Incremental Density-Based Clustering on Multicore Processors

Son Mai, Jon Jacobsen, Sihem Amer-Yahia, Ivor Spence, Nhat-Phuong Tran, Ira Assent, Quoc Viet Hung Nguyen

▶ **To cite this version:**

Son Mai, Jon Jacobsen, Sihem Amer-Yahia, Ivor Spence, Nhat-Phuong Tran, et al.. Incremental Density-Based Clustering on Multicore Processors. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2022, 44 (3), pp.1338-1356. 10.1109/TPAMI.2020.3023125 . hal-04239831

HAL Id: hal-04239831

https://hal.science/hal-04239831v1

Submitted on 16 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Incremental Density-based Clustering on Multicore Processors

AUTHORS' COPY

Son T. Mai, Jon Jacobsen, Sihem Amer-Yahia, Ivor T. A. Spence, Nhat-Phuong Tran, Ira Assent, Quoc Viet Hung Nguyen

ABSTRACT

The density-based clustering algorithm is a fundamental data clustering technique with many real-world applications. However, when the database is frequently changed, how to effectively update clustering results rather than reclustering from scratch remains a challenging task. In this work, we introduce IncAnyDBC, a unique parallel incremental data clustering approach to deal with this problem. First, IncAnyDBC can process update in *bulks* rather than *batches* like state-of-the-art methods for reducing update overheads. Second, it keeps an underlying cluster structure called object node graph during the clustering process and use it as a basis for incrementally update clusters wrt. changes in databases by propagating changes around affected nodes only. In additional, IncAnyDBC *actively* and *iteratively* examine the graph and choose only a small set of most meaningful objects to produce exact clustering results of DBSCAN at the end as well as approximate results under arbitrary time constraints. And thus it is more efficient than other existing methods. Third, by processing objects in *blocks*, IncAnyDBC can be efficient parallelized on multicore CPUs, thus creating a *work-efficient* method. It runs much faster than existing techniques using one thread while still scaling well with multiple threads. Experiments are conducted on various large real datasets for demonstrating the performance of IncAnyDBC.

Keywords

Density-based clustering, anytime clustering, incremental clustering, active clustering

1. INTRODUCTION

Data clustering is a fundamental problem in exploratory data analysis and has many applications in different fields, e.g., data cleaning, data compression, machine learning, and pattern recognition [2, 25]. Given a dataset O, a clustering algorithm separates it into groups of similar objects. However, when objects are inserted into or deleted from O, how to efficiently update the results rather than reclustering from the scratch is an important research focus [5,16,45]. In many clustering methods, the cluster label of an object highly depends on many other ones, making an efficient cluster update process a challenging task [20]. One example is the density-based clustering algorithm DBSCAN [17], one of the most widely used data clustering methods with many real-world applications [30, 31, 38, 42].

In DBSCAN, a cluster is determined by a set of connected dense objects, and separated from other clusters by sparse areas. An object p is in a dense area if it has more than μ neighbors within a specific distance threshold ϵ. If it is, p is a *core* and its label will be propagated to all neighbors. This label propagation scheme of DBSCAN can be exploited for efficiently updating clusters due to the locality of changes as in IncDBSCAN [16]. For example, an inserted object may merge some existing clusters within its neighborhood. On the other hand, a deleted objects may break clusters into smaller pieces. In this case, these clusters need to be regrouped due to the label dependency of objects. Thus, in the worst case, the whole dataset will be affected, which is obviously as expensive as re-clustering from scratch. When the dataset and the number of inserted or deleted objects are large, this leads to significant computation effort and thus limits the applicability of the algorithm.

Contribution. In this work, we focus on an efficient approach for incremental updating clusters following the notion of DBSCAN. Our algorithms, called IncAnyDBC, have some uniques properties as follows.

First, before updating, existing techniques like Inc-DBSCAN [16] relies on the original DBSCAN algorithm [17] to group objects and determine their core properties. However, DBSCAN requires all neighborhood queries to be performed, which degrade its performance. It also does not keep enough information on the cluster structure to serve as a basis for efficiently update the clustering results when changes occur in the database. Our algorithm IncAnyDBC first summarizes objects into small density-connected groups called object nodes. These nodes and their connections are served as an underlying structure to predict the final clusters. Based on this information, IncAnyDBC repeatedly chooses a subset of objects to perform the neighborhood queries and connect nodes to build clusters until it finishes or is terminated by users. This *active clustering* scheme brings up some benefits: (1) IncAnyDBC can produce the same result as DBSCAN with fewer number of queries, thus enhancing performance; (2) it can be suppressed and resumed at any time to provide good approximate results (or exact results of DBSCAN

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

at the end), while most existing techniques can only produce a single approximate or exact result, e.g., [19,21]. This *anytime* property makes IncAnyDBC useful to system with limited time constraints or to cope with very large datasets; (3) since IncAnyDBC only builds clusters based on neighborhood queries, it can be used with arbitrary distance metrics instead of only Euclidean distance like state-of-the-art grid-based techniques such as [19,21,23]; and (4) the underlying node structure is preserved after the clustering and can be exploited to efficiently update the clusters after object insertions or deletions instead of reclustering from scratch.

Second, when there are changes in the data, existing techniques such as [16,20] update clusters in a *batch* mode (i.e., processing changes one-by-one). This scheme incurs many redundant overhead, especially when the number of changes is large. Our algorithm, in contrast, update clusters in a *bulk* mode (i.e., all changes at the same time). Hence, it reduces update overhead and thus is more efficient. During the updating process, the final cluster structure of IncAnyDBC is exploited to identify affected areas and to serve as a basis for building the final clustering results. Similar to the clustering process, clusters are rebuilt in an iterative way by letting the algorithm *actively* choose a subset of objects to query at each iteration. Thus, at the end, the clusters are updated with fewer number of queries compared to Inc-DBSCAN [16], thus making IncAnyDBC much efficient. Moreover, the *anytime* property is still guaranteed. Users can suspend and resume the updating process at any time for examining current results or looking for better ones. To the best of our knowledge, none existing incremental techniques for DBSCAN has this useful property. IncAnyDBC is also not restricted in low dimension Euclidean distance like other state-of-the-art techniques such as [20].

Third, by processing neighborhood queries in a block at each iteration to build clusters, IncAnyDBC can be efficiently parallelized on shared memory structures such as multicore CPUs. This makes it a *work-efficient* parallel method. It runs much faster than state-of-the-art sequential techniques such as DBSCAN [17] and IncDBSCAN [16] using one thread, while scaling very well with the total number of threads. Moreover, the *anytime* property still retains in the parallel mode, uniquely making IncAnyDBC both a parallel and an anytime method at the same time. To the best of our knowledge, IncAnyDBC is the first shared memory parallel approach for incrementally updating clusters in DBSCAN.

Summarization. To summarize, our major contributions are as follows:

- We introduce an efficient clustering algorithm for initializing cluster structures before updating. It uses much fewer number of queries to build the same clustering result as DBSCAN and thus is more efficient. Moreover, it can work under arbitrary time constraints due to its anytime property.

- We introduce an incremental scheme to update clusters wrt. changes in the data in a *bulk* mode rather than a sequential *batch* mode. Similar to the clustering phase, it has an efficient query pruning scheme and thus it is more efficient than existing techniques like IncDBSCAN. Moreover, the anytime property is also supported while updating clusters.

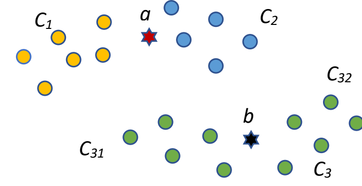

Figure 1: Incremental clustering: (1) the inserted object a merges two clusters C_1 and C_2 into a single cluster and (2) the deleted object b breaks cluster C_3 into two small clusters

- We propose a way to efficiently parallelizing IncAnyDBC on shared memory architectures such as multicore CPUs for further accelerating the performance.

To the best of our knowledge, IncAnyDBC is the first *work-efficient* and *anytime* parallel approach on multicore CPUs for incrementally update clusters in DBSCAN. Experiments are conducted on very large real and synthetic datasets for demonstrating the performance of our algorithms.

2. PRELIMINARY

Density-based clustering. The density-based clustering algorithm DBSCAN [17] separates each object into clusters based on the cardinality of its neighbors w.r.t. two given parameters $\mu \in \mathbb{N}^+$, $\epsilon \in \mathbb{R}^+$, and a distance function d.

DEFINITION 1. *(Neighborhood) The neighborhood of an object p, denoted as N_p, is the set of objects q where $d(p,q) \leq \epsilon$.*

DEFINITION 2. *(Core property) An object p is called a core object if $|N_p| \geq \mu$. Otherwise, if one of its neighbors is a core, p is called a border. If none of its neighbors are core, it is a noise.*

DEFINITION 3. *(Reachability) Given a core object p and an object $q \in N_p$, we say that q is density-reached from p, denoted as $p \triangleright q$.*

DEFINITION 4. *(Connectivity) Two object p and q are connected if there exists a sequence of core objects x_1 to x_n such that $p \triangleleft x_1 \triangleleft x_2 \cdots \triangleright x_n \triangleright q$, denoted as $p \bowtie q$.*

DEFINITION 5. *(Cluster) A cluster is a maximal set of density-connected objects.*

DBSCAN builds clusters by performing neighborhood queries on all objects to determine their core properties and chains of density-connected objects (or clusters). Thus, it has $O(n^2)$ complexity, where n is the number of objects. Note that, each core object belong to only one cluster, while a border object might be shared by some clusters.

Incremental DBSCAN. When there is a change (insertion or deletion), instead of re-building clusters from scratch, Ester et al. [16] introduce IncDBSCAN for incrementally update clusters by exploiting the locality of cluster structures as illustrated in Figure 1. Overall, there are two cases:

- **Insertion.** An inserted object may change a border or a noise object into a core one or may act as a core object to connect two density-connected sets. Thus, clusters may be merged or new clusters are raised from noise objects. E.g., the inserted object a merges clusters C_1 and C_2 into a cluster.

Figure 2: The general idea of IncAnyDBC

- **Deletion.** A deleted object may be a core or may change other core objects into a non-core ones. This causes clusters to be splited or vanished. E.g., the deleted object b breaks cluster C_3 into two smaller clusters C_{31} and C_{32}.

IncDBSCAN processes changes in a *batch* mode. For each inserted or deleted object, IncDBSCAN determines a set of objects that change their core property and uses them as seeds for update clusters locally, e.g., merge clusters. However, if a cluster may split, it needs to be fully reclustered from scratch. Each update takes $O(n^2)$ time complexity.

3. OUR PROPOSED ALGORITHM

We assume a database O of n arbitrary objects, grouped into arbitrarily shaped clusters by DBSCAN. Let B be a set of m changes on O including (1) insert an object into O, and (2) delete an object from O.

3.1 General idea

Our approach includes: (1) an efficient clustering algorithm that not only cluster the data effectively but can keep track off necessary information for updating the results later, (2) we want to incrementally update the cluster labels for all objects following the changes made in B rather than reclustering from scratch which is time consuming, (3) the algorithm can process data (clustering and updating) in an *anytime* fashion, and (4) an efficient parallel processing scheme for further improving the performance on multicore CPUs.

Figure 2 illustrates general ideas of IncAnyDBC. They include: summarization, active clustering, block processing, bulk processing, and parallel processing.

Summarization. IncAnyDBC first summarizes all objects into (possible overlapped) subgroups (called object nodes) using neighborhood queries. Each contains a neighborhood of an object (c.f. Definition 6), e.g., nodes v_p and v_q . If two nodes are close enough, they may belong to the same cluster and thus will be connected by an edge, e.g., the edge (v_p, v_q). These nodes and their connectivity will be served as an underlying structure to build clusters by connecting them into separated components since each node already is a part of a cluster (c.f. Lemma 1). This scheme brings up many different benefits: (1) it allows the algorithm to build clusters without having to perform all neighborhood queries, thus significantly improving performance; (2) it keeps track off necessary cluster structure to efficiently rebuild clusters after data changes.

Active clustering. Instead of performing all queries like DBSCAN, IncAnyDBC iteratively examines the current node structure and choose most meaningful objects to execute queries and to build clusters. E.g., if we choose h, it will connect two nodes v_p and v_q into a cluster if it is a core (c.f. Lemma 2). However, if we choose g, it will not help to clarify clusters regardless its core property and thus g can be safely ignored. Consequently, IncAnyDBC can produce the same clustering result as DBSCAN with fewer neighborhood queries and thus it is much efficient than DBSCAN.

Block processing. At each iteration, IncAnyDBC chooses a set of objects to perform queries instead of a single object. This scheme trades off between the cost of repeatedly examine and choose objects in the active clustering scheme above and the number of used queries, thus bringing up better performance. Moreover, it can be exploited to create an efficient parallel algorithm as discussed below.

Incremental processing. The node structure can be exploited to effectively update clusters. When there are changes, the first step is to update the current node structure by adding new objects or removing deleted objects from current object nodes. Then, we need to update the connectivity among affected nodes before updating clusters. E.g., the new inserted object a will be absorbed into the existing node v_p since $d(p, a) \leq \epsilon$. Since node v_p has a new object, its connections to other nodes may be changed (c.f. Lemma 2). However, other connections will not be affected and thus can be ignored. Thus, we can limit the updated area by v_p and its surrounding nodes only (denoted by red edges) for reducing computation cost.

Bulk processing. Instead of updating clusters with each object change, we propose to process in a bulk scheme. By this way, all possible changes in the node structure will be captured and will be updated at the same time, which is much efficient. This scheme fits well with the summarization approach of IncAnyDBC.

Parallel processing. The underlying node structure and block processing scheme allows us to design an efficient parallel technique. The general idea is processing each queries in a block independently with each other. The results can be stored in a buffer (for saving memory) and then be used for construct clusters. This scheme reduces the synchronization costs and the sequential costs while propagating cluster labels among nodes since the number of nodes is much smaller than the number of objects.

3.2 The algorithm IncAnyDBC

Generally, the algorithm IncAnyDBC consisting of several steps. In Step 1, objects are summarized into small (perhaps overlapped) object nodes representing subclusters of density connected objects. In Step 2, we built a graph $G = (V, E)$ where each vertex is a node and each edge represent the connectivity statuses of these nodes. Step 3 checks if the algorithm must continue. In Step 4, some objects are selected to perform neighborhood queries. In Step 5, IncAnyDBC updates the graph G according to the new changes. Step 3 to 5 is repeated until a termination condition is reached. Then, the final Step 6 looks for the remaining border objects. The pseudocode for IncAnyDBC is summarized in Figure 3.

Step 1: Summarization. In the beginning, all objects are assigned an *untouched* initial state indicating that they have not been processed in any ways. Since we only use a subset of objects to build clusters iteratively, the state of each object p changes accordingly and is summarized in Figure 4. E.g., an object p has *processed-core* (denoted as *pcore*) state indicating that we already performed a neighborhood query

```
1   function C = IncAnyDBC (O, d, μ, ε, α, β)
2   input:    dataset O, distance function d, parameters μ, ε of
3             DBSCAN, the query block sizes α, β
4   output:   the final clustering result C
5   begin
6       /* Step 1: Summarization */
7       while there are untouched objects do
8           select a set B of α untouched objects to query
9           for each object p in B do
10              perform query and mark st(p)
11              mark st(q) and increase nei(q) where q ϵ N_p
12              put p into V or L based on its core property
13      build a list V_p of nodes for each object p
14      /* Step 2: Build connectivity graph */
15      connect pairs of nodes (v_p, v_q) if d(p, q) ≤ 3ε
16      for each object p in O do
17          if p is a core then set the yes states for edges in V_p
18          else set the weak state for edges in V_p
19      /* Iteratively update clustering results */
20      while true do
21          /* Step 3: Check stopping condition */
22          label nodes via their yes connected components
23          for each cross-edge (v_p, v_q) do
24              if st(v_p, v_q) is unknown, weak, or link then cont = true
25          if cont = false then break
26          /* Step 4: Select objects for querying */
27          calculate node degrees for all nodes
28          calculate object scores for all unprocessed objects
29          chose a set B of β top scores objects
30          /* Step 5: Update graphs */
31          perform queries for all objects in B, mark the state, and
32                  increase the neighborhood counts for objects
33          for each new core object p do
34              set the yes state for edges in V_p
35          for each core object p in B do
36              for each object q in N_p do
37                  if q is a core then st(V_p[1], V_q[1]) = yes
38                  else st(V_p[1], V_q[1]) = link
39          for each cross-edge (v_p, v_q) do
40              if st(v_p, v_q) = weak or unknown then
41                  if usize(v_p) = 0 ∨ usize(v_q) = 0 then st(v_p, v_q) = no
42              else if st(v_p, v_q) = link then
43                  if usize(v_p) = 0 ∧ usize(v_q) = 0 then st(v_p, v_q) = no
44      /* Step 6: Check the noise list */
45      for each object p in L do
46          check if p is a border object
47  end
```

Figure 3: Pseudocode for IncAnyDBC

on p and it is a core. If we have not performed a query on p but we knew it is a core, p is assigned an *unprocessed-core* (denoted as *ucore*) state. Each arrow shows the state transition of an object p during the clustering process. E.g., if we perform a query on an *untouched* state and it is not a core, we mark it as *processed-noise* (*pnoise*). However, in the next iterations, if one of its neighbors is a core, p is a border and it state will be changed to *processed-border* (*pborder*).

We also store the number of known neighbors for each object p, denoted as $nei(p)$, for determining the core property of p. Beside that, we assign for p a special number called the database level, denoted as $lev(p)$, which is specially used to guarantee the consistence of the neighborhood counts in the insertion and deletion modes presented in Section 3.3.

At each iteration, IncAnyDBC randomly chooses a set S of α *untouched* objects and queries their neighbors. If $p \in S$ is a core, we create a node $v_p \in V$ consisting of N_p and represented by p (cf. Definition 6). In additional, we set

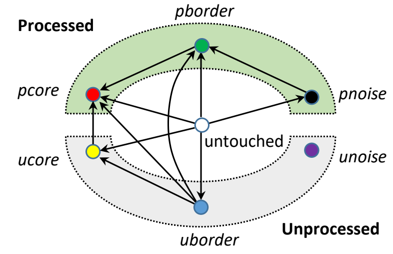

Figure 4: The state transitions of objects

the current state of p ($st(p)$) as a processed core ($pcore$). Otherwise, we set p as a processed noise ($pnoise$) and stores p and N_p into a special list called the noise list L for the post processing step in Step ???. Moreover, we set $lev(p) = n$ stating that p is processed when the database has n objects.

DEFINITION 6. *(Object node). An object node $v_p \in V$ consists of the object p as a representative and all of its neighbors in N_p.*

LEMMA 1. *([37]) All objects inside v_p are density-connected, i.e., belong to the same cluster.*

For each object $q \in N_p$, we set its state following the transition scheme for objects summarized in Figure 4 and Lemma 1. Concretely, if q is *untouched*, it will be changed to *uborder* if $nei(q) < \mu$ or *ucore* if $nei(q) \geq \mu$. If q is *pnoise*, it becomes *pborder*. Otherwise, $st(q)$ remains unchanged. Moreover, if q is not processed, we increase it neighbor count $nei(q)$ by 1, since q has p as its neighbor.

Step 1 end when there is no *untouched* objects left. At the end of Step 1, we build for each object p a list of nodes containing it, denoted as V_p. Since each core object belongs to only one cluster in DBSCAN, each node also belongs to one cluster following its representative (though its non-core members may be shared among different clusters). Thus, instead of labeling each object as in DBSCAN, we only need to label each node in V. The label of an object will be acquired from the node containing it following Lemma 1.

In the next Steps, IncAnyDBC performs additional queries on *unprocessed* objects to connect nodes to form clusters.

DEFINITION 7. *(Directly-connected). Two nodes v_p and v_q are directly connected, denoted as $v_p \Leftrightarrow v_q$, if there exists a set of objects $x_i \in N_p \cup N_q$ so that $p \lhd x_1 \cdots x_m \rhd q$.*

Following Definition 4 and 5, if $v_p \Leftrightarrow v_q$, they belong to the same cluster. There are two connect (merge) cases in IncAnyDBC, either via a shared core objects or a link between their two core objects as described in Lemma 2.

LEMMA 2. *Two nodes v_p and v_q are directly connected if:*

- *Case A: they share an object a where $st(a) = \text{ucore}$ or $st(a) = \text{pcore}$ (or core for simplicity).*
- *Case B: there exist two core objects $a \in N_p$ and $b \in N_q$ such that $d(a, b) \leq \epsilon$.*

PROOF. Case A: We have $p \lhd a$ and $a \lhd q$ (Definition 3). Thus, $p \lhd a \rhd q$. Case B: We have $p \lhd a$ and $b \rhd q$ (Definition 3). Since a and b are core and $d(a, b) \leq \epsilon$, we have $a \bowtie b$. Thus, $p \lhd a \rhd b \rhd q$. Thus, $p \Leftrightarrow q$ (Definition 7). $\square$

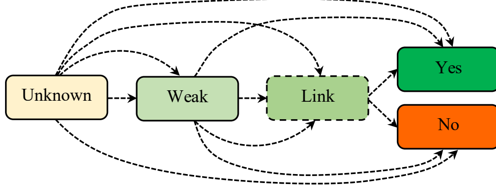

Figure 5: The state transitions of edges

Step 2: Build the connectivity graph. In this step, we create a graph G that captures all possible merges among nodes w.r.t. additional queries.

LEMMA 3. *Given two nodes v_p and v_q, if $d(p,q) > 3\epsilon$, v_p and v_q will never be directly connected.*

PROOF. Let a and b be two arbitrary objects in N_p and N_q, respectively. Due to the triangle inequality, we have $d(p,q) \leq d(p,a) + d(q,b) + d(a,b)$. Since $d(p,a) \leq \epsilon$ and $d(q,a) \leq \epsilon$ (Definition 6), we have $d(a,b) > \epsilon$. Thus, v_p and v_q will not be directly-connected (Definition 7). □

Following Lemma 3, we initialize the graph G by creating an edge (v_p, v_q) between v_p and v_q in V if $d(p,q) \leq 3\epsilon$. The graph G roughly captures the cluster structure of the data following Lemma 4 described below.

LEMMA 4. *If two core objects a and b are density-connected in DBSCAN, there exists a path of nodes in G that connects v_p and v_q, where $a \in v_p$ and $b \in v_q$.*

PROOF. Let $a = x_1 \triangleleft x_2 \cdots \triangleright x_n = b$ be a chain of core objects connecting a and b (Definition 4). After Step 1, if x_i is core, it must be covered inside a node v_j of V. Since $d(x_i, x_{i+1}) \leq \epsilon$, their nodes will be connected in G. □

Each edge represents a pair of nodes that may be directly-connected wrt. additional queries. For each edge (v_p, v_q) of G, we assign a state reflexing the connectivity status of two nodes v_p and v_q.

DEFINITION 8. *(Edge state). The state of an edge (v_p, v_q), denoted as $st(v_p, v_q)$, captures the connectivity status of v_p and v_q. If $st(v_p, v_q)$ = unknown, we do not know if v_p and v_q are directly-connected or not. If v_p and v_q share an object, $st(v_p, v_q)$ = weak meaning that they are more likely to be directly-connected. If $st(v_p, v_q)$ = yes, v_p and v_q are directly-connected and are in the same cluster. If $st(v_p, v_q)$ = no, v_p and v_q will never be directly-connected.*

During the operation of IncAnyDBC, the states of edges change as summarized in Figure 5. The *link* state is a special trick of IncAnyDBC and will be explained in Step 5. Note that, the *no* state does not mean that v_p and v_q are not in the same cluster. They may be connected via a chain of directly-connected nodes.

At the end of Step 2, we update the states of edges. Following Lemma 2, if p is a core (either *ucore* or *pcore*), all nodes in V_p will belong to the same cluster. For each edge $(V_p[i], V_p[i-1])$ where $V_p[i]$ is the node at position i of V_p, we set its state to *yes*, meaning that they will be in the same cluster. If p is not a core, we do not know that all nodes in V_p are in the same clusters or not. But, since they overlap, they have higher chances to be. Thus, we assign for each edge $(V_p[i], V_p[i-1])$ the *weak* state. Note that we do not need to change all edges among nodes in V_p for avoiding computational overheads.

Step 3: Check stopping condition. At the beginning of Step 3, we label all nodes of G by finding connected components of *yes* edges of G. If two nodes v_p and v_q belong to the same connected component, they are in the same cluster following Definition 7. Let $label(v_p)$ be the current cluster label of a node v_p.

DEFINITION 9. *(Cross-edge). If an edge $(v_p, v_q) \in E$ has $label(p) \neq label(q)$, it is called a* cross-edge *since it connects two different clusters.*

LEMMA 5. *If there is a* cross-edge *(v_p, v_q) where $st(v_p, v_q) \in \{\text{weak}, \text{unknown}, \text{link}\}$, the cluster structure may change.*

PROOF. Since v_p and v_q have different labels, $st(v_p, v_q) \neq$ *yes*. If $st(v_p, v_q)$ = *no*, they will never be directly-connected. Thus, the cluster structure may only change if $st(v_p, v_q)$ is *weak*, *unknown*, or *link*, since it may be changed to *yes* wrt. new queries, leading to the merge of two clusters. □

Following Lemma 5, we scan through all edges of G looking for *weak*, *unknown*, or *link cross-edges*. If they exist, the algorithm should continue. Otherwise, IncAnyDBC can be stopped since the clustering result will not change regardless of any other queries.

Step 4: Select objects for querying. The general purpose of this step is too select *unprocessed* objects for processing so that the clusters are formed quickly, i.e., more nodes to be connected at each iteration. At the same time, we want the algorithm to be terminated as quick as possible to ensure the final performance. To do so, the graph G is used as a basis for ranking objects based on their impact on the changes of the current cluster structure.

DEFINITION 10. *(Node degree). The degree of a node v_p, denoted as $deg(v_p)$, is defined as follows:*

$$deg(v_p) = \sum_{v_q \in adj(v_p)} \omega(v_p, v_q) stat(v_q)$$

where $adj(v_p)$ is the set of adjacent nodes v_q of v_p where $label(v_q) \neq label(v_p)$ (i.e., (v_p, v_q) is a cross-edge*); $w(v_p, v_q)$ is the predefined weight for each edge based on its state (1 if $st(v_p, v_q)$ = unknown, 2 if $st(v_p, v_q)$ = link, and 4 if $st(v_p, v_q)$ = weak); $stat(v_p)$ is the current processing score of v_p and is defined as:*

$$stat(v_p) = (1 - \frac{|N_p|}{n}) + \frac{usize(v_p)}{|N_p|} + \psi(v_p)$$

where $usize(v_p)$ is the number of unprocessed objects of N_p and $\psi(v_p) = 1$ if v_p consists of an pborder *object and $\psi(v_p) = 0$ otherwise.*

The degree of a node v_p ($deg(v_p)$) measure the uncertainty of v_p wrt. the current structure. If v_p is lying closer to borders of many different clusters (it has larger $|adj(v_p)|$ or contains a processed border object), its label is more uncertain than one lying deep inside a cluster. Thus, if we perform a query on $q \in v_p$, it will connect more nodes following Lemma 2 or will break some undetermined edges faster following Lemma 6. Besides that, if $st(v_p, v_q)$ = *weak*, p and q have stronger influence to each other than *unknown* state. Thus, we assign a higher weight for *weak* edges. Moreover,

for each node p, we assign higher *stat* score for p if $|N_p|$ is small since it is more likely to be a border node. We also prefer nodes that contains fewer unprocessed objects since fully processing them will break undetermined edges, making IncAnyDBC converge faster following Lemma 5.

DEFINITION 11. *(Object score). The score of an object p, denoted as score(p), is defined as follows:*

$$score(p) = \sum_{v_q \in V_p} deg(v_q)$$

Similar to the node degree, higher $score(p)$ means that p is in a highly uncertain area (covered by uncertain nodes). Thus, processing it first may bring bigger changes to the cluster structure. Moreover, we prefer object with lower number of neighbors since its core property is more uncertain than the higher one.

At the end of Step 4, we choose a set B of β objects with highest scores for processing in Step 5.

Step 5: Update graphs. In the beginning of this step, we performing queries on all β selected objects. For each object p, we mark its core properties as *pcore* if it a core or *pborder* otherwise, and set $lev(p) = n$. We also increase the number of neighbors $nei(q)$ for each unprocessed object $q \in N_p$. We also set new states for all objects $q \in N_p$ following the transition states of objects in Figure 4.

Following Lemma 2 Case A, for each new *ucore* or *pcore* object $p \in O$, all nodes in V_p will be directly-connected. Thus, for each edge $(V_p[i], V_p[i+1])$, we set its state to *yes*.

For each core object $p \in B$, we need to check if p connect its node to other nodes via its neighbors following Lemma 2 Case B. To do so, we scan through each object $q \in N_p$. If q is *pcore* or *ucore*, all nodes in V_p and V_q are directly connected. However, we only need to set edge $(V_p[1], V_q[1])$ as *yes* (since V_p and V_q are processed in Case A above). If q is *uborder*, the connection may be available if future queries reveal that q is a core. Thus, we set for edge $(V_p[1], V_q[1])$ a special state called *link*, indicating that they are more likely to be connected via a pair of core objects.

LEMMA 6. *Given a* cross-edge (v_p, v_q) *where* $st(v_p, v_q) \in \{\text{weak, unknown, link}\}$*, if* $usize(v_p) = 0 \wedge usize(v_q) = 0$*,* v_p *and* v_q *will never be directly-connected, i.e.,* $st(v_p, v_q) = no$*, where* $usize(v_p)$ *is the number of unprocessed objects of* v_p*.*

PROOF. Assume that their exists a chain of objects $x_i \in v_p \cup v_q$ so that $p \lhd x_1 \cdots x_m \rhd q$. Since all x_i are *pcore*, v_p and v_q will belong to the same cluster following Lemma 2. It leads to contradiction since $label(p) \neq label(q)$. ☐

Optimization. Following Lemma 6, we need to perform all queries on their objects to break a *cross-edge* (v_p, v_q) into *no* state if v_p and v_q finally belong to different clusters. When there are many of such *cross-edge* between two clusters, redundant queries may occur, making IncAnyDBC converge slower following Lemma 5. Thus, IncAnyDBC uses a special trick to reduce the number of required queries.

For each *weak* or *unknown cross-edge* (v_p, v_q), if $usize(v_p) = 0$ or $usize(v_q) = 0$, we set $st(v_p, v_q) = no$ (even though v_p and v_q may be directly-connected if more queries are performed). However, if $st(v_p, v_q) = link$, we only change it to *no* if both nodes are fully processed. We need to prove that this scheme still guarantees a correct clustering result.

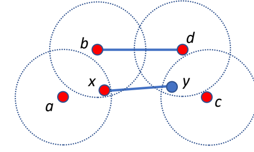

Figure 6: Illustration of Lemma 7

LEMMA 7. *Assume that* (v_a, v_c) *is a* cross-edge *at the current iteration of IncAnyDBC (but a and c belong to the same cluster in DBSCAN). We prove that when IncAnyDBC stops,* v_a *and* v_c *will be put in the same cluster.*

PROOF. (Sketch) Wlog., we assume that $usize(v_a) = 0$ and hence $st(v_a, v_c) = no$. There must exist a pair of object $x \in N_a$ and $y \in N_c$ so that $d(x, y) \leq \epsilon$ and $st(x) = pcore \wedge st(y) \neq pborder$ (y is actually a core in DBSCAN).

If $st(y)$ is core, v_a and v_c are put into the same connected component. Thus, $label(v_a) = label(v_c)$.

If $st(y) = uborder$, a *link* state is set to a pair of nodes containing x and y. Assume that $st(v_a, v_c)$ is *link*, IncAnyDBC cannot stop until (v_a, v_c) is not a *cross-edge* or they are fully processed. In both ways, v_a and v_c are finally in the same cluster. Assume that the *link* state is assigned to (v_b, v_d) as illustrated in Figure 6, where $x \in v_b$ and $y \in v_d$. If $label(b) = label(d)$, v_a, v_b, and v_d are in the same cluster (since x is core). Thus, if $label(c) = label(d)$, we have $label(v_a) = label(v_c)$. Otherwise, (v_c, v_d) is a *cross-edge*. In the worst case, v_c and v_d are fully processed, revealing that y is core. Hence, v_a, v_b, v_c, v_d will be in the same connected component, making $label(v_a) = label(v_c)$. If $label(b) \neq label(d)$, (v_b, v_d) is a *cross-edge*. Thus, in the worst case, v_b and v_d are fully processed. Processing y will connect v_a and v_c together as the above case. The other cases are proven similarly. ☐

According to Lemma 7, if two nodes v_p and v_q belong to the same cluster, it will be detected correctly by IncAnyDBC, though (v_a, v_c) may be assigned as *no* due to the optimization process described above.

At the end of Step 5, IncAnyDBC goes back to Step 3 to check if it should stop. If not, it chooses another set of objects to process until the termination condition is reached.

Step 6: Check the noise list. This post-processing step of IncAnyDBC checks the noise list L to find remaining border objects. For each object $p \in L$ with *pnoise* state, if there is a core object $q \in N_p$, p will be a border object and is assigned the same label as q. Otherwise, we need to perform a query on each *uborder* object $q \in N_p$ and set $level(q)$ to n until we find a core to assign p to. If there is no core found, p is surely a noise.

Correctness. When it reaches its final stage, IncAnyDBC produces the identical results as DBSCAN, except for shared border objects. They may be labeled differently in both DBSCAN and IncAnyDBC according to the examining order of objects.

LEMMA 8. *IncAnyDBC produces the identical final clustering results as DBSCAN.*

PROOF. (Sketch) If two core objects $a \bowtie b$ in DBSCAN, there exists a path of nodes $v_1 \cdots v_m$ in G such that $a \in v_1$ and $b \in v_m$ (Lemma 4). Due to Lemma 7, all nodes v_i belong to the same clusters in IncAnyDBC. Hence, $label(a) = label(b)$ (Lemma 1). Moreover, if $a \not\bowtie b$ in DBSCAN, for

each chain of core objects $a = x_1 \cdots x_m = b$, there exists a pair x_i and x_{i+1} where $d(x_i, x_{i+1}) > \epsilon$. Moreover, $x_i \in v_p$ and $x_{i+1} \in v_q$ where $p \neq q$ (otherwise $a \bowtie b$). Thus, v_p and v_q will never be directly-connected. Consequently, $label(a) \neq label(b)$ in IncAnyDBC. Step 6 of IncAnyDBC ensures that we do not miss any border objects. Thus, the results of IncAnyDBC and DBSCAN are identical. □

Monotonicity. Since IncAnyDBC merges nodes with new queries, the number of clusters will decrease at each iteration. This is useful when we want to track the progress of IncAnyDBC. E.g., if the number of clusters remains stable at some iterations, we can stop IncAnyDBC for saving runtime, while having almost the same clustering result as DBSCAN.

Complexity. Let $v = |V|$, $e = |E|$, and $l = |L|$. Step 1 needs $O((v + l)n)$ for neighborhood queries and $O(vn + l\mu)$ for marking object states. Step 2 consumes $O(v^2)$ for building G and $O(vn)$ for updating edges. Step 3 uses $O(v^2)$ for connected component finding and $O(e)$ for checking termination condition. Step 4 requires $O(e)$ for node degree calculations, $O(vn)$ for calculating object scores, and $O(n \log n)$ for sorting objects. Step 5 spends $O(\beta n)$ time for queries, $O(vn + \beta n)$ for updating *yes* edges, and $O(e)$ for *no* edges. Step 6 needs $O(l\mu n)$ time for querying neighbors. Overall, IncAnyDBC has $O(vn + ln + v^2 + \frac{n}{\beta}(v^2 + e + vn + n \log n + \beta n) + l\mu n)$ time complexity. In the worst case, v and l are $O(n)$ and $e = v^2$ (roughly speaking). And let $\beta = O(n)$. The time complexity of IncAnyDBC is $O(n^2)$, which is similar to that of DBSCAN.

IncAnyDBC needs to store all nodes, the graph G and the noise list L. Thus, its space complexity is $O(vn + v^2 + l\mu)$.

3.3 Dynamic cluster update

Reverse query. In IncAnyDBC, we have two different kinds of neighborhood queries. The first one is the normal query where we find the neighbors for an object p on the whole database O. The second one is called *reverse-query*, where we only perform neighborhood queries on the set B of inserted or deleted objects. Since $m \ll n$, the *reverse-query* is much faster than the normal query. Let M_p be the neighbors of p under a *reverse-query*.

3.3.1 Insertion

In the insertion case, the clusters will be merged into bigger ones or new clusters are raised as described above 2. IncAnyDBC first updates the current noise list L and the node list V in Step 1 and 2. Then it creates new nodes for new objects if it is necessary in Step 3. The graph G is updated for reflexing changes in Step 4. And the clusters are updated in Step 5. Figure **??** shows the pseudocode of the insertion case.

Step I0: Preparation. Before updating clusters, we mark the state of each new object as *untouched*. Each object o is assigned a flag called *ptou* indicating that it is in *processed* state but may change to *unprocessed* due to inserted objects.

Step I1: Update the noise list. Some noise objects may become core ones if new objects come into its their neighborhoods. Thus, for each object $p \in L$, we perform a *reverse-query* on p, looking for new objects in its neighborhood. If $nei(p) + |M_p| \geq \mu$, p becomes a *pcore*. We also mark $st(q) = uborder$ for $q \in N_p \cup M_p$ if q is a new object and *pborder* if q is an old object. In both cases, we update the neighborhood for p as $N_p \cup M_p$, increase the number of neighbors $nei(q)$ by 1 for new object $q \in M_p$, set the database level $level(p) = m + n$ (since the database has m objects more), and set $ptou(p) = 0$ (since p is surely a processed objects after insertions). At the end, we remove p and put it into the set V of nodes if it is a core.

Step I2: Update the node list. Similar to the noise list, some existing nodes in G may change wrt. new inserted objects and thus need to be updated. Hence, for each node $v_p \in V$, if $M_p \neq \emptyset$, we add M_p into the neighbor set N_p of p and update $nei(q)$ for $q \in M_p$. We also set $level(p) = m + n$ and $ptou(p) = 0$ like Step I1.

Step I3: Create new nodes. After Step I2, some new objects have been covered in new nodes or existing nodes. Some remain outside with the *untouched* state. We need to cover these objects inside nodes.

Similar to Step 1 of IncAnyDBC (Section 3.2), we repeatedly choose a set A of α *untouched* new objects. For each object p in A, we perform a range query on p, if $N_p \geq \mu$, we set $st(p) = pcore$ and put v_p to V. Otherwise, $st(p) = pnoise$. Now, we increase the number of neighbors $nei(q)$ for $q \in N_p$ only if $level(q) < n + m$. Here, the database level ensures that $nei(q)$ is correctly recorded since some currently processed objects have been checked without new inserted objects by IncAnyDBC during its clustering phase.

Step I4: Connect new nodes into G. Let V^N be the set of new nodes created in Step 1 and 3. We need to determine their relationships with other nodes. Following Lemma 3, for each node $v_p \in V^N$, if $d(p, q) \leq 3\epsilon$, where $v_q \in V$, we add an edge (v_p, v_q) into the edge set E of G, indicating that they can be directly-connected. We also temporarily set $st(v_p, v_q) = no$ (it will be fixed later).

Step I5: Identify change core nodes. At the end of Step I3, all new objects are either inside nodes or in the noise list. Let V^1 be the set of nodes that contain new objects and L^1 be the set of new objects in L. Let O^A be the set of objects in $\cup_{v_p \in V^1} adj(v_p)$ and $\cup_{p \in L_1} N_p$, i.e., all objects inside nodes with new objects and its adjacency and inside the new none-core nodes.

LEMMA 9. *All* processed *objects $o \notin O^A$ will not change their core properties after the insertion by new objects.*

Lemma 9 is directly referred from the triangular inequality in Lemma 3. All *processed* non-core objects in O^1 may change to core ones due to the new inserted objects. Thus, for each *pnoise* or *pborder* object $o \in O^A$ (that has not been processed in Step I1, i.e., $ptou(o) = 1$), if $nei(o) \geq \mu$, we mark it as a changed object. Otherwise, we perform a reverse query on o to check if o is a core (if $nei(o) + |M_o| \geq \mu$), set $ptou(o) = 0$, and $level(o) = n + m$. For each object $p \in M_o$, we update $nei(p)$ if $nei(p) < n + m$.

For each change core objects, we add their nodes to the list of change core nodes V^2 together with all newly created nodes in Step I1 (since they contain change core objects).

Step I6: Fix the core property of objects. Due to additional range queries in Step I1, I2, I3 and I5, the core properties of objects may change and need to be updated. For each old object $o \in O^A$, if $ptou(o) = 1$, o may change from *processed* to *unprocessed* states. If $st(p) \neq pcore$ and $nei(p) \geq \mu$, we change it to *ucore*. Otherwise, it state remains unchanged. If $st(p) = pcore$, we change it to *ucore*.

If o is not processed before insertions or has been updated ($ptou(o) = 0$), $st(o) = uborder$, and $nei(o) \geq \mu$, we set $st(o) = ucore$.

If $o \notin O^A$, $st(o)$ will not change (Lemma 9). Thus, we set $level(o) = n + m$ and $ptou(o) = 0$ if p is in processed states.

Step I7: Update cluster structures. For each affected object $o \in O^A$, we update the graph G by setting a *yes* connection among pairs of nodes in V_o as in Step 5 of IncAnyDBC. Then, we update the labels of nodes following their connected components of *yes* edges like Step 3 of IncAnyDBC.

Due to the merge of clusters, some existing *cross-edges*with the *no* state will be changed. Let $V^A = V^1 \cup V^2$ be the sets of nodes with new objects and change cores objects.

LEMMA 10. *For each* cross-edge (v_p, v_q), *if* $v_p \notin V^A \wedge v_q \notin V^A$, $st(v_p, v_q)$ *will not be affected by new objects.*

PROOF. (Sketch) Since (v_p, v_q) are a *cross-edge* before insertion, $st(v_p, v_q)$ must be *no*. And, there is no pair of core objects that will connect them as described in Lemma 2. Thus, edge $st(v_p, v_q)$ only changes if there is a new object coming into them or a processed border object inside v_p or v_q becomes a core object. These objects possibly become shared core objects or create a core-core link between them, causing the state changes. Note that an *uborder* object in v_p or v_q does not contributed to the connectivity. Thus if it becomes a core, it will not cause any change. □

Following Lemma 10, for updating clusters, an obvious way is reseting all possible edges related to V^A back into the *unknown* state and taking all objects inside nodes of V^A and its adjacency nodes to rebuild the connections among them. However, this still incurs redundant queries as shown in Figure 2. Since v_v and v_p already belong to the same clusters, examining (v_v, v_p) will not lead to any changes in the result. Thus, we follow a more efficient way as following. The general idea is reducing the total number of nodes and links that need to be examined. Consequently, this saves unnecessary queries, thus improving the performance.

Step I8: Fix the links in G following new objects. Let V^{1A} be the set of nodes v_q where (v_p, v_q) is a *cross-edge* and $v_p \in V^1$.

LEMMA 11. *Given a new object* $a \in v_p$, *if* $d(a, q) > 2\epsilon$, *a does not change the state of* (v_p, v_q).

For finding exactly nodes will be affected by new objects, for each node $v_q \in V^{1A}$, we perform a *reverse-query* on q with threshold 2ϵ (as demonstrated in Figure **??**). If $M_q^{2\epsilon}$ does not contain a core or or *uborder* new object, $st(v_p, v_q)$ will obviously not be affected and can be excluded from V^{1A}. Similarly, we remove a node from V^1 if it has no *cross-edge* counter parts in V^{1A}.

Let O^{1A} be the set of objects in V^{1A} (exclude *pnoise* and *pborder* due to no contribution and change core objects (will be processed later)). For each object $o \in O^{1A}$, we perform a reverse query on o to get new neighbors M_o. These neighbors will be used to limit the involved nodes.

Before further processing, we need to update the core properties of objects due to new queries. For each object $o \in O^{1A}$, if $level(o) = n$, we update the numbers of neighbors for o by increasing $nei(o)$ and $nei(p)$ for each $p \in M_o$ with $level(p) < n + m$. If $nei(p) \geq \mu$ and $st(p) = uborder$, we change $st(p)$ to *ucore*. Similarly, if $nei(o) \geq \mu$, we assign $st(o) = ucore$. And set $level(o) = n + m$ to update its query information. For each new core objects, we set the *yes* connections among its nodes following Lemma 2.

For each $o \in O^{1A}$ and for each object $p \in M_o$, if o and p are core, we set a *yes* connection between two nodes in $V_o[1]$ and $V_p[1]$ following Lemma 2. Otherwise, if p is not *pborder* or *pnoise*, o and p may link these nodes together. Thus, we keep all nodes of V_o and V_p inside the sets V^1 and V^{1A} respectively. At the end, V^1 and V^{1A} contains only nodes that can cause the changes in cluster structures.

If there are new *yes* edges, we re-update the cluster labels to reduce the number of *cross-edges*. Then, for each *cross-edges* (v_p, v_q) where $v_p \in V^1$ and $v_q \in V^{1A}$, if $st(v_p, v_q) = no$, we set $st(v_p, v_q) = unknown$. This makes the algorithm to reupdate clusters following Lemma 5.

Step I9: Fix the links in G following change core objects. Similar to new objects, change core ones cause clusters to be merged as in Lemma **??**. Let V^{2A} be the set of nodes v_q where (v_p, v_q) is a *cross-edge* and $v_p \in V^2$. Let O^2 be the set of change core objects in V^2.

LEMMA 12. *For each object* $o \in O^2$ *and node* $v_p \in V^{2A}$, *if* $d(o, p) > 2\epsilon$, *o does not change the state of* (v_p, v_q).

Following Lemma 12, for each object $o \in O^2$ and $v_p \in V^{2A}$, if $d(o, p) \leq 2\epsilon$, we do not remove v_p from V^{2A} and o from O^2 since they may cause the cluster changes.

For each remaining object $o \in O^2$ and $p \in N_o$, if p is a core object, we set $st(V_o[1], V_p[1]) = yes$ following Lemma 2. If p is *uborder*, p and o may form a link later if p is truly a core. Thus, we do not remove nodes of V_o and V_p from V^2 and V^{2A}, respectively.

Similar to Step I8, we update the labels of nodes if a *yes* edge occurs above before changing each *cross-edge* (v_p, v_q) where $v_p \in V^2$ and $v_q \in V^{2A}$ to the *unknown* state if it is in the *no* state.

Step I10: Choose objects to be examined. Let $V^E = V^1 \cup V^{1A} \cup V^2 \cup V^{2A}$ be the set of nodes that may be merged and are detected in Steps I8 and I9. Let O^E be the set of objects in $v_p \in V^E$ (exclude node center, *pborder* and *pnoise* objects).

PROPOSITION 1. *We only need to examine objects in* O^E *to fully update clusters after the insertions.*

Proposition 1 can be seen directly from Lemma 10 and the Steps I8 and I9 described above. In these steps, edges that do not affected by new inserted objects will be excluded from the cluster update by keeping their states as *no*. Thus, V^E contains all nodes belong to changing edges. Following Lemma 2 and Lemma 6, we need to examine all objects in O^E to clarify these edges as *yes* or *no* ones.

Following Proposition 1, we remove the processed mark for each object $o \in O^E$ from *pcore* to *ucore*. This allows the algorithm to re-perform the query to connect nodes.

Step I11: Update the clusters. In this step, we update cluster structures by examining all unprocessed objects in O^E to connect nodes in the similar ways to Step 3 to 6 of IncAnyDBC in Section 3.2.

Concretely, at each iteration, we choose a set of β objects in O^E to perform queries by assessing their roles on the changes of cluster structures as in Step 4 of IncAnyDBC.

However, we calculate $usize(v_p)$ for each node v_p by objects inside O^E only. Then, for each selected object p, we perform the range queries on it and update the state and current number of neighbors for each *unprocessed* object $q \in N_p$ as in Step 5 of IncAnyDBC. Since there are new objects inserted into the database at different times, we only update $nei(q)$ if $q \geq level(p) \wedge p \geq level(q)$ for ensuring consistency. After that, we update the states of edges of G following the changes of objects as in Step 5 of IncAnyDBC. The process stops when the termination condition in Lemma 5 in Step 3 of IncAnyDBC is reached with new *usize* values of objects. Finally, a post processing step as in Step 6 of IncAnyDBC is performed to identify the remainder border objects.

Correctness. We prove that IncAnyDBC produces identical results to those of DBSCAN after insertions.

LEMMA 13. *IncAnyDBC produces identical results to those of DBSCAN after insertions.*

PROOF. (Sketch) Steps I1 to I3 guarantee that if a new object a is a core, it will be covered inside a node. Moreover, Steps I8 to I9 ensures that all possible changes in G wrt. new objects can be captured. And the result can be fully updated by following the set O^E as in Proposition 1. Thus, similar to Lemma 8, if two core object $a \in v_x$ and $b \in v_x$ are density-connected, v_x and v_y will be assigned the same label at the end and vice versa. And the post-processing process will assign the labels for border objects accordingly. □

Complexity. Steps I0 to I10 take $O(lm)$, $O(vm)$, $O(mn)$, $O(v^2)$, $O(v^2+vn+l\mu+nm)$, $O(n+m)$, $O(v^2+vn)$, $O(mv^2+vn+vm+mn+m^2)$, $O(v^2+vn+vm+mn+n^2)$, and $O(v^2+vn+vm)$, respectively. Step I11 has the similar time complexity as in Steps 3 to 5 of IncAnyDBC since $O^E = n+m$ in the worst cases. Thus, the overall time complexity of IncAnyDBC will be $O(mn^2)$ (roughly speaking) for inserting m objects. And it is similar to IncDBSCAN. IncAnyDBC consumes $O(vn+v^2+l\mu+nm)$ overall space complexity.

3.3.2 Deletion

In the deletion case, some objects may loose their core property, thus leading to the split of clusters [16].

Step D0: Preparation. Before updating clusters, each object o is assigned a flag called *ptou* indicating that it is in *processed* state but may change to *unprocessed* due to deleted objects.

Step D1: Update the non-core list. Objects in L will not change to core ones due to deleted objects. However, we need to clean their deleted neighbors. To do so, we scan through each object $p \in L$ and remove the deleted objects from its neighbors (or p itself). And we mark p as an updated object by setting $ptou(p) = 1$.

Step D2: Update the node list. In contrast to the non-core one, objects inside the node list V may loose their core property due to deleted objects. For each node $v_p \in V$, we remove deleted objects from its neighbors or v_p itself if p is deleted. If $N_p < \mu$, p is not a core anymore. We remove v_p from V and put it into the non-core list L. All edges related to v_p also need to be removed from the graph G. We mark p as updated objects ($ptou(p) = 0$). Let V^1 be the set of nodes that contains deleted objects.

Step D3: Fix orphan objects. Due to deleted nodes in step D2, some objects may become orphans since they are not covered inside any nodes or in the non-core list. And they need to be fixed. To do so, we first assign the *untouched* state for all those objects. And then, we repeat the below procedure until there is no *untouched* one.

At each iteration, we choose a set A of α *untouched* objects to perform queries. For each object $p \in A$, if $N_p < \mu$, we set $st(p) = pnoise$ if $st(p) = untouched$ or $st(p) = pborder$ if $st(p) = uborder$ and put p into L. Otherwise, we set $st(p) = pcore$ and put it into V. For each object $q \in N_p$, we assign $st(q) = uborder$ if p is a core and $st(q) = untouched$. We also increase the neighbor count $nei(q)$ by 1 if $q \geq level(p)$ and $p \geq level(q)$ and q has not been updated ($ptou(q) = 1$).

Step D4: Update the graph. Let V^3 be the set of new nodes created in step D3. For each node $v_p \in V^3$, we add an edge to other node v_q if $d(p,q) \leq 3\epsilon$ following Lemma 3. Initially, we set $st(v_p, v_q) = no$.

Step D5: Fix the numbers of neighbors. Since deleted objects may be covered inside the neighborhoods of processed ones, we need to update the neighborhood count for some related objects. For each deleted object $p \in B$, we perform a range query on p. For each $q \in N_p$, we decrease $nei(q)$ by 1 if q may contribute to the neighborhood count of p before. This happens if q is a processed object and has not been updated or if q is an unprocessed one, and $q < level(p)$ or $level(q) > p$.

Step D6: Identify change core objects. Let $O^A = \cup_{p \in B} N_p$ where p is a deleted object. Obviously, all objects in O^A may change their core properties. Thus, for each object $o \in O^A$, we mark o as a change core one if $st(o) = pcore \vee ucore$ and $nei(p) < \mu$, and we put all nodes V_o of o into the set of change core nodes V^2.

Step D7: Fix the core properties. Since the numbers of neighbors change in steps D3 and D5, the core properties of objects must be fixed. We first extend O^A by adding objects in the adjacent nodes of the degenerated ones in step D2. Following Lemma 3, additional queries in step D3 only affects the neighbor counts for objects in O^A. Thus, for each object $o \in O^A$ if o is not updated, we fix its core property. If $st(o) = pcore$ and $nei(o) < \mu$, we change $st(o)$ to *pborder* if o is inside a node or *pnoise* otherwise. If $st(o) = uborder$ and $nei(o) \geq \mu$, $st(o) = ucore$. If $st(o) = ucore$ and $nei(o) < \mu$, $st(o) = uborder$.

Given a *yes* edge (v_p, v_q), if v_p or v_q is deleted in Step D2, the edge (v_p, v_q) will be deleted from G. Otherwise, (v_p, v_q) still remains but their *yes* state may loose if a core object is deleted from their neighbors following Lemma 2. This might cause the connected components of nodes to be broken, thus causing the splits of clusters. Let $V^A = V^1 \cup V^2 \cup V^3$ be the set of involved nodes. A simple approach would reset all their *yes* edges to the *unknown* states and rerun the clustering algorithm to re-build clusters. However, this still incurs redundant calculations. Thus, we follow a more efficient scheme as follows.

Step D8: Fix the links in G by deleted objects. Let V^{1A} be the set of nodes v_q where $st(v_p, v_q) = yes$ and $v_p \in V^1$.

LEMMA 14. *Given a deleted object $a \in v_p$, if $d(a,q) > 2\epsilon$, a does not break the* yes *state of (v_p, v_q).*

Following Lemma 14, we perform a reverse query on deleted objects for each $v_q \in V^{1A}$ with a threshold of 2ϵ. If $M_q^{2\epsilon}$

does not contain a deleted core object, the *yes* connection between v_p and v_q will not be affected by deleted objects. Thus, we remove v_q and its partner v_p from V^{1A} and V^1, respectively. At the end, for each edge (v_p, v_q) where $v_p \in V^1$ and $v_q \in V^{1A}$, we reset $st(v_p, v_q)$ to *unknown* state since it may be affected by the deletions.

Step D9: Fix the links in G by change core objects. If an object looses its core status, it may break the *yes* connection between its nodes and their adjacency. Let V^{2A} be the set of nodes v_q where $st(v_p, v_q) = yes$ and $v_p \in V^2$. We remove nodes that do not have its *yes* counter parts in V^{2A} from V^2. Let O^2 be the set of change core objects in V^2.

Lemma 15. *For each object $o \in O^2 \wedge o \in v_p$ and node $v_q \in V^{2A}$, if $d(o, q) > 2\epsilon$, o do not break the* yes *state of (v_p, v_q).*

Following Lemma 15, we do not remove node v_q from V^{2A} if there exist an object $o \in O^2$ such that $d(o, q) \leq 2\epsilon$ since $st(v_p, v_q)$ may be changed by the deletions. Then, for each edge (v_p, v_q) where $v_p \in V^2$ and $v_q \in V^{2A}$, if $st(v_p, v_q) = yes$, we change $st(v_p, vq) = unknown$, waiting for this edge to be re-updated.

Step D10: Update cluster structures. For each object $o \in O \setminus B$, if o is a *pcore* or *ucore*, we set the *yes* connections for edges $(V_o[i], V_o[i-1])$ where V_o is the set of nodes containing o and $1 \leq i \leq |V_o|$. After that, we re-update the labels of nodes following the connected components of *yes* edges as in Step 3 of IncAnyDBC. This step helps to reduce the possible split causing by the delegation of *yes* edges in Step D8 and D9.

Step D11: Detect possible splits. Given two arbitrary nodes v_p and v_q that belong to the same cluster c. If $label(v_p) \neq label(v_q)$ after Step D10, c is affected by the deletions (indicated by the changes of *yes* edges) and need to be re-checked if it really splits. Let C^A be the set of affected clusters (including all nodes in V^3, which are assigned the same special cluster labels initially).

Lemma 16. *Any cluster $c \notin C^A$ will not be affected by the deletions.*

Proof. (Sketch) Steps D2, D8, and D9 guarantee that all possible broken *yes* edges are changed to *unknown*, waiting for the cluster updates. Thus, c will not be changed. ☐

For each cluster $c \in C^A$, we need to re-cluster it to check if c is really be spitted. To do so, all we need is to change all *no* edges in c back into *unknown* states and rerun the clustering algorithm to look for clusters again. Let V^A be the set of nodes inside C^A. For each node $v_p \in V^A$ and $v_q \in adj(v_p)$, if v_p and v_q currently belong to a splited cluster, we put them into the set of nodes V^E to be examined later. Moreover, if $st(v_p, v_q) = no$, we change it to *unknown* as discussed before.

Step D12: Choose objects to be examined. Let O^E be the set of objects inside $v_p \in V^E$ (exclude node centers, *pborder* and *pnoise* ones).

Proposition 2. *We only need to examine objects in O^E to fully update clusters after the deletions.*

Proposition 2 is straightforwardly drawn from Lemma 16 and Steps D8 to D11 of IncAnyDBC. All edges that are not affected by deleted objects are excluded in steps D8 and D9. Steps D10 and D11 guarantee that if a cluster may be broken, it will be re-examined by placing all of its nodes into the examined node set V^E. Following Lemma 2 and Lemma 6, we need to examine all objects in O^E to clarify these edges as *yes* or *no* ones.

Following Proposition 2, for each object $o \in O^E$, if $st(o) = pcore$ and $ptou(o) = 1$, we change $st(o)$ to *ucore*, indicating that a neighborhood query may need to be repeated on o to build clusters.

Step D13: Update clusters. The cluster update process in the deletion case is also build upon the Step 3 to 6 of IncAnyDBC in Section 3.1, but is limited on the set of objects O^E only like the Step I11 of IncAnyDBC in the insertion case.

Correctness. At the end, IncAnyDBC produces the same clustering results as DBSCAN after the deletions.

Lemma 17. *IncAnyDBC produces identical results to those of DBSCAN after the deletions.*

Proof. (Sketch) Steps D1 to D3 ensure that a core object will be covered in a node after the deletions. Steps D8 and D9 ensure that all possible affected *yes* edges are reversed back to *unknown* states to be re-checked. Step D11 detects any possible broken cluster. All changes can be captured by examining O^E in Step D13 following Proposition 2. Thus, if two core objects $a \in v_p$ and $b \in v_q$ are density-connected in DBSCAN, they will be placed into the same connected component in IncAnyDBC and vice versa. Consequently, IncAnyDBC produces identical results as DBSCAN after the deletions, except for the shared border objects. ☐

Complexity. Steps I0 to I12 take $O(n)$, $O(l\mu)$, $O(vn+v^2)$, $O(n^2)$, $O(v^2)$, $O(mn)$, $O(nv)$, $O(n)$, $O(v^2+vm+mv^2)$, $O(v^2+nv)$, $O(nv)$, $O(v^2)$, $O(v^2)$, and $O(vn)$. Step D13 has the similar time complexity as in Steps 3 to 5 of IncAnyDBC. Thus, the overall time complexity is $O(mn^2)$ like IncDBSCAN. IncAnyDBC requires $O(vn+v^2+l\mu+nm)$ space complexity.

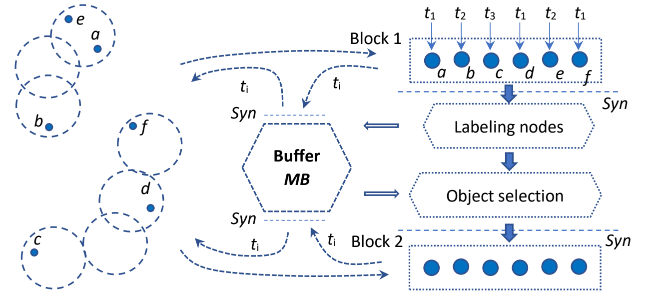

Figure 7: The parallel processing model of IncAnyDBC on multicore CPUs

3.4 Parallel processing

As discussed before, the general idea is processing queries in block and using the results for building clusters by changing the connections among object nodes. Figure 7 illustrates the parallel processing model of IncAnyDBC.

At each iteration, a block of unprocessed objects are selected from the database to processing queries using multiple threads, e.g., objects a to f. We propose to execute each

query independently of each other using a single thread, e.g., thread t_1, t_2, and t_3 processes object a, b, and c, respectively. This is more effective than executing each queries in parallel, especially with index structures since not all of them can be executed in parallel efficiently. Since the neighborhood query times may vary, dynamic scheduling would be employed for better balancing the overall workload of threads. However, since the neighborhoods of objects may overlap, we need to wait for all queries to be completed before being able to update the information of objects and connectivity among nodes. Thus, we use a memory buffer (MB) to temporarily store the neighbors of selected objects. And a barrier is placed for synchronizing all threads after query processing. After that, each thread will grab a stored neighborhood from the buffer MB to update the core information and to connect object nodes into clusters until all of them are processed. Since the neighborhood sizes of objects vary significantly, we propose to use dynamic scheduling for balancing threads' workload. IncAnyDBC then synchronizes all threads and do some necessary sequential tasks before starting the object selection process until it reaches to the final stages or it is terminated by users.

Instead of propagating labels from objects to objects like DBSCAN, IncAnyDBC assigns labels for nodes by following connected components of the *yes* connections. Due to the monotonicity of the cluster structures as described in Section 3.2, connected components change incrementally wrt. new *yes* edges. Thus, we use a Disjoint Set (DJS) data structure to efficiently update the components rather than relooking them from scratch. Each object node will be placed into the DJS. The DJS supports two operations: (1) $FindSet(v_p)$ looks for the label of a node v_p and (2) $Union(v_p, v_q)$ merges two nodes v_p and v_q into the same component. The Union operation is not thread-safe. Thus, it is placed in a critical section for synchronization.

For object selections, we use multiple threads to calculate object's scores. Since each node may have different number of adjacent nodes, we use dynamic scheduling to balance the workload. And top score objects are selected sequentially to be processed in the next block.

IncAnyDBC needs to hold a list of nodes V_p for each object p. Building it is expensive and strongly affect the scalability of IncAnyDBC over multiple threads, especially when we have high numbers of nodes. To do so, each node is first assigned to a fixed thread t. Then each thread t will build its own node list V_p^t for each object p independently to each others. Finally, for each object p, we build V_p by merging all node lists of threads in parallel.

4. EXPERIMENTS

Datasets. We perform experiments on various real datasets acquired from different sources including:

- Farm: contains 3,627,086 objects. Each has 5 VZ-features of a satellite farm image in Saudi Arabia[1] [19].
- Household: consists of 2,049,280 objects with 7 dimensions US census data for electricity and mortgage expenses acquired from the UCI archives [18].
- Sdss2Mass: contains 1,258,127 8-dimension objects describing locations and gravities of different galaxies taken from a cosmological database [15].
- GasSensor: records values of 16 different sensors exposed to Ethylene and CO with 4,178,504 objects acquired from the UCI archives [18].
- PAMAP2: describes the physical activities using inertial measurement units acquired from the UCI archives [18] with 974,479 objects in 39 dimensions.
- Precipitation: contains data of mean monthly surface climate such as precipitations and temperatures over global land areas[2] with 566,268 12-d objects.
- OSFP: acquired from the French national registry of sleep apnea[3]. It describes sets of syndromes for 39,252 patients with Obstructive Sleep Apnea (OSA) such as snoring and stroke.

Systems. Experiments are conducted on Linux server with 2.6 GHz CPUs and 128GB RAM using g++ 4.9.2.

Outline. In Section 4.1 and 4.2, we will first demonstrate the performance of IncAnyDBC during the clustering phase using single and multiple threads. Then we study the cluster update phase in Section 4.3 using single thread and Section 4.4 using multiple threads.

4.1 Clustering performance

Unless otherwise stated, we use default parameters $\mu = 50$, $\alpha = 512$, and $\beta = 4096$.

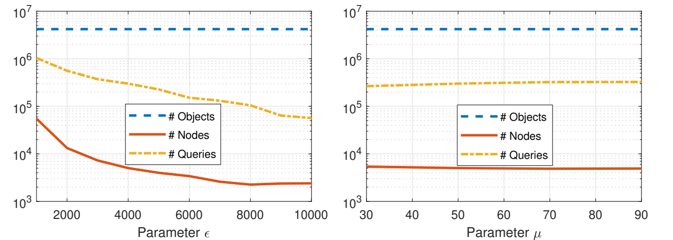

Figure 8: The numbers of object nodes and queries for the dataset GasSensor ($\epsilon = 4000$)

The pruning power of IncAnyDBC. Figure 8 shows the numbers of queries and object nodes of IncAnyDBC for the dataset GasSensor with different parameters μ and ϵ. IncAnyDBC requires much fewer queries to build cluster than DBSCAN (from 4.0 to 74.3 times). Moreover, the number of object nodes is also much smaller than the number of objects (from 76.5 to 1866.1 times). Thus, the query processing time and label propagation time of DBSCAN are significantly reduced (since we only label the nodes). Consequently, IncAnyDBC is much faster than DBSCAN as we shall see in Figure 9.

Performance comparisons. Figure 9 shows the performance of IncAnyDBC compared to DBSCAN [17] and its fastest variants including ρ-DBSCANv2 [21] (a significant improved version of ρ-DBSCAN [19][4]) and an improved version of AnyDBC [36, 37] (where we slightly optimize some steps for saving runtimes) using different parameters ϵ and μ. As suggested from [29, 44], we vary the parameter ϵ from very small to very large to study the performance of these

[1]http://www.satimagingcorp.com/gallery/ikonos/ikonos-tadco-farms-saudiarabia

[2]http://www.cru.uea.ac.uk/

[3]http://www.osfp.fr

[4]Binary file provided by authors (https://sites.google.com/view/approxdbscan)

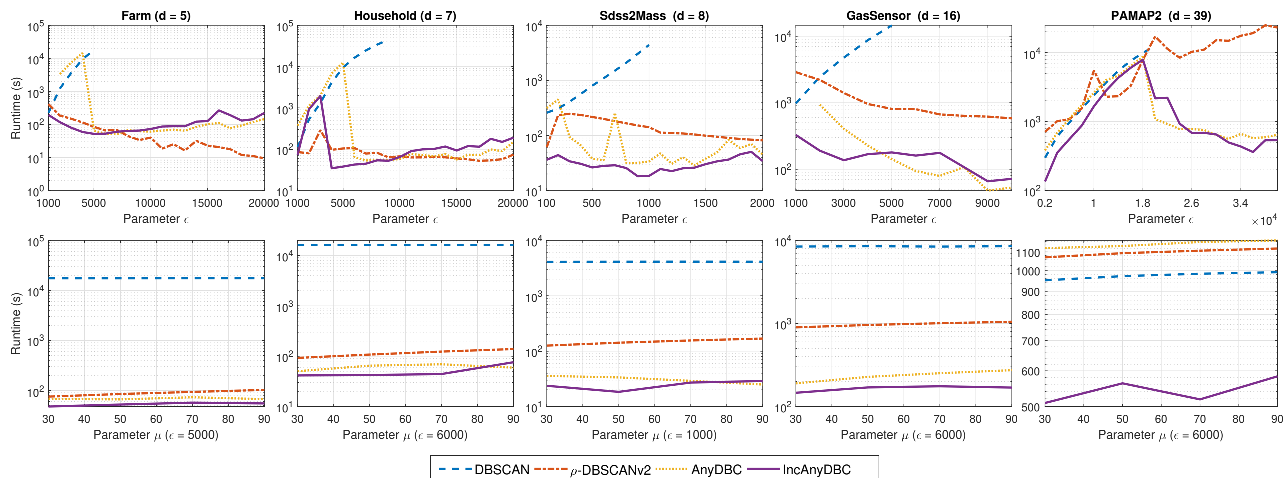

Figure 9: Clustering performance of IncAnyDBC on real datasets (We only run DBSCAN with some parameters ϵ for saving times. AnyDBC ran out of memory when $\epsilon = 1000$ for the datasets Farm and GasSensor)

algorithms. For example, when $\epsilon = 100$, 79% objects are noise for the Sdss2Mass. For the Household data, there is only one cluster containing 99.99% objects when $\epsilon = 20000$. For database indexing, we use *kd*-tree [4][5]

Compared to DBSCAN, IncAnyDBC is much faster in most cases. E.g., the speedup factor ranges from 7.0 to 238.5 times for the Sdss2Mass dataset and from 0.57 to 853.5 times for the Household dataset. This is due to the pruning power of IncAnyDBC as discussed in Figure 8 above. However, when the number of used queries is too large (e.g., when $\epsilon =$ 2000 to 3000 for the Household dataset), IncAnyDBC will run slightly slower than DBSCAN due to its active clustering overheads such as object selections in Step 4. We will discuss more on the performance of IncAnyDBC in the next parts.

When ϵ is large, AnyDBC and IncAnyDBC acquires comparable performance on all the datasets. However, when ϵ is very small, both IncAnyDBC and AnyDBC needs to spend more queries to go to the terminate stage as demonstrated in Figure 8. Thus, the overhead increases. However, since IncAnyDBC does not need to merge clusters and queries like AnyDBC, its overhead is much smaller than that of AnyDBC and thus is much faster. E.g., for the Farm dataset with $\epsilon = 4000$, IncAnyDBC needs only 59.1 seconds (244.2 times faster than AnyDBC with 14,454.5 seconds).

When the dimension d of the data is low (e.g., the Farm and Household datasets), the performances of IncAnyDBC and ρ-DBSCANv2 are comparable. However, when d is larger, IncAnyDBC runs much faster since it does not rely on the grid structure like ρ-DBSCANv2, where the number of cells increases exponentially wrt. to the dimension d and causes significant overheads. E.g., IncAnyDBC is from 1.6 to 8.3 times, from 3.7 to 11.6 times, and from 0.53 to 52.8 times faster than ρ-DBSCANv2 on the datasets Sdss2Mass, GasSensor, and PAMAP2, respectively.

Roles of indexing techniques. Since IncAnyDBC relies on neighborhood queries, using different indexing methods will affect the performance difference between it and ρ-DBSCAN as demonstrated in Figure 10 using kd-tree[6] [4]

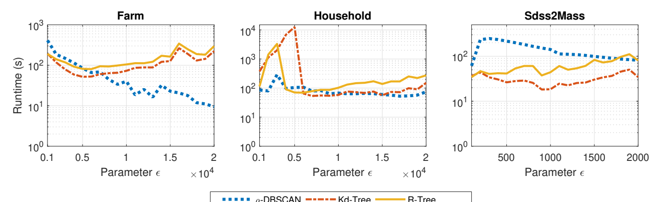

Figure 10: The effects of indexing techniques on IncAnyDBC

and $r-tree$ [7] [24]. Depending on ϵ, the overall runtimes of IncAnyDBC vary several times while rho-DBSCAN is not affected since it does not need to index data.

Effects of parameters μ and ϵ. As shown in Figures 8 and 9, ϵ has a very strong effect on performances of all methods. However, when ϵ increases, the runtime of IncAnyDBC fluctuates instead of increasing like DBSCAN. The main reason is its query pruning scheme. The performance of IncAnyDBC depends on the number of queries its use, which is affected by the final cluster structure. If clusters are well separated, i.e., less *cross-edges* among them, fewer queries need to be used to break these edges following Lemma 6. Thus, the algorithm runs faster and vice versa. In our real datasets, when ϵ is too small, we have many small clusters that stay close to each others and thus are harder to clarify. Thus, IncAnyDBC uses more queries than larger values of ϵ. Moreover, the number of object nodes decreases since more objects will be covered inside a node if ϵ increases as shown in Figure 8 (left). Thus, the performance gap between IncAnyDBC and DBSCAN typically becomes larger when ϵ increases as we can see from Figure 9 (top).

Larger values of μ means it is harder to detect the core properties of objects without doing neighborhood queries on them. Thus, the number of used queries increase slightly as show in Figure 8 (right). This makes the runtime of IncAnyDBC increases in most cases as seen in Figure 9 (bottom) (if the cluster structure does not change much).

Effects of parameter α and β. Figure 11 shows the robustness of IncAnyDBC to the two parameters α and β.

[5]Source from PDSDBSCAN [40]: http://cucis.ece.northwestern.edu/projects/Clustering/.

[6]Source: https://github.com/jmhodges/kdtree2/

[7]Source: https://www.boost.org/

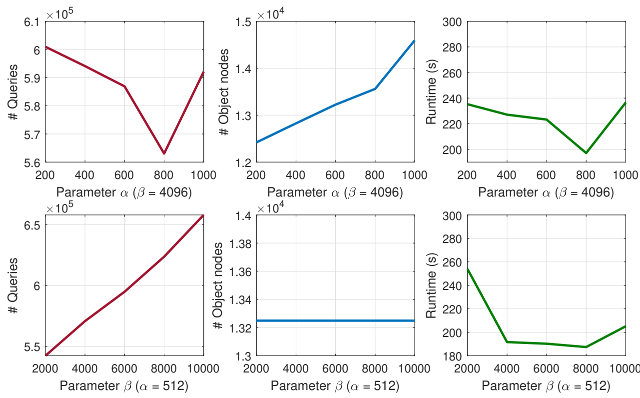

Figure 11: The effects of parameters α and β on the dataset GasSensor ($\epsilon = 2000$)

Generally, when α increases, there will be more nodes due to the block processing scheme in Step 1. Thus, more core objects will be revealed at an early stage, making the algorithm to finish earlier. Hence, the number of used queries goes down. However, when the number of nodes increases, there are chances that two nodes from different clusters are placed close enough to each other, thus creating a *cross-edge* between them following Lemma 3. Following Lemma 6, clarifying these *cross-edges* requires more queries to be performed. This lead to the increase of the number of used queries. As a result, the runtime of IncAnyDBC slightly decreases and increases again as seen in Figure 12 (top). However, when α changes from 200 to 1000, the runtime only changes between 197.0 to 236.5 seconds.

The parameter β is used for balancing the overheads of IncAnyDBC and its pruning power. Smaller β means that objects are frequently evaluated and selected to build clusters (Steps 3 to 5). Thus, better objects are selected and fewer queries are required compared to bigger values of β. On the other hand, the overheads of the active clustering scheme are bigger due to more iterations. These facts affect the runtime of IncAnyDBC as shown in Figure 12 (bottom). When $\beta = 2000$, it takes IncAnyDBC 253.9 seconds. However, when $\beta = 4000$, the runtime reduces to 191.6 seconds (though the number of used queries increases from 542,299 to 570,681) since the overheads are reduced. However, when $\beta = 10000$, the runtime goes up again to 205.0 seconds since the number of increased queries overwhelms the overhead reduction. The performance changes, however, are small, ranging from 187.3 to 253.9 seconds.

In our datasets, we suggest to set α from 400 to 800 and β from 4000 to 8000 to acquire good performance overall.

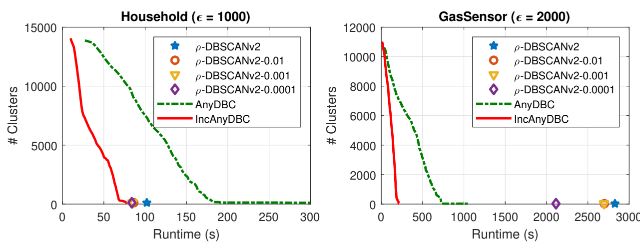

Figure 12: Anytime properties of IncAnyDBC on the Household and GasSensor datasets

Anytime properties. One major advantage of IncAnyDBC is that it can be interrupted at any time to provide approximate results, while other methods like [17, 19, 21] can only provide either an exact result or an approximate result.

Figure 12 shows the anytime property of IncAnyDBC. Due to the monotonicity property (c.f. Section 3.2), the number of clusters reduce very quickly at each iteration to the final number of clusters of DBSCAN at the end. AnyDBC is the only existing algorithm that has the same property. However, it usually has larger initial overheads due to its cluster intersection and merge strategies. ρ-DBSCANv2 can produce exact or approximate results of DBSCAN. However, it is a batch algorithm. We need to set the approximate value ρ and wait for the algorithm to finish to have the result. It cannot work under arbitrary time constraints like IncAnyDBC and AnyDBC.

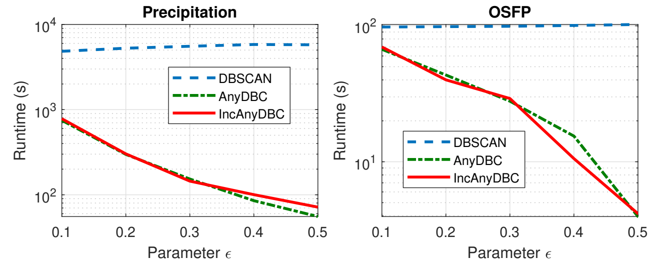

Figure 13: Performance of IncAnyDBC on the dataset Precipitation (Manhattan distance) and OSFP (Jaccard distance)

Other distance functions. While other grid-based methods like [19, 21, 23] can only work under Euclidean distance, IncAnyDBC can work under arbitrary distance metrics.

Figure 13 shows the performance of IncAnyDBC and AnyDBC on the datasets Precipitation and OSFP using Manhattan and Jaccard distance metrics [47] ($\alpha = 128$ and $\beta = 1024$). IncAnyDBC is comparable to AnyDBC on both datasets and is from 1.4 to 135.5 times faster than DBSCAN on both datasets.

4.2 Parallel clustering

Scalability over multiple threads. Figure 14 illustrates the performance of IncAnyDBC, AnyDBC [36], HPDBSCAN [22], and PDSDBSCAN [40] on different datasets using 16 threads. Due to its grid-based scheme, HPDBSCAN can only work on low dimensional datasets Farm and Household with large values of ϵ. PDSDBSCAN, on the other hand, consumes too much memory due to its object storing scheme when the neighborhoods of objects overlap. Thus, when ϵ is large enough, it runs out of memory. Since both HPDBSCAN and PDSDBSCAN do not focus on workload reduction like IncAnyDBC, their performance is significantly overwhelmed by IncAnyDBC. E.g., HPDBSCAN is from 90.9 to 679.6 times slower than IncAnyDBC on the Household dataset and PDSDBSCAN is from 3.4 to 49.8 times slower than on the GasSensor dataset. The bigger the value of ϵ, the larger the performance gap. Compared to AnyDBC, IncAnyDBC is faster in most cases[8], especially when ϵ is small, e.g., 55.4 times when $\epsilon = 2000$ (GasSensor) and 212.4 times when $\epsilon = 5000$ (Household). In terms of scalability, IncAnyDBC also performs better than AnyDBC in most cases

[8]We slightly modified AnyDBC to make it faster. However, its scalability becomes worse than the original version [36].

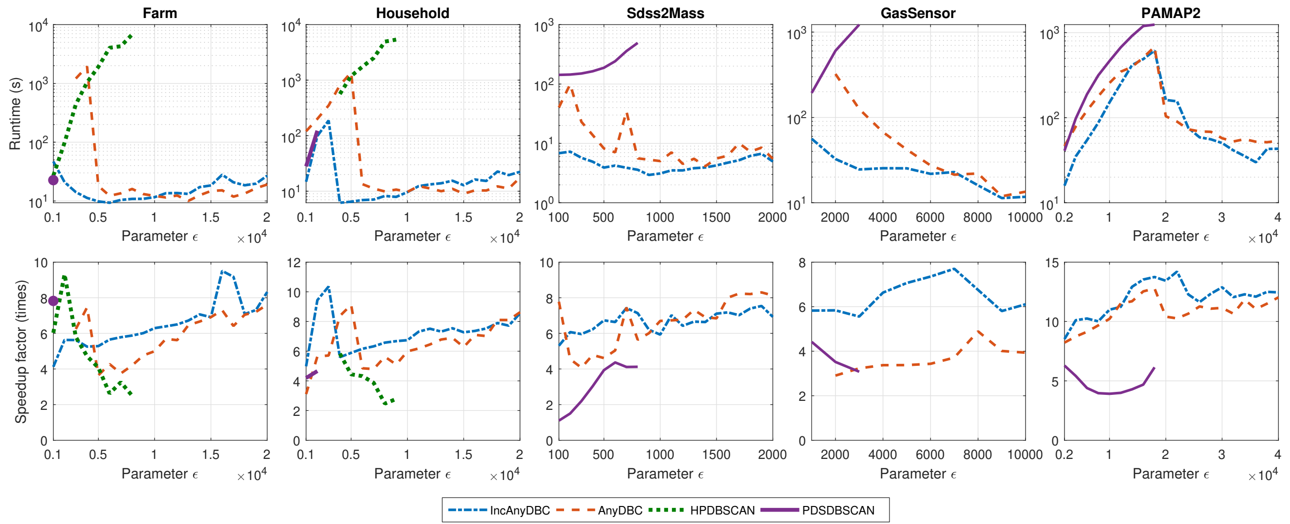

Figure 14: Runtimes (top) and speedup factors (bottom) of different algorithms using 16 threads. HPDBSCAN only works for the dataset Farm and Household when ϵ is large. PDSDBSCAN runs out of memory when ϵ is large enough. AnyDBC runs out of memory when $\epsilon = 1000$ and 2000 (Farm) and $\epsilon = 1000$ (GasSensor)

and much better than HPDBSCAN and PDSDBSCAN. It reaches speedup factors of 9.5, 10.3, 7.5, 7.7, and 14.2 over 16 threads on the datasets Farm, Household, Sdss2Mass, GasSensor, and PAMAP2, respectively. Note that, experiments are ran on two 8-core CPUs, and thus they suffers from the NUMA effects. Without it, the speedup factors of IncAnyDBC would be even better. Currently, IncAnyDBC is not designed as a NUMA aware method.

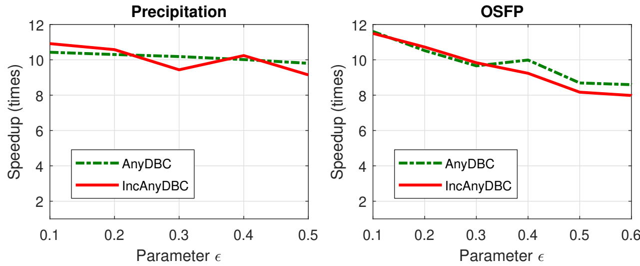

Figure 15: Scalability of IncAnyDBC on the dataset Precipitation (Manhattan distance) and OSFP (Jaccard distance) using 16 threads

Other distance functions. Figure 15 shows the scalability of IncAnyDBC using 16 threads on two datasets Precipitation (L_1) and OSFP (Jaccard distance). It has very good performance on both datasets. The speedup factors are from 9.1 to 10.9 (Precipitation) and from 8.1 to 11.5 (OSFP) over 16 threads. HPDBSCAN [22] and PDSDBSCAN [40] can only work on Euclidean distance and thus are excluded, while AnyDBC acquires a comparable performance to IncAnyDBC.

Which steps of IncAnyDBC scale worse? Figure 16 shows the scalability of IncAnyDBC using 16 threads on three bad cases of PAMAP2, Household, and GasSensor (with lowest scalabilities). Steps 1, 2, 4, and 5 typically takes up most runtimes of IncAnyDBC. However, the scalability of IncAnyDBC typically is very bad on Step 2. There is not a surprise since it has to perform many sequential tasks. When

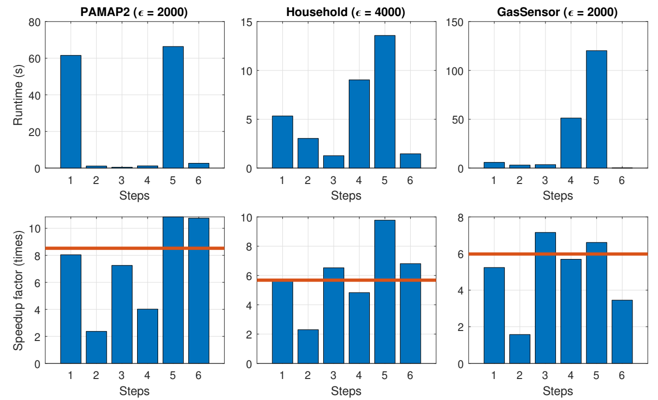

Figure 16: Scalability of each step of IncAnyDBC using 16 threads (for three bad cases). The red horizontal lines show the overall speedup factors

ϵ decreases, the number of nodes increases. This causes more works on Step 2 and thus the scalability of IncAnyDBC is affected more. For GasSensor, since the neigborhood sizes of objects vary significantly, they strongly affect the workload balance for threads, thus the overall speedup factors for Step 1 and 5 (two most expensive steps) decreases, causing the performance degeneration.

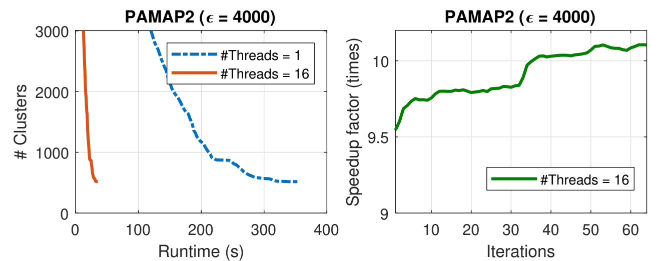

Figure 17: Performance at each step of IncAnyDBC for the PAMAP2 ($\epsilon = 4000$) using 16 threads

Parallel anytime properties. One interesting property of IncAnyDBC is that each step can be parallelized, making it a unique anytime parallel algorithm. As shown in Figure 17 (left), using 16 threads, we can acquire the same clustering result at each iteration of IncAnyDBC much faster (from 9.5 to 10.1 times). Figure 17 (right) shows the speedup factors at each iteration. The more iterations, the better the speedup factors. It is due to the fact that Step 5 usually scales better than other steps as illustrated in Figure 16.

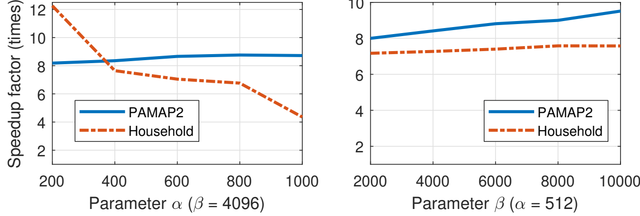

Figure 18: Effects of parameters α and β on the scalability of IncAnyDBC using 16 threads for PAMAP2 ($\epsilon = 2000$) and Household ($\epsilon = 12000$)

Effects of parameters α and β. Figure 18 shows the effects of parameters α and β on the performance of IncAnyDBC. Increasing β makes the overall workload at each iteration larger. This helps to balance threads better. And thus, the scalability of IncAnyDBC typically increases as we can see from Figure 11 (right). The role of α, however, is unclear. On different datasets, it shows different behaviours. E.g., when ϵ increases from 200 to 1000, the speedup factor increases from 8.1 to 8.7 times on the PAMAP2 dataset but decreases significantly from 12.2 to 4.3 times on the Household dataset. Unfortunately, there is no way to predict such behaviors. Thus, in our experiments, we choose α around 500 to bring up average performances for all our datasets.

4.3 Dynamic clustering

We study the performance of IncAnyDBC for both insertion and deletion cases. For each dataset D, we randomly remove a set D' of 100,000 objects from it and use the remainder n objects for clustering. Then we randomly delete m objects from D and randomly insert m objects from D' into D. This process mimics the real behaviors of dynamic data. Unless otherwise stated, we use default parameters $\mu = 20$, $\alpha = 512$, $\beta = 4096$, and $m = 2000$.

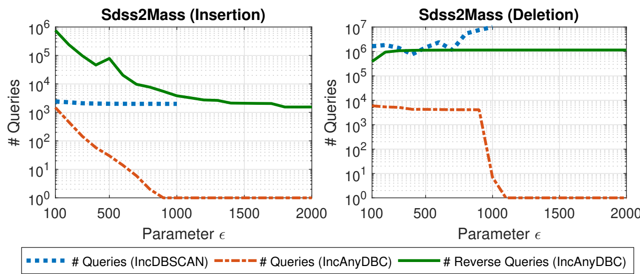

Figure 19: Numbers of queries and reverse queries of IncAnyDBC and IncDBSCAN for Sdss2Mass

The query pruning scheme of IncAnyDBC. Figure 19 (left) shows the number of queries and reverse queries of IncAnyDBC and IncDBSCAN over 2000 insertions. Since IncDBSCAN needs to determine all change core objects before further processing, it requires at least 2,000 queries regardless the parameter ϵ. However, the total number of used queries does not vary much (from 2,003 to 2,428 over 1,158,127 points), meaning that IncDBSCAN works quite stable and very efficient compared to the re-clustering choice. By using reverse queries to detect potential changes and updating clusters under the active scheme, IncAnyDBC uses much less queries than IncDBSCAN (from 0 to 1,485). Since reverse queries are significantly faster than full queries, IncAnyDBC is faster than IncDBSCAN as show in Figure 20. Moreover, when ϵ grows bigger, the cluster structure tends to be more stable and there are fewers border objects (that may change to cores). Thus, the number of queries is typically reduced. Note that, since the query processing time typically increases with ϵ, it does not mean that the runtime of IncAnyDBC will reduces.

The deletion case is much expensive than the insertion case shown in Figure 19 since clusters may be broken and need to be re-clustered. Thus, the total number of queries IncDBSCAN used is much higher, ranging from 1.6 to 10.1 millions (from 676.8 to 5,064.6 times higher than the insertion case). Compared to the data size, it is better to recluster from scratch rather than updating results in this *batch* mode. Since IncAnyDBC processes deleted objects in a *bulk*, it does not need to repeatedly re-verify a cluster many times. Together with its active clustering scheme, it needs only from 0 to 5,965 full queries to update clusters. This dramatically improves the performance compared to IncDBSCAN as we will see in Figure 20 below. Similar to the insertion case, the bigger the parameter ϵ, the fewer queries it uses typically.

Performance comparisons. Figure 20 (top) shows the runtimes of IncAnyDBC and cumulative runtimes IncDBSCAN [16] over 2,000 insertions. When ϵ become biggers, the runtimes of IncAnyDBC fluctuates rather than increasing like IncDBSCAN due to its query pruning scheme as discussed above. Due to its *active bulk processing* scheme, IncAnyDBC significantly outperforms IncDBSCAN in most case, e.g., from 2.3 to 41.0 times faster on Sdss2Mass. The bigger ϵ, the larger the performance gaps. However, in some cases, e.g., Household ($\epsilon = 1,000$) or PAMAP2 ($\epsilon = 10,000$), IncAnyDBC runs slower than IncDBSCAN. The reason is that we must update some *no* edges to *unknown* states in Step I8 and I9 to let the algorithm re-update clusters for capturing all possible cluster merges. In the worst cases, if the changed edges are *cross-edges*, it will be hard to break them as discussed in Section 4.1. Thus, IncAnyDBC consumes more queries than IncDBSCAN and is slower.

The major difference between IncAnyDBC and IncDBSCAN is on the deletion case (c.f. Figure 20 (bottom)), where IncAnyDBC completely outperforms IncDBSCAN in all cases, e.g., from 35.8 to 10,755.9 times on Sdss2Mass. Since the deletion case is much expensive than the insertion case, the overall performance of IncAnyDBC fully dominates IncDBSCAN. With $m = 2,000$ changes, updating clusters using IncAnyDBC is also more efficient than re-clustering the whole database using IncAnyDBC or DBSCAN.

Other distance metrics. Figure 21 shows the performance of IncAnyDBC for OSFP and Precipitation using Jaccard and Manhattan distances ($\alpha = 128$, $\beta = 1,024$, and $m = 1,000$). The same results are observed.

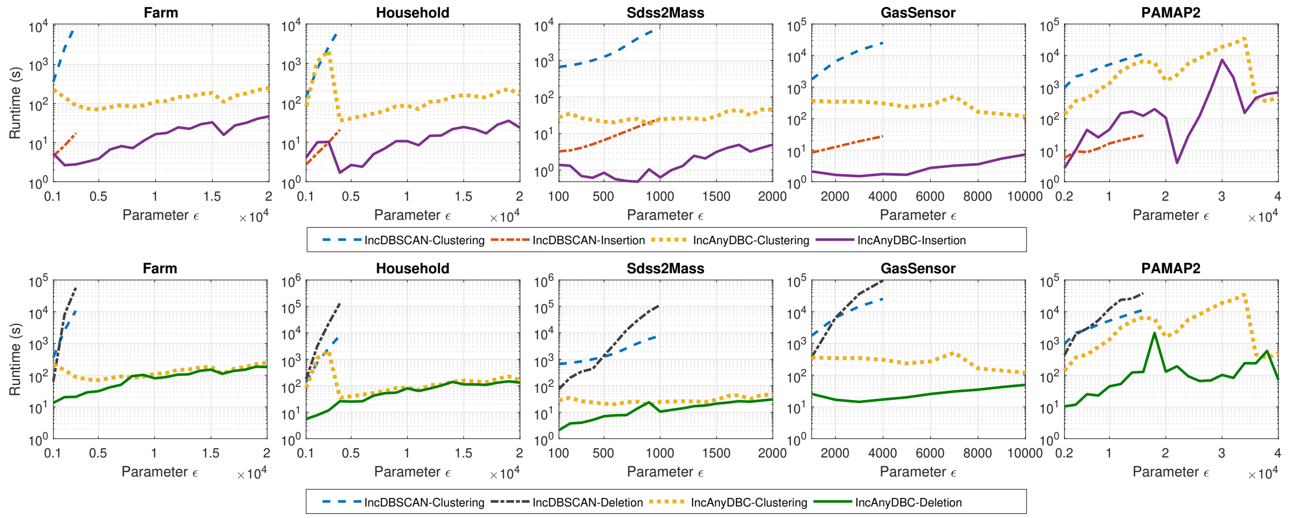

Figure 20: Performance of IncAnyDBC and IncDBSCAN for various real datasets

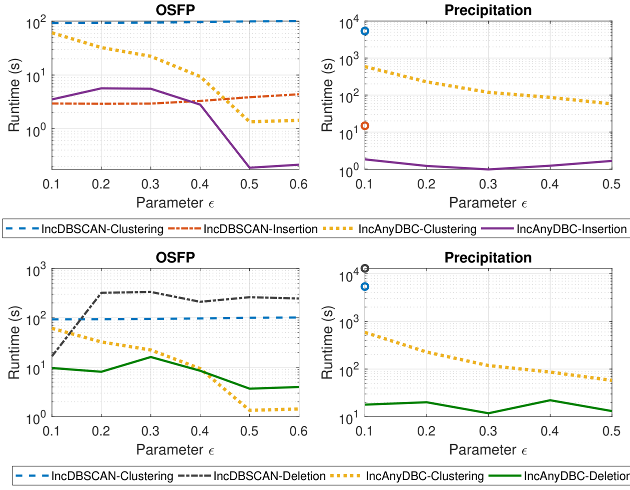

Figure 21: Performance of IncAnyDBC and IncDBSCAN for datasets Precipitation and OSFP

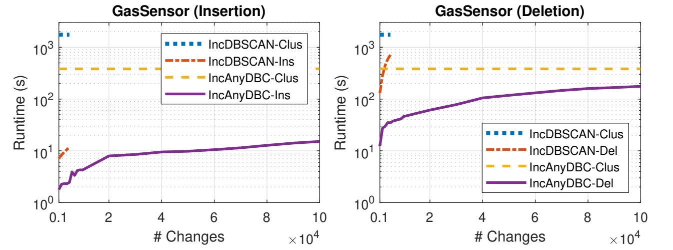

Figure 22: Performance of IncAnyDBC wrt. the numbers of data changes for GasSensor ($\epsilon = 1,000$)

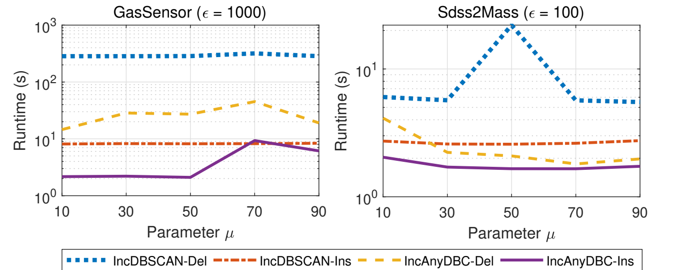

Figure 23: Effects of the parameter μ

Scalability wrt. the numbers of object changes? Figure 22 shows how IncAnyDBC and IncDBSCAN scale when the numbers of inserted and deleted objects vary from 1,000 to 100,000 for GasSensor ($\epsilon = 1,000$). The performance gap between IncAnyDBC and IncDBSCAN increases with ϵ, especially for the deletion case. E.g., the speedup factors of IncAnyDBC over IncDBSCAN changes from 10.4 times to 20.5 times when m increases from 1,000 to 5,000. With $5,000$ changes, updating clusters using IncAnyDBC is 10.3 times and 47.1 times faster than fully reclustering using IncAnyDBC and DBSCAN, respectively. With $100,000$ changes, updating clusters still 2.0 and 9.1 times faster than re-doing whole results using IncAnyDBC and DBSCAN.

Effect of parameters μ and ϵ. Figure 20 and Figure 23 show the effects of parameters μ and ϵ on the performance of IncAnyDBC. As discussed in Section 4.1, the performance of IncAnyDBC depends strongly on the cluster structure of the data that changes wrt. different input parameters. Thus, the actual runtimes of IncAnyDBC fluctuates considerably. However, when the cluster structure remains stable, we can theoretically expect the number of queries decreases with ϵ and increases with μ as seen in Figures 8 and 19.

Effect of parameters α and β. Figure 24 shows the robustness of IncAnyDBC over two parameters β and especially α. When α varies from 100 to 900 and β varies from 1,000 to 9,000, the runtimes of IncAnyDBC changes negligible. Compared to the clustering phases in Section 4.1, the changes are harder to see since IncAnyDBC updates clusters very efficiently. But we can see the runtime increases with β due to redundant queries during the re-clustering phases I11 and D13 of IncAnyDBC. Changes caused by α could not be clearly observed since the number of newly created nodes in Steps I3 and D3 are typically too small to have any visible effects on the overall performance of IncAnyDBC.

Anytime cluster update. Similar to the clustering phase, IncAnyDBC can update clusters in an *anytime* fashion as

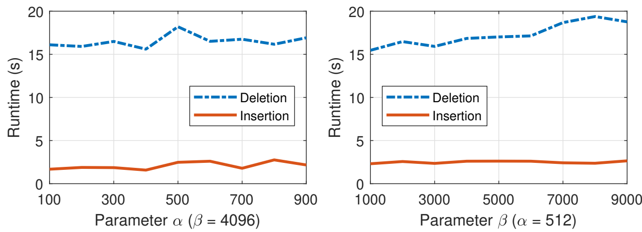

Figure 24: Effects of the parameter α and β for Sdss2Mass ($\epsilon = 500$ and $m = 10,000$)

seen in Figure 29. The number of clusters gradually reduces during its execution until it comes to the final result of DBSCAN following the monotonicity property presented in Section 3.2. The deletion case typically starts with high numbers of clusters (due to the removal of *yes* edges in Step D8 and D9). But the numbers of clusters reduce very quickly at each iteration. On the other hand, the insertion case usually starts with closer numbers of clusters to the final ones since the number of newly created objects in Step I3 is usually small. These mean we can stop the algorithm at early iterations while still having a very close result to the final one of DBSCAN. None of existing methods for dynamic clustering has this *anytime* property.

4.4 Parallel dynamic clustering

Scalability wrt. the number of threads. Figure 25 shows the scalability of clustering, deletion, and insertion phases of IncAnyDBC over different datasets using 16 threads. Though the results fluctuate with different values of ϵ, large values of ϵ typically bring up higher speedup factors due to the increasing of the parallel-able workload compared to the non-parallel one (Amdahl's law). Overall, IncAnyDBC scales very well on 16 threads. The speedup factors are up to 11.3 (9.1), 9.7 (9.2), 8.4 (9.2), 9.1 (7.4), and 15.2 (15.3) times for the insertion (deletion) case on Farm, Household, Sdss2Mass, GasSensor, and PAMAP2, respectively. Without the NUMA effect due to the use of 2 CPUs, the speedup factors would be even better.

Scalability for other data. On OSFP and Precipitation, IncAnyDBC still has very good scalability as seen in Figure 21. However, the speedup factor decreases with ϵ. The reason is that we do not use indexing technique. Thus, the query processing time is well-balanced for threads regardless of ϵ. In this case, bigger ϵ means bigger object nodes and overheads. This drags the overall scalability goes down in many cases, especially when the dataset is small like OSFP.

Scalability wrt. the numbers of object changes? When the number of changes increases from 1,000 to 100,000 as illustrated in Figure 22, the overall workload of IncAnyDBC increases. Thus, it leads to the improvement of the scalability of IncAnyDBC. E.g., for Household, the speedup factor is 6.6, 10.4, and 12.6 times over 16 threads when m changes from 1,000 to 10,000 and 100,000, respectively.

Effect of parameters α and β. When α and β increases as demonstrated in Figure 28, the scalability of IncAnyDBC over 16 threads remains quite stable, especially for the deletion case. Theoretically, increasing β will balance workload of threads better, thus leading to better speedup factor as we have seen in Figure 18. However, since the overall update times are too small, the effect is thus not visible clearly.

Anytime cluster update. Since IncAnyDBC is an *parallel anytime* method, multiple threads can be used to have intermediate results faster as shown in Figure 29. During the execution time, the intermediate speedup factor changes slightly at each iteration like the clustering phase shown in Figure 17. However, it does not show clear increasing or decreasing trend due to small update times.

5. RELATED WORKS AND DISCUSSION

Incremental density-based clustering. Finding clusters in dynamic databases is an important research focus for many years [1,5,12,16,20,33,35]. Most of them focus on incrementally updating existing clusters when changes occur in the databases instead of reclustering from scratch. As a fundamental clustering algorithm with many real-life applications [38], DBSCAN [17] is an attractive target for this incremental clustering approach, e.g., [16,20].

In [16], the locality natur of DBSCAN is exploited to limit the update areas, thus saving computation costs. An inserted object may merge existing clusters. And a deleted object may break a cluster into smaller parts and needs to be rebuilt, which is very expensive. Gan and Tao [20] introduces a grid-based approach for updating clusters very efficiently. However, their algorithm can only approximate the result of DBSCAN when the data dimension $d > 2$. Its grid-based scheme also limits it to low-dimensional data under Euclidean distance only, thus reducing its applicability. Both IncDBSCAN [16] and ρ-DBSCAN [20] work in a *batch* scheme. They update clusters with each change. In contrast, IncAnyDBC can update clusters in a *bulk* mode to reduce overheads. Thus, it is much faster than IncDBSCAN. Moreover, IncAnyDBC has other attractive properties. First, its can work under arbitrary time constraints and can provide both exact and approximate results of DBSCAN. To the best of our knowledge, IncAnyDBC is the first *anytime* algorithm for incrementally update DBSCAN's clusters. Second, it is not limited to Euclidean distance like ρ-DBSCAN and can work with arbitrary distance metrics like IncDBSCAN.

Density-based methods for streaming data such as DenStream [11] and [48] has an incremental natur like IncAnyDBC. However, they produce different clustering results to DBSCAN and thus are out of scopes of this work.

Parallel incremental clustering. To the best of our knowledge, IncAnyDBC is the first parallel method for incrementally updating DBSCAN's clusters on multicore CPUs. It not only tries to increase the computation throughput like other traditional parallel algorithms but also tries to reduce the overall workload. Combined with the *anytime* property, IncAnyDBC is a *unique anytime work-efficient* technique for finding clusters in dynamic databases.

Density-based clustering. Reclustering the whole database from scratch is a potential approach when the number of changes are too large as discussed in Section 3.3. To do that, many different methods for enhancing performance of DBSCAN such as [19,21,23,37] can be employed.

Gunawan [23] introduces a grid-based method for clustering 2-d data in $O(nlogn)$ time complexity rather than $O(n^2)$ of DBSCAN. The result is further strengthened by Gan and Tao in [21] using the concepts of Voronoi diagram and Delaunay triangulation. ρ-DBSCAN [19] is another grid-based method built upon the Bichromatic Closest Pair (BCP) problem for connecting grid cells into density-

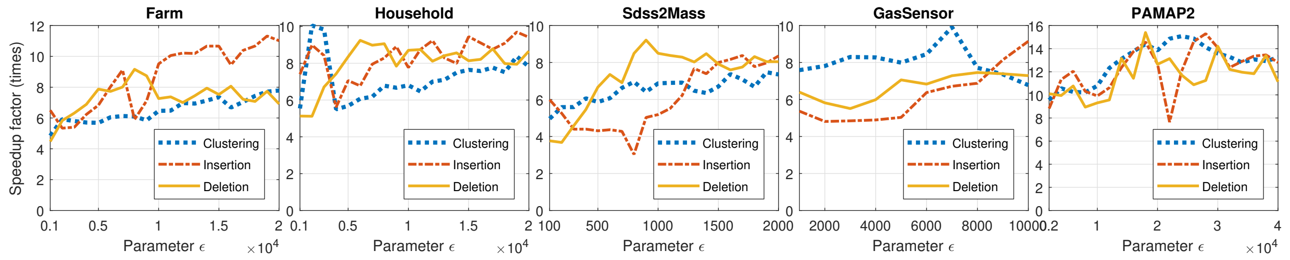

Figure 25: Scalability of IncAnyDBC over 16 threads for various real datasets

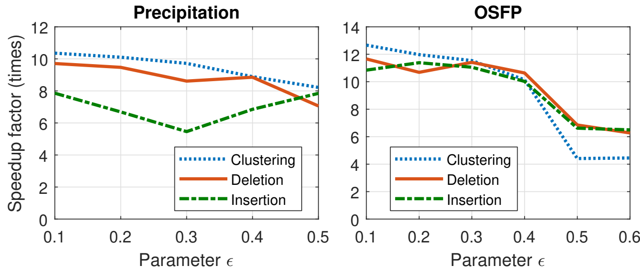

Figure 26: Scalability over 16 threads of IncAnyDBC for datasets Precipitation and OSFP

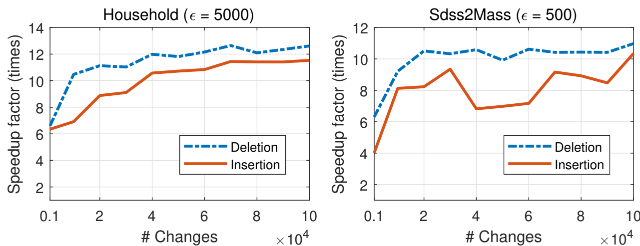

Figure 27: Scalability of IncAnyDBC using 16 threads for Household and Sdss2Mass

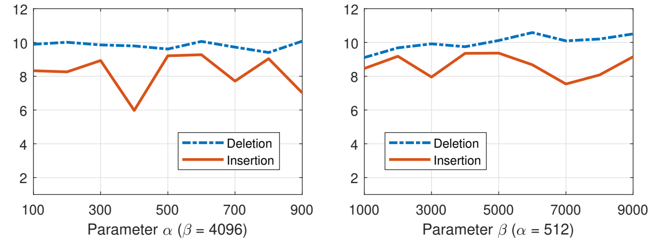

Figure 28: Effects of the parameter α and β on the scalability of IncAnyDBC using 16 threads for Sdss2Mass ($\epsilon = 500$ and $m = 10000$)

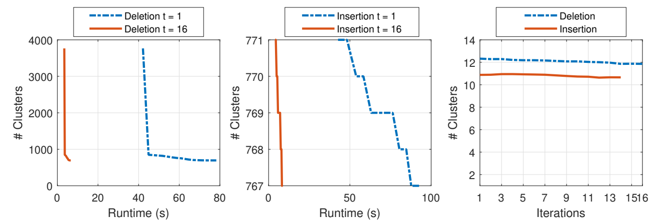

Figure 29: Runtime, number of clusters, and scalability at each iteration of IncAnyDBC for PAMAP2 ($\epsilon = 2000$, $m = 100,000$)

connected clusters. It works very well on low-dimensional data. However, since the number of cells grows exponential with the data dimension, it suffers from performance degradation on high dimensional data. Moreover, it can only work with Euclidean distance. There exists several extensions of ρ-DBSCAN, e.g., [32,43]. In [21], the authors present an optimized version of ρ-DBSCAN (denoted as ρ-DBSCANv2), which is 10-20 times faster than ρ-DBSCAN when dealing with small values of ϵ. Other methods such as [6, 10, 39] exploit lower-bounding distances to reduce the clustering time of DBSCAN under filter-refinement schema. However, they are more suitable for small datasets with very expensive distance functions. AnyDBC [37] is the closest work to IncAnyDBC. Both of them are based on the general idea of summarization objects into sub-groups using a few neighborhood queries and iteratively merge them with additional queries to form clusters. However, IncAnyDBC and AnyDBC follows two completely different algorithmic schema. AnyDBC merges connected sub-groups into a single one at each iteration. Thus, it suffers from higher overheads than IncAnyDBC which only changes the connectivity statuses of subgroups. On the other hand, IncAnyDBC needs a special trick to reduce the required queries to build clusters as presented in Step 5 of IncAnyDBC in Section 3.2. Moreover, AnyDBC looses important information on local sub-group connections that can be exploited to efficiently update clusters as presented in IncAnyDBC. Compared to existing methods described above, both IncAnyDBC and AnyDBC can work on arbitrary distance metrics. They are *anytime* algorithms and can work under arbitrary time constraints to provide exact or approximate results of DBSCAN.

There are many other methods aiming at improving DBSCAN's performance such as BRIDGE [14], DBR [49], SDBSCAN [52], and IDBSCAN [8]. However, they can only produce (coarse) approximate results instead of exact results of DBSCAN like IncAnyDBC.

Parallel density-based clustering. Parallelizing DBSCAN is one of the most active research topics for enhancing DBSCAN with many proposed techniques such as [**?**, 3, 7, 9, 13, 22, 26, 27, 34, 40, 41, 46, 50, 51]. Most of them focus on parallelizing DBSCAN on distributed systems including MPI-based master-slave schema [9, 28, 51], MapReduce [13, 26], Spark [34, 46], and Parameter Server [27]. Patwary et al. [40] point out that these algorithms do not scale well when ported to shared memory systems such as multicore CPUs. Some other algorithms aim at extending DBSCAN on Graphic Processing Units (GPUs) such as [3, 7, 50]. There are some methods that are designed to work with multicore CPUs including [22, 40, 41].

PDSDBSCAN [40] employs a Disjoint Set data structure

to merge two points if they belong to the same cluster in a bottom-up clustering scheme. However, it must perform all queries, thus suffering from very high workload compared to IncAnyDBC. Consequently, it is much slower than IncAnyDBC as studied in Section 4.2. Pardicle [41] is more efficient than PDSDBSCAN but it can only produce approximate results of DBSCAN. HPDBSCAN [22] exploits the data grid structures to build clusters in parallel. However, it suffers from performance degradation on high dimensional data due to the exponential number of cells. None of these methods has an *anytime* property like IncAnyDBC. Moreover, since they do not focus on workload reduction, they are not *work-efficient* and thus run much slower than IncAnyDBC using single or multiple threads. AnyDBC-MC [**?**] is the closest work of IncAnyDBC. However, it differs significantly with IncAnyDBC in its algorithmic operation as discussed above. Moreover, since it uses bit vectors to find cluster intersections and to merge them, it consumes much memory than IncAnyDBC.

6. CONCLUSION

In this paper, we introduce the first and *unique anytime work-efficient parallel* algorithm, called IncAnyDBC, to efficient update density-based clusters for very large complex data on multicore CPUs. For data clustering, IncAnyDBC *actively* chooses a subset of objects to build clusters in an iterative manner. As a result, it consumes only few queries to build the same clustering results as DBSCAN. Thus, it is orders of magnitudes faster than DBSCAN and its variant. IncAnyDBC reserves local cluster structures of data and exploits them to *actively* update clusters when there are changes in the databases such as inserted or deleted objects. Thus, it needs much less queries than the state-of-the-art method IncDBSCAN for updating results. Moreover, changes are update in *bulks* rather than *batches* like existing techniques for reducing overheads. Hence, it is much efficient than IncDBSCAN, especially for the deletion case. IncAnyDBC, due to its *anytime* property, can work under arbitrary time constraints and provides exact or approximate results of DBSCAN on demands. Its *block* processing scheme allows it to be parallelized efficiently on multicore CPUs. Experiments on our systems with 16 CPU cores show that IncAnyDBC scales very well with the number of threads (up to ≈ 15 times over 16 threads).

AcknowledgmentThis work is supported by the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-15-IDEX-02) and a Villum postdoc fellowship.

7. ADDTIONAL MATERIAL

8. REFERENCES

[1] M. Ackerman and S. Dasgupta. Incremental Clustering: The Case for Extra Clusters. In *NIPS*, pages 307–315, 2014.

[2] C. C. Aggarwal and C. K. Reddy, editors. *Data Clustering: Algorithms and Applications*. CRC Press, 2014.

[3] G. Andrade, G. S. Ramos, D. Madeira, R. S. Oliveira, R. Ferreira, and L. C. da Rocha. G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering. In *ICCS*, pages 369–378, 2013.

[4] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] P. Bhattacharjee and A. Awekar. Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for Dynamic Datasets. In *ECIR*, pages 568–574, 2017.

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

[7] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther. Density-based Clustering using Graphics Processors. In *CIKM*, pages 661–670, 2009.

[8] B. Borah and D. K. Bhattacharyya. An Improved Sampling-Based DBSCAN for Large Spatial Databases. In *ICISIP*, pages 92–96, 2004.

[9] S. Brecheisen, H. Kriegel, and M. PfeifIe. Parallel Density-Based Clustering of Complex Objects. In *PAKDD*, pages 179–188, 2006.

[10] S. Brecheisen, H. Kriegel, and M. Pfeifle. Efficient Density-Based Clustering of Complex Objects. In *ICDM*, pages 43–50, 2004.

[11] F. Cao, M. Estert, W. Qian, and A. Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*, pages 328–339. SIAM, 2006.

[12] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental Clustering and Dynamic Information Retrieval. In *STOC*, pages 626–635, 1997.

[13] B.-R. Dai and I.-C. Lin. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. In *CLOUD*, pages 59–66, 2012.

[14] M. Dash, H. Liu, and X. Xu. '1 + 1 > 2': Merging Distance and Density Based Clustering. In *DASFAA*, pages 32–39, 2001.

[15] G. De Lucia and J. Blaizot. The hierarchical formation of the brightest cluster galaxies. *Monthly Notices of the Royal Astronomical Society*, 375(1):2–14, 2007.

[16] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental Clustering for Mining in a Data Warehousing Environment. In *VLDB*, pages 323–333, 1998.

[17] 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.

[18] A. Frank and A. Asuncion. UCI machine learning repository.

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

[20] J. Gan and Y. Tao. Dynamic Density Based Clustering. In *SIGMOD*, pages 1493–1507, 2017.

[21] J. Gan and Y. Tao. On the Hardness and Approximation of Euclidean DBSCAN. *ACM Trans. Database Syst.*, 42(3):14:1–14:45, 2017.

[22] M. Götz, C. Bodenstein, and M. Riedel. HPDBSCAN: highly parallel DBSCAN. In *MLHPC*, pages 2:1–2:10, 2015.

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

[24] A. Guttman. R-Trees: A Dynamic Index Structure for

Spatial Searching. In *International Conference on Management of Data (SIGMOD)*, pages 47–57, 1984.

[25] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques, Third Edition*. Morgan Kaufmann Publishers, 2012.

[26] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan. MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce. In *ICPADS*, pages 473–480, 2011.

[27] X. Hu, J. Huang, and M. Qiu. A Communication Efficient Parallel DBSCAN Algorithm based on Parameter Server. In *CIKM*, pages 2107–2110, 2017.

[28] E. Januzaj, H.-P. Kriegel, and M. Pfeifle. Scalable Density-Based Distributed Clustering. In *PKDD*, pages 231–244, 2004.

[29] H. Kriegel, E. Schubert, and A. Zimek. The (black) art of runtime evaluation: Are we comparing algorithms or implementations? *Knowl. Inf. Syst.*, 52(2):341–378, 2017.

[30] J. Lee, J. Han, and K. Whang. Trajectory clustering: a partition-and-group framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 593–604, 2007.

[31] C. Li, A. Datta, and A. Sun. Mining latent relations in peer-production environments: a case study with wikipedia article similarity and controversy. *Social Netw. Analys. Mining*, 2(3):265–278, 2012.

[32] T. Li, T. Heinis, and W. Luk. Hashing-Based Approximate DBSCAN. In *ADBIS*, pages 31–45, 2016.

[33] J. Lin, M. Vlachos, E. J. Keogh, and D. Gunopulos. Iterative Incremental Clustering of Time Series. In *EDBT*, pages 106–122, 2004.

[34] A. Lulli, M. Dell'Amico, P. Michiardi, and L. Ricci. NG-DBSCAN: Scalable Density-Based Clustering for Arbitrary Data. *PVLDB*, 10(3):157–168, 2016.

[35] S. T. Mai, S. Amer-Yahia, I. Assent, M. S. Birk, M. S. Dieu, J. Jacobsen, and J. Kristensen. Scalable Interactive Dynamic Graph Clustering on Multicore CPUs. *IEEE Trans. Knowl. Data Eng.*, 31(7):1239–1252, 2019.

[36] S. T. Mai, I. Assent, J. Jacobsen, and M. S. Dieu. Anytime parallel density-based clustering. *Data Min. Knowl. Discov.*, 32(4):1121–1176, 2018.

[37] S. T. Mai, I. Assent, and M. Storgaard. AnyDBC: An Efficient Anytime Density-based Clustering Algorithm for Very Large Complex Datasets. In *KDD*, pages 1025–1034, 2016.

[38] 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.

[39] 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.

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

[41] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey. Pardicle: Parallel Approximate Density-Based Clustering. In *SC*, pages 560–571, 2014.

[42] D. Q. Phung, B. Adams, S. Venkatesh, and M. Kumar. Unsupervised context detection using wireless signals. *Pervasive and Mobile Computing*, 5(6):714–733, 2009.

[43] T. Sakai, K. Tamura, and H. Kitakami. Cell-Based DBSCAN Algorithm Using Minimum Bounding Rectangle Criteria. In *DASFAA Workshop*, pages 133–144, 2017.

[44] E. Schubert, J. Sander, M. Ester, H. Kriegel, and X. Xu. DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. *ACM Trans. Database Syst.*, 42(3):19:1–19:21, 2017.

[45] S. Singh and A. Awekar. Incremental shared nearest neighbor density-based clustering. In *CIKM*, pages 1533–1536, 2013.

[46] H. Song and J. Lee. RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning. In *SIGMOD*, pages 1173–1187, 2018.

[47] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[48] L. Wan, W. K. Ng, X. H. Dang, P. S. Yu, and K. Zhang. Density-based clustering of data streams at multiple resolutions. *ACM Transactions on Knowledge discovery from Data (TKDD)*, 3(3):14, 2009.

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

[50] B. Welton, E. Samanas, and B. P. Miller. Mr. scan: extreme scale density-based clustering using a tree-based network of gpgpu nodes. In *SC*, page 84, 2013.

[51] X. Xu, M. Ester, H.-P. Kriegel, and J. Sander. A Distribution-based Clustering Algorithm for Mining in Large Spatial Databases. In *ICDE*, pages 324–331, 1998.

[52] 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.