we will prove that the final results of AnyDBC and DBSCAN are identical.

Step 6: Selecting objects for range queries. A naive way is randomly choosing an unprocessed object for performing a range query. However, as we demonstrate in Section 5, it is inefficient since the current cluster structure is not exploited. Thus, we introduce an *active learning* [15] scheme to significantly reduce the number of used range queries for building the final clustering result. The general idea is letting the algorithm *iteratively and actively* learn the current cluster structure from the graph G and choose objects that it *considers* most effective for updating the cluster structure. According to Theorem 3, the faster we eliminate all *weak* and *unknown* edges from G, the sooner AnyDBC reaches its final state, i.e., the fewer queries are used. To do so, AnyDBC first evaluates the importance of each node of G. Then, it ranks all unprocessed objects according to its current neighbors and its position inside G. Thoses with highest scores are chosen as targets for performing range queries.

DEFINITION 11. *(Node statistic) At a given time T, the statistical information of a node $u \in V$, denoted as $stat(u)$, is defined as follows:*

$$stat(u) = \frac{usize(u)}{|pclu(u)|} + \frac{|pclu(u)|}{n}$$

where $usize(u)$ is the number of unprocessed objects inside $pclu(u)$ and n is the number of objects.

DEFINITION 12. *(Node degree) Given a node u and its adjacent nodes $N(u)$ in the graph G. The degree of u, denoted as $deg(u)$, at a given time T is defined as follows:*

$$deg(u) = w\left(\sum_{v \in N(u) \wedge state(u,v)=weak} stat(v)\right) + \sum_{v \in N(u) \wedge state(u,v)=unknown} stat(v) - \psi(u)$$

where $\psi(u) = 0$ if u does not contain border objects otherwise it is the total number of weak *and* unknown *edges of u.*

The degree of a node u measures how certain the node u is wrt. its adjacent nodes. Intuitively, high $deg(u)$ means that u lies inside a highly uncertain area with many undetermined connections. Thus, if a range query is performed on u, it has higher change to either connect u and its adjacent nodes v ($state(u, v) = yes$) or to separate u from its adjacent nodes ($state(u, v) = no$). Moreover, if two nodes u and v have some common objects, they are more likely to be directly density connected. In contrast, if $state(u, v) = unknown$, it is much harder to determine the true connection status of u and v. Therefore, we assign a higher weight $w = |V|$ for edges with *weak* states than edges with *unknown* states. Moreover, if node u contains border objects, it should be processed later due to the effect of Lemma 5 and the merging scheme of each new query in Step 7 for ensuring an earlier termination of the algorithm. The general idea is surrounding a promising node with *processed* objects so that any future queries can bring it closer to the empty state as stated in Lemma 5.

DEFINITION 13. *(Object score) The score of an unprocessed object p, denoted as $score(p)$, at a given time T is defined as follows:*

$$score(p) = \sum_{u \in V \wedge p \in pclu(u)} deg(u) + \frac{1}{nei(p)}$$

where $nei(p)$ is the number of neighbors of p at time T.

The score of an object p is calculated based on the sum of degrees of all nodes $v \in V$ that contain p and its current number of neighbors. Beside the node degrees discussed above, we prefer objects with lower numbers of neighbors since performing range queries on them would lead to more core objects to be revealed with each query. Moreover, since q has fewer neighbors, it is more likely to be in a highly uncertain area. Thus, examining it earlier can help to eliminate this uncertainty faster. Note that only *unprocessed-border* and *unprocessed-core* objects need to be examined.

Step 7: Performing range queries. Assume that an object p is selected for performing range queries. If p is not a core object, it is marked as *processed-border* since it is already inside a cluster. Otherwise, it is marked as *processed-core.* And for all objects $q \in N_\epsilon(p)$, the status of q changes following the transition schema in Figure 3. Moreover, $N_\epsilon(p)$ is merged into all nodes u that contain p.

THEOREM 4. *At a given time T, if $core(p) \wedge p \in pclu(u)$, $N_\epsilon(p) \cup pclu(u)$ is a primitive cluster.*

PROOF. Since $p \in pclu(u) \Rightarrow p \bowtie u$ (Definition 7) $\Rightarrow \forall q \in N_\epsilon(p)$: $p \bowtie q$ (Definition 4) $\Rightarrow q \bowtie u$ $(core(p)) \Rightarrow N_\epsilon(p) \cup pclu(u)$ is a primitive cluster (Definition 7). $\square$

Step 8: Updating the cluster graph. In this step, the graph G is updated to reflect changes in the current cluster structure after the new queries q following Definition 10, and Lemma 4 and 5 described below.

LEMMA 4. *Given two nodes u and v at a given time T, if $usize(u) = 0 \vee usize(v) = 0$ and $\neg pclu(u) \bowtie pclu(v)$, then $\forall T' > T : state(u, v) = no$, where $usize(u)$ and $usize(v)$ are the numbers of unprocessed objects of $pclu(u)$ and $pclu(v)$.*

PROOF. Assume that a node u is fully processed wlog. and $\exists T' > T : pclu(u) \bowtie pclu(v)$, there exists a chain of core objects $X = \{v, \cdots, d, e, \cdots, u\}$ that connects the two core objects v and u at T' (Definition 8). However, if e is a core object, all of its neighbors including d are merged into $pclu(u)$ at a time $T_1 \leq T$ in step 7. Thus, $pclu(u) \bowtie pclu(v)$ at the time T. This leads to a contradiction. $\square$

According to Lemma 4, if a node u is fully processed, all of its adjacent nodes v, where $state(u, v) = weak$ or *unknown*, end up having their link broken, i.e., removed from E ($state(u, v) = no$). This pushes the algorithm moving faster to the stop condition described in Theorem 3.

LEMMA 5. *Given two primitive clusters $pclu(u)$ and $pclu(v)$, if $\exists k \in pcir(u) \cap pcir(v) : state(k) = processed\text{-}core \vee state(k) = unprocessed\text{-}core \Rightarrow pcir(u) \bowtie pcir(v)$.*

PROOF. Similar to the case C of Lemma 1. $\square$

Step 9: Processing outliers. In Step 1, any noise object p together with its neighbors is stored in a noise list L for a post processing procedure in this step. The goal here is to determine whether p is a true outlier or it is a border object of a cluster. To do so, AnyDBC scans all objects p in L to see whether p has been placed inside the nodes of the graph

```
function $C$ = AnyDbClu ($O, d, \mu, \varepsilon, \alpha, \beta$)
input:   dataset $O$, distance function $d$, and parameters $\mu, \varepsilon$ of DBSCAN
         the query block size $\alpha, \beta$
output:  the final clustering result $C$
begin
  /* step 1: building an initial cluster structure */
  while there exist untouched objects in $O$ do
    $S$ = set of $\alpha$ untouched objects
    for all objects $o$ in $S$ do
      perform range query on $o$ and mark the state of $o$
      if $o$ is a core object then mark the states of its neighbors in $N_\varepsilon(o)$
      if $o$ is a noise object then put $o$ and $N_\varepsilon(o)$ into the noise list $L$
  /* step 2: creating a cluster graph $G = (V, E)$*/
  put all primitive clusters into $V$ as nodes
  determine the states of all edges $e$ in $E$
  /* repeatedly select objects for range queries until terminated */
  do
    /* step 3: finding connected components */
    find all connected components of $G$ via the yes states
    /* step 4: merging connected components */
    merge each connected component of $G$ into a single node
    calculate the state of each edge of the new graph $G$
    return an intermediate clustering result $C'$ if required
    /* step 5: checking the stopping condition */
    $b$ = check if $G$ only contains edges with yes or no states
    if $b$ = false then
      /* step 6: selecting objects for range queries */
      for all nodes $v$ in $V$ do
        calculate the node statistic for $v$
        calculate the node degree for $v$
      calculate object scores for all unprocessed objects in $O$
      $S$ = set of $\beta$ objects with highest scores
      /* step 7: performing range queries */
      for all objects $o$ in $S$ do
        perform range queries on the object $o$
        update the states of $o$ and its neighbors $N_\varepsilon(o)$
        merge $N_\varepsilon(o)$ to all nodes that contain $o$
      /* step 8: updating the cluster graph */
      update the states of all edges $e$ in $E$
  while the stopping condition is not reached ($b$ = false)
  /* step 9: processing outliers */
  for all objects $o$ in $L$ do
    check if $o$ is truly a noise or a border object
return the final clustering result $C$
```

Figure 4: The pseudo code of AnyDBC

G, i.e., $state(p) = processed\text{-}border$. If so, we do not need to process p anymore. Otherwise, if there exists an object $q \in N_\epsilon(p)$ so that $state(q) = unprocessed\text{-}core$, p is surely a border object connected to q and its cluster. Conversely, we sequentially perform range queries on *unprocessed* objects $q \in N_\epsilon(p)$ looking for a core object. If a core object q is found, p is determined as a border object and has the same label as q. Otherwise, p is a noise object. It is worth recalling that any object $p \in L$ has fewer than μ neighbors, thus the overhead of this step is very small. Note that, for a shared border object p of two different core objects a and b, depending on the object examining order, it will be assign the label of a or b as in DBSCAN.

3.3 The algorithm AnyDBC

Figure 4 shows the pseudo code of the algorithm AnyDBC following the nine main steps described above. However, while we illustrate the operation of AnyDBC in Section 3.2 using a single query for each iteration for clarity, AnyDBC actually queries objects in blocks of size α for step 1 and β for step 6 and 7, i.e., it chooses α and β objects for performing queries at each iteration of step 1, 6 and 7. This provides major benefits: (1) enhancing the intermediate clustering qualities at earlier steps by allowing overlapping primitive circles, and (2) reducing the overall overhead of the anytime scheme of AnyDBC (e.g., merging nodes and finding connected components), which is the major bottleneck of AnyDBC, thus significantly enhancing the performance. Unless otherwise stated, we always use $\alpha = \beta$ for simplicity. In Section 5, we will extensively analyze the effect of the block sizes α and β on the performance of our algorithm.

Correctness of the algorithm. Unlike most techniques for enhancing DBSCAN, AnyDBC is an exact technique, i.e., it produces the same final results as DBSCAN.

LEMMA 6. *Assume that AnyDBC is run to the end, its final result is completely identical to that of DBSCAN.*

PROOF. (Sketch) As described in Section 3.2, AnyDBC produces clusters following the notion of DBSCAN. Therefore, if all queries are performed, AnyDBC and DBSCAN obviously produce the same results. Though AnyDBC only uses a small sets of all range queries at each iteration, it still produces clusters following the density-connected notion of DBSCAN as demonstrated by Corollary 1, 2, and 3. According to Theorem 3, at a given time T when all edges of G have *yes* or *no* states, there is no more change in the graph G and thus the cluster structure regardless of how many additional queries will be performed. Thus the clustering result of AnyDBC at T is similar to that at T_∞ when all queries are executed with only a minor difference on some shared border objects, which are assigned to clusters based on the examined order of objects in both DBSCAN and AnyDBC. □

Complexity analysis. For simplicity, we analyze here a worst case complexity of AnyDBC. Let n be the number of objects, $v = |V|$ be the number of initial nodes of G (after Step 2), $l = |L|$ be the size of the noise list L, and b is the total number of update iterations of AnyDBC. Step 1 needs $O(vn)$ time for querying and marking objects. Creating the graph in Step 2 consumes $O(v^2n)$ time. $O(\sum_{i=1}^{b} v_{i-1}^2)$ is the time needed to find connected components in Step 3, where v_i is the number of graph nodes at iteration i and $v_0 = v$. Merging nodes in Step 4 costs $O(\sum_{i=1}^{b} v_{i-1}n)$ time for merging nodes, $O(\sum_{i=1}^{b} v_i^2)$ time for relabeling the states of edges, and $O(\sum_{i=1}^{b} v_i n)$ time for updating the number of unprocessed objects inside each node. Step 5 simply requires $O(\sum_{i=1}^{b} v_i^2)$ time for checking the stopping condition. Step 6 needs $O(\sum_{i=1}^{b} v_i^2)$ time for calculating node degrees, $O(\sum_{i=1}^{b} v_i(n-v-(i-1)\beta))$ time for calculating object scores, $O(\sum_{i=1}^{b}(n-v-(i-1)\beta)log(n-v-(i-1)\beta))$ time for sorting unprocessed objects. Step 7 has $O(b\beta n)$ time for querying, $O(\sum_{i=1}^{b} v_i\beta n)$ time for merging nodes, and $O(\sum_{i=1}^{b} v_i n)$ for updating the node size. It takes $O(\sum_{i=1}^{b} v_i^2 n)$ time to update the graph in Step 8. The last Step 9 consumes $O(l\mu n)$ time for detecting true outliers. It is important to note that the true time complexities at all steps of AnyDBC are much smaller than those described above if indexing techniques are used. Moreover, in the experiments (see Section 5.1 for an experimental analysis), $v_i \ll v \ll n$ and $b \ll b_{max}$, where $b_{max} = (n-v)/\beta$ is the maximal number of iterations of AnyDBC. Consequently, the final complexity of AnyDBC is much smaller than $O(n^2)$ time of DBSCAN.

AnyDBC needs $O(v^2 + vn + n + v + l\mu)$ space for storing the graph G, objects inside nodes, the number of neighbors for unprocessed objects, the number of unprocessed objects inside nodes, and the noise list L. Since $v \ll n$ (see Section 5.1), the overall space complexity of AnyDBC is thus $O(vn)$ in the worst case, which is linear to the number of objects.

4. DISTANCE FUNCTION

In this paper, we use two different kinds of distance measures for assessing the performance of our algorithm including L_p distances and Edit Distance.

L_p distances are among the most well-known distance measures for traditional vector data. Most state-of-the-art techniques for enhancing DBSCAN, e.g. [6], are specifically designed for these data under the L_p distance.

For complex data such as time series or trajectories, there are many kinds of complex distance measures [3], which is more effective than L_p. However, all of them have quadratic complexity in contrast to the linear time of L_p, which is a major bottleneck during the clustering process. There are only few extensions of DBSCAN which are capable of dealing with these complex data and distances, e.g., [2,12]. Here we use some trajectory datasets and Edit distance with Real Penalty (ERP) [3] as a distance measure for demonstrating the performance of our algorithm. Due to space limitation, interested reader please refer to [3] for more details.

5. EXPERIMENTS

In Section 5.1, we study important characteristics of AnyDBC. Then, we study its performance in comparison with state-of-the-art techniques on vector data and complex data in Section 5.2 and 5.3. All experiments are conducted on a Linux workstation with 3.1 GHz CPU and 128 GB RAM using C++. Averaged runtimes of all rival algorithms as well as averaged cumulative runtimes of AnyDBC are reported over 10 runs. For speeding up the range query process, we use *kd*-trees [14] for indexing data. However, any other indexing techniques like R-tree [5] can also be used.

5.1 Characteristics

For studying the behavior of AnyDBC, we create four synthetic 2D datasets DS1-DS4, which contain from 3254 to 9554 points belonging to 16 to 32 clusters. Based on them, we can construct bigger datasets by randomly placing additional points in the unit circle of each existing object inside each cluster (except outliers). Here, DS1x100 means 99 more objects are additionally placed for each object of the original dataset DS1. Thus, we can visually study the effect of arbitrarily shaped clusters on AnyDBC. We can also study the behavior of AnyDBC when the number of objects increases while the cluster structures are preserved. Moreover, we have ground truths for assessing the intermediate clustering results of AnyDBC. Experiments on higher dimensional datasets will be conducted in next Sections. Unless otherwise stated, we use $\mu = 5$, $\epsilon = 1.0$, and $\alpha = \beta = 512$.

Performance of AnyDBC. Figure 5 shows the performances of AnyDBC with increasing numbers of objects for DS1-DS4. As we can see from (B), AnyDBC is much faster than DBSCAN. Moreover, the denser the clusters, i.e., the more objects in the datasets, the higher the speedup factors (compared to DBSCAN (see (B)). The reasons can be found in (C) and (D). First, AnyDBC consumes much fewer queries than DBSCAN, e.g., it requires only 6964.4 (0.25%) range queries on average for clustering DS1x300 with 2783567 objects. Second, the initial numbers of graph nodes are also very small, e.g., there are only 3441.6 (0.12%) initial nodes on average for clustering DS1x300. Moreover, during its runtime, the number of graph nodes is significantly reduced at each iteration of AnyDBC (see Figure 8 (D)). Thus, the label propagation time is significantly reduced compared to DBSCAN. These two reasons make AnyDBC orders of magnitude faster than DBSCAN at the end. Moreover, when we increase the number of objects, the total number of queries as well as the number of graph nodes does not change much. And so do the runtimes of AnyDBC (see (A)). However, the runtimes of DBSCAN change significantly due to its full quadratic complexity. Thus, the speedup factors increase.

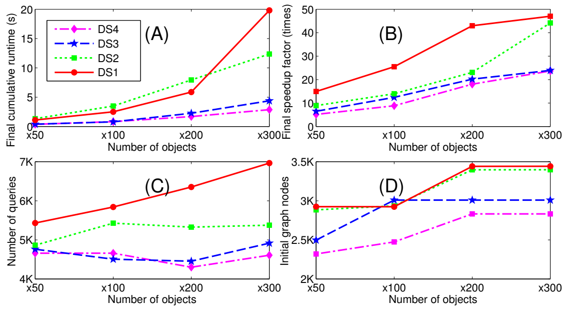


Figure 5: Performances of AnyDBC with increasing number of objects for DS1-DS4.

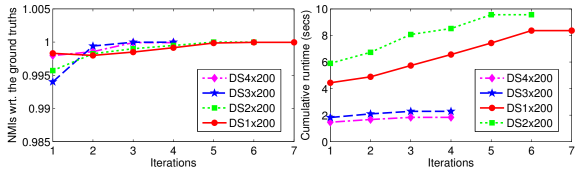


Figure 6: NMI scores and cumulative runtimes of AnyDBC at each iteration for DS1-DS4. It acquires near perfect clustering results at the first iteration.

Anytime property of AnyDBC. For assessing the intermediate clustering results of AnyDBC, we use Normalized Murtual Information (NMI) [17] to compare them with the ground truths. NMI results are in [0,1] where 1 means a perfect clustering result and vice versa. As we can see from Figure 6, AnyDBC acquires almost perfect clustering results indicated by very high NMI scores even at the first step. For DS1x200 (with 1855767 points) as an example, AnyDBC starts with a NMI = 0.998 which is almost identical to the NMI = 0.999 of DBSCAN. However, while DBSCAN requires 252.4 secs, AnyDBC needs only 4.4 secs, which is 57.3 times faster. Even if AnyDBC runs till the end, it only consumes 8.3 secs, around 30.4 times faster. This anytime scheme of AnyDBC thus provides a very efficient and effective approach for coping with very large datasets.

The *active* selection scheme of AnyDBC. Figure 7 shows the NMI scores and cumulative runtimes of the *active* scoring scheme of AnyDBC for selecting objects in Step 6 and a random scheme, i.e., objects are randomly selected, for DS1x0200. Due to its efficient scheme, AnyDBC stops after only 7 iterations, 8.3 secs and 6149 queries while the random scheme requires 225 iterations 87.7 secs and 117761 queries to finish.

The role of the block sizes α and β. Figure 8 shows the effects of the block sizes α and β on the performance of AnyDBC. Obviously, increasing α will increase the number of initial nodes in G due to the overlap between primitive circles in step 1 (see (B)). This overlap, however, leads to the faster reduction of the graph nodes due to the merging scheme in Step 4. Beside, since more queries are performed in each step, the number of nodes also decreases faster at each iteration (see (D)) due to the node merging scheme (Step 4) and so do the overall operation costs on graph G. Thus, the final cumulative runtimes of AnyDBC are significantly reduced (see (A)). However, since nodes are merged earlier, they become bigger and therefore more queries are required for clearly identifying the *no* states of edges (Lemma 4). Consequently, the total numbers of required queries in-

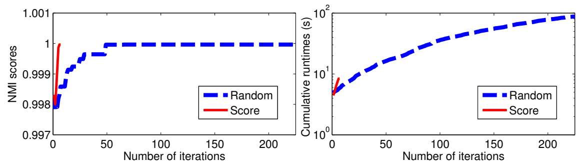


Figure 7: Performance of the active selection scheme of AnyDBC for DS1x0200

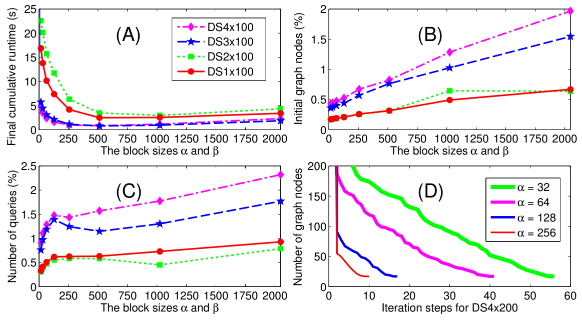


Figure 8: The effects of the block sizes α and β ($\alpha = \beta$) on the performance of AnyDBC

crease and remain relatively stable if β is large enough since more core objects are detected at each step also leading to faster identifying the *yes* states of edges (Lemma 5) (see (C)). However, the additional costs for performing queries are still small enough to be dominated by the operation cost reductions described above. Thus, overall, the cumulative runtimes of AnyDBC are still reduced ((A)). When α and β are too big, redundant queries will happen, thus leading to the moderate degradation of the overall performance. For DS4 as an example, when $\alpha = \beta = 1024$, AnyDBC requires only 1.1 secs to finish, while it consumes 2.2 secs when $\alpha = \beta = 2048$. Thought 512 seems to be a good choice, the larger a dataset, the bigger value of α, β we should choose. One simple scheme is choosing $\alpha = \beta$ so that the maximal number of iterations should be several thousands at most.

The effects of the parameters μ and ϵ. Figure 9 (bottom) shows the effect of the parameter μ on the performance of AnyDBC. As we can see, increasing μ slightly increases the runtimes of AnyDBC since more queries are required to identify the core properties of unprocessed objects. However, the change diminishes with larger μ since the size of graph G decreases, thus leading to the reduction of the operation cost. In contrast, the number of initial graph nodes is slightly reduced since there are more noise objects.

The parameter ϵ has a stronger impact than μ on AnyDBC. Obviously, increasing ϵ will reduce the number of initial graph nodes since more objects are marked inside each primitive circle. Moreover, more unprocessed objects will be determined as core ones with each additional range query. Thus, the runtimes as well as the total numbers of queries are also reduced. However, when ϵ is large enough, the states *weak* and *unknown* may occur more frequently as replacements for the *no* state. Consequently, more range queries are required for breaking these edges according to Lemma 4. Thus, the runtime of AnyDBC increases as we can see from Figure 9 (top). However, it is interesting to note that, the bigger the ϵ, the faster AnyDBC reaches the final clustering result of DBSCAN at each iteration. Thus, in term of the

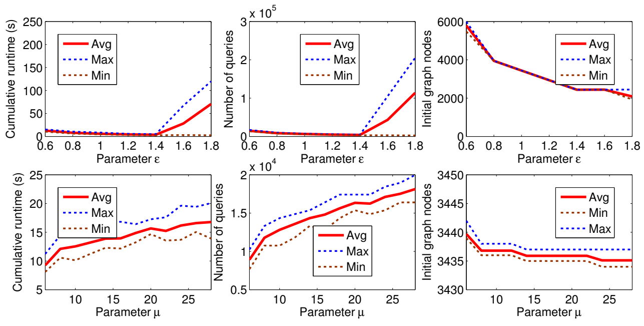


Figure 9: The effects of the parameters μ ($\epsilon = 1.0$) and ϵ ($\mu = 5$) for DS1x200 (1855767 points)

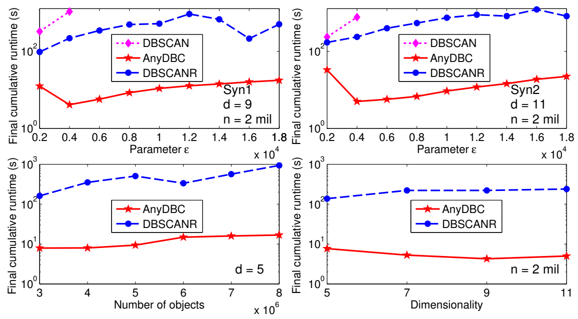


Figure 10: The performance of AnyDBC on various synthetic datasets

anytime scheme, users can stop the algorithm sooner for acquiring higher speed up factors. Due to the space constrain, we do not show a Figure for this experiment here.

5.2 Performance on vector data

We compare the performances of AnyDBC in with state-of-the-art techniques including the original DBSCAN [5] and its fastest variant in [6] (here denoted as DBSCANR) using the binary file provided by the authors. We include the building time for *kd*-tree into the final runtimes of AnyDBC and DBSCAN since DBSCANR does not use indexing techniques for maximal comparison fairness. A-DBSCAN [11] is another variant of DBSCAN. However, it must store the whole ϵ-neighborhood graph and repeatedly performs clustering at each level using DBSCAN. Obviously, it is both memory and time consuming, especially for large datasets. Thus, we exclude it from the comparison for clarity.

Figure 10 (top) shows the performance on synthetic datasets, generated by the data generator scheme of DBSCANR [6] with 2 million points in 9 (Syn1) and 11 (Syn2) dimensions, with different values of ϵ ($\mu = 5$). AnyDBC is much faster than and DBSCANR and DBSCAN, e.g., for the Syn1 dataset and $\epsilon = 4000$, AnyDBC requires 3.68 secs, which is 60 and 297.1 times faster than DBSCANR (221 secs) and DBSCAN (1093.6 secs), respectively. Overall, AnyDBC is up to 82.2 times faster than DBSCANR. It is mainly due to its efficient range query reduction and label propagation reduction scheme as explained above. Figure 10 (bottom) shows the scalability of AnyDBC and DBSCANR wrt. the numbers of objects ($\mu = 5$ and $\epsilon = 5000$) and dimension of data ($\mu = 5$ and $\epsilon = 4000$). The higher the number of objects and dimensions, the better the performance of AnyDBC. For clustering 5 million objects, AnyDBC finishes within 9.3 secs while DBSCANR and DBSCAN take 505.4 secs and 19388.8 secs, respectively. Thus, it is 54.3 times and 2084.8 times faster than DBSCANR and DBSCAN, respectively. Overall AnyDBC is up to 55.5 times faster than DBSCANR and several thousand times faster than DBSCAN.

Figure 11 (top) shows the performance of AnyDBC wrt. different values of ϵ on two real datasets: the Gas sensor array data (4208261 points in 16D - $\mu = 50$ and $\alpha = \beta = 2048$) and the PAMAP2 dataset (974479 points in 39D - $\mu = 50$ and $\alpha = \beta = 512$) acquired from the UCI archive (http://archive.ics.uci.edu/ml/). AnyDBC significantly outperforms the others. For Gas data with $\epsilon = 6000$, AnyDBC requires 69.1 secs which is 89.0 and 245.2 times faster than DBSCANR (6151.4 secs) and DBSCAN (16946.6 secs), respectively. For the PAMAP2 dataset, we stop DBSCANR after 24 hours and thus the speedup factors should be greater than 297.9 times overall. If we stop AnyDBC right after Step 1 (while still having nearly optimal results as demonstrated in Figure 6), the speedup factors are much higher. For $\epsilon = 30000$ as an example, AnyDBC Step 1 and AnyDBC

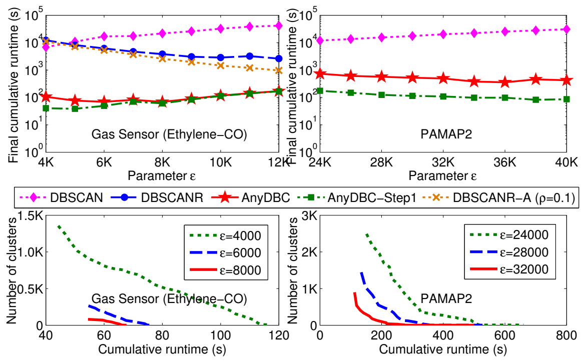


Figure 11: The performance of AnyDBC on real datasets

requires 112.6 and 521.6 secs, which are 155.6 and 34.4 times faster than DBSCAN (17520.9 secs), respectively. Overall, AnyDBC is up to 267.6 and more than 298.6 times faster than DBSCAN and DBSCANR on both datasets. Though DBSCANR is an elegant algorithm and works very well on low dimensional datasets ($d \leq 5$), it has problems when coping with high dimensional ones due to its grid-based scheme. The number of cells increases exponentially with the dimension of data and thus significantly affects its performance.

Approximation. We further compare AnyDBC with one of the best approximation techniques for DBSCAN (denoted as DBSCANR-A) with different approximation factors $\rho = 0.001, 0.01, 0.1$ as suggested in [6]. For the Gas data, DBSCANR-A is up to 2.7 times faster than DBSCANR. For $\epsilon = 4000$ and $\rho = 0.1$ as an example, it requires 11114.3 secs (compared with 12247.7 secs of DBSCANR) and discovers 8 clusters like DBSCAN. AnyDBC requires 104.8 secs to produce 8 clusters like DBSCANR-A and ends after 105.2 secs, which is around 105 times faster than DBSCANR-A. Overall, AnyDBC is up to 105.2 times faster than DBSCANR-A. If we increase ρ to 1 (extremely coarse), the runtime of DBSCANR-A slightly decreases to 10814.2 secs. However, it discovers only 7 instead of 8 clusters like DBSCAN. For the PAMAP2 dataset, we have to stop DBSCANR-A after 12 hours running each.

Anytime. Since we do not have ground truths, we plot the numbers of clusters vs. the runtimes at each iteration of AnyDBC in Figure 11 (bottom). As we can see, the numbers of clusters decrease quickly at each iteration and remain stable when they are close to the results of DBSCAN. This implies that the intermediate results of AnyDBC comes very quickly to the final results as studied in Section 5.1.

5.3 Performance on complex data

We study the performance of AnyDBC on two trajectory datasets under the ERP distance [3]. The synthetic Cross dataset contains 410 trajectories consisting 21 to 23 points in 3D. The Corpus dataset [12] consists of the Corpus Callosum bundle, one of the biggest white matter structures in human brain that connects left and right hermit spheres. It has 1103 fibers consisting of 22 to 73 3D points each. Based on them, we produce bigger datasets by adding some Gaussian noise to each trajectory and inserting them into the dataset. Here Crossx10 means each trajectory is duplicated 9 times. There are some variants of DBSCAN for coping with complex datasets such as DBSCAN-XS [2]and A-DBSCAN-XS [12], which uses LB distances and a data structure called Xseedlist to reduce the distance calculations, and DBSCAN-M [12], which uses a LB distance for speeding up the range query process in a *filter-and-refinement* scheme. Thus, they are our comparison targets. We use LB_{ERP} [3] as a LB distance for ERP [3].

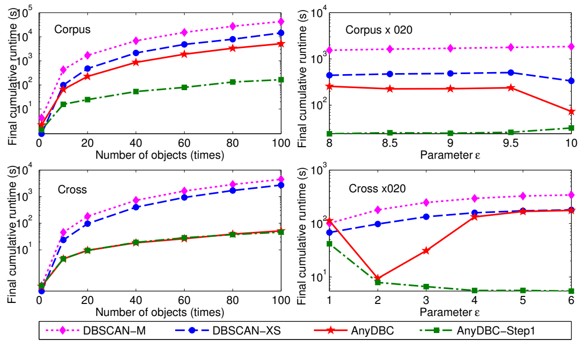


Figure 12: The performance of AnyDBC on trajectory datasets

Figure 12 (left) illustrates the performance of these algorithms when the sizes of the datasets increase from 1 to 100 times ($\mu = 5$, $\alpha = 128$, and $\epsilon = 9$ for Corpus and 2 for Cross). The bigger the datasets, the better the performance enhancement of AnyDBC. The performances of AnyDBC and DBSCAN-M are not different much on the original Cross dataset. However, if we increase the size of Cross by 100 times, AnyDBC is 86 times faster than DBSCAN-M due to its calculation reduction scheme. DBSCAN-XS is slightly faster than AnyDBC on the original datasets. However, when the sizes of the datasets increase, DBSCAN-XS performs worse than AnyDBC (52.5 times on Crossx100). The reasons are: (1) it requires much distance calculation than AnyDBC, especially when the data volume increases and (2) the high operation cost of its Xseedlist. Moreover, it consumes quadratic memory wrt. the number of objects. Thus, it is only applicable for small datasets. The final cumulative runtimes of A-DBSCAN-XS are almost similar to those of DBSCAN-XS. Thus, we exclude it from the Figure for clarity. Figure 12 (right) shows the performances of all algorithms on Crossx020 and Corpusx020 datasets with different values of ϵ. AnyDBC is up to 29 and 10.2 times faster than DBSCAN-M and DBSCAN-XS, respectively. Moreover, if AnyDBC is stopped after Step 1, it acquires much higher speedup factors compared to others. For Corpusx020 (38000 trajectories), AnyDBC requires 25.3 secs at Step 1 (with $\epsilon = 8.5$) which is 8.9, 177.3, 18.7 times faster than AnyDBC (225.1 secs), DBSCAN-M (1631.9 secs), and DBSCAN-XS (473.1 secs), respectively.

6. RELATED WORKS AND DISCUSSION

Approximation techniques. Many techniques such as BRIDGE [4] accelerate DBSCAN by compressing data into smaller subsets using k-Means. Then DBSCAN is performed on features of subsets for reducing runtimes. AnyDBC also summarizes data into primitive clusters using range queries. However, instead of performing clustering on features and consequently approximating the result of DBSCAN, AnyDBC uses these primitive clusters as a guideline for studying the potential cluster structure and producing clustering results but does not disregard the original object representation. DBRS [18] only performs range queries on some objects and merges the neighborhoods of two core objects to form clusters if they share some arbitrary objects. SDBSCAN [19] only selects a subset of objects to perform clustering. Then, the rest is assigned labels according to their nearest clusters. IDBSCAN [1] divides the neighborhood of an object p into smaller quadrants. For each quadrant, IDBSCAN identifies its nearest object and only use this one as a seed for expanding the cluster. Though these techniques are fast, they can only coarsely approximate the results of DBSCAN as pointed out in [18]. AnyDBC follows a total different approach from these techniques. It consists of an unique *active learning* scheme for *iteratively* examining the cluster structure, and using the acquired knowledge for minimizing the number of required range queries for producing the same clustering result as DBSCAN at the end. Moreover, none of these techniques has an *anytime* property like AnyDBC.

Exact techniques. Most exact techniques for enhancing DBSCAN use grids. For example, Gunawan et al. [7] introduce an $O(nlogn)$ algorithm for clustering points in 2D by dividing the data space into grid cell of width $\epsilon/\sqrt{2}$. Gan et al. [6] significantly extend the work of [7] for higher dimensional data using the Bichromatic Closest Pair (BCP) problem for finding density-connected grid cells. For these methods, since the number of cells is exponential wrt. the dimensionality of the data, their performance significantly decreases when facing high dimensional data, while AnyDBC still works well as illustrated in Section 5. Due to their grid scheme, they are limited to vector data and the L_p metric and cannot deal with complex data. Moreover, they do not utilize knowledge of data and are not *anytime* techniques.

Complex data. In [2], a fast lower bound distance is used for guiding the clustering process in a complex database such as CADs and trajectories, thus reducing the total number of slow true distance calculations. This technique relies on a data structure called XSeedlist to sort objects according to their distances to processed objects. A-DBSCAN [11] and A-DBSCAN-XS [12] are anytime techniques that operate in multiple steps and rely on a sequence of LB distances and the monotonicity of cluster structures for enhancing performance. Act-DBSCAN [13] is an active clustering algorithm which aims at maximizing the clustering quality of DBSCAN under a pairwise distance calculation budget. AnyDBC can also be regarded as an active clustering algorithm. However, it does not focus on a pairwise distance budget like Act-DBSCAN but on minimizing the total numbers of queries instead. Similar to [2], all these techniques are mainly designed to cope with small datasets and expensive distance measures due to their high time and space complexity, while AnyDBC focuses on clustering large complex datasets. Moreover, AnyDBC do not require special conditions like LB distances [2,11,13] which may not be available.

Parallelism. AnyDBC consists of high degree parallelism steps, thus allowing the design of a scalable parallel method. The detail, however, is left out due to the space constraint.

7. CONCLUSION

Though DBSCAN is a fundamental clustering algorithm with many applications, its quadratic time complexity still remains a major challenge. AnyDBC proposes a unique approach to cope with this problem. The heart of AnyDBC is an *active learning* scheme that allows AnyDBC to actively learn the current cluster structure and to select only the most meaningful objects for performing range queries. As a result, it produces the same clustering result as DBSCAN with much fewer queries, thus significantly improving the performance. Moreover, its anytime property provides a very efficient approach for coping with large datasets. Experiments on real and synthetic datasets show that AnyDBC is orders of magnitudes faster than competing techniques, even if it is run until the end.

Acknowledgment We special thank Prof. Yufei Tao for providing us the binary file of DBSCANR and anonymous reviewers for their invaluable comments. Part of this research was funded by a Villum postdoc fellowship.

8. REFERENCES

[1] B. Borah and D. K. Bhattacharyya. An Improved Sampling-Based DBSCAN for Large Spatial Databases. *ICISIP*, 2004.

[2] S. Brecheisen, H. Kriegel, and M. Pfeifle. Efficient density-based clustering of complex objects. In *ICDM*, pages 43–50, 2004.

[3] L. Chen and R. T. Ng. On The Marriage of Lp-norms and Edit Distance. In *VLDB*, pages 792–803, 2004.

[4] M. Dash, H. Liu, and X. Xu. '1 + 1 > 2': Merging distance and density based clustering. In *DASFAA*, pages 32–39, 2001.

[5] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density- Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*, pages 226–231, 1996.

[6] J. Gan and Y. Tao. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *SIGMOD*, pages 519–530, 2015.

[7] A. Gunawan. A Faster Algorithm for DBSCAN. Msc thesis, TU Eindhoven, 2013.

[8] T. Kobayashi, M. Iwamura, T. Matsuda, and K. Kise. An Anytime Algorithm for Camera-Based Character Recognition. In *ICDAR*, pages 1140–1144, 2013.

[9] S. Mahran and K. Mahar. Using grid for accelerating density-based clustering. In *CIT*, pages 35–40, 2008.

[10] S. T. Mai, S. Goebl, and C. Plant. A Similarity Model and Segmentation Algorithm for White Matter Fiber Tracts. In *ICDM*, pages 1014–1019, 2012.

[11] S. T. Mai, X. He, J. Feng, and C. Böhm. Efficient Anytime Density-based Clustering. In *SDM*, pages 112–120, 2013.

[12] S. T. Mai, X. He, J. Feng, C. Plant, and C. Böhm. Anytime density-based clustering of complex data. *Knowl. Inf. Syst.*, 45(2):319–355, 2015.

[13] S. T. Mai, X. He, N. Hubig, C. Plant, and C. Böhm. Active Density-Based Clustering. In *ICDM*, pages 508–517, 2013.

[14] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. keng Liao, F. Manne, and A. N. Choudhary. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *SC*, page 62, 2012.

[15] B. Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[16] A. Tramacere and C. Vecchio. gamma-ray dbscan: a clustering algorithm applied to fermi-lat gamma-ray data. i. detection performances with real and simulated data. 2012.

[17] N. X. Vinh, J. Epps, and J. Bailey. Information theoretic measures for clusterings comparison: is a correction for chance necessary? In *ICML*, pages 1073–1080, 2009.

[18] X. Wang and H. J. Hamilton. DBRS: A Density-Based Spatial Clustering Method with Random Sampling. In *PAKDD*, pages 563–575, 2003.

[19] S. Zhou, A. Zhou, J. Cao, W. Jin, Y. Fan, and Y. Hu. Combining Sampling Technique with DBSCAN Algorithm for Clustering Large Spatial Databases. In *PAKDD*, pages 169–172, 2000.

[20] S. Zilberstein. Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, 17(3):73–83, 1996.