



**QUEEN'S  
UNIVERSITY  
BELFAST**

## Incremental Density-based Clustering on Multicore Processors

Mai, T. S., Jacobsen, J., Amer-Yahia, S., Spence, I., Assent, I., Tran, N. P., & Nguyen, Q. V. H. (2020). Incremental Density-based Clustering on Multicore Processors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Advance online publication. <https://doi.org/10.1109/TPAMI.2020.3023125>

**Published in:**

IEEE Transactions on Pattern Analysis and Machine Intelligence

**Document Version:**

Peer reviewed version

**Queen's University Belfast - Research Portal:**

[Link to publication record in Queen's University Belfast Research Portal](#)

**Publisher rights**

© 2020 IEEE.

This work is made available online in accordance with the publisher's policies. Please refer to any applicable terms of use of the publisher.

**General rights**

Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.


**Take down policy**

The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact [openaccess@qub.ac.uk](mailto:openaccess@qub.ac.uk).

**Open Access**

This research has been made openly available by Queen's academics and its Open Research team. We would love to hear how access to this research benefits you. – Share your feedback with us: <http://go.qub.ac.uk/oa-feedback>

# Incremental Density-based Clustering on Multicore Processors

Son T. Mai, Jon Jacobsen, Sihem Amer-Yahia, Ivor 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 changes in *bulks* rather than *batches* like state-of-the-art methods for reducing update overheads. Second, it keeps an underlying cluster structure called the object node graph during the clustering process and uses it as a basis for incrementally updating clusters wrt. inserted or deleted objects in the database by propagating changes around affected nodes only. In addition, IncAnyDBC *actively* and *iteratively* examines the graph and chooses only a small set of most meaningful objects to produce exact clustering results of DBSCAN or to approximate results under arbitrary time constraints. This makes it more efficient than other existing methods. Third, by processing objects in *blocks*, IncAnyDBC can be efficiently 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.

**Index Terms**—Density-based clustering, anytime clustering, incremental clustering, active clustering, multicore CPUs



## 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 [1], [2]. 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 scratch is an important research focus [3]–[5]. In many clustering methods, the cluster label of an object is highly dependent on many other ones, making an efficient cluster update process a challenging task [6]. One example is the density-based clustering algorithm DBSCAN [7], one of the most widely used data clustering methods with many real-world applications [8]–[10].

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  $\mu$  neighbors within a specific distance threshold  $\epsilon$ . 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 [3]. For example, an inserted object may merge some existing clusters within its neighborhood. On the other hand, a deleted object may break clusters into smaller pieces. In this case, these clusters need to be re-grouped due to the label dependency of objects. Thus, in

some cases, 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 efforts and thus limits the applicability of the algorithm.

**Contribution.** In this work, we focus on an efficient approach for incrementally updating clusters following the notion of DBSCAN. Our algorithm, called IncAnyDBC, has some unique properties as follows.

First, before updating, existing techniques like IncDBSCAN [3] rely on the original DBSCAN algorithm [7] to group objects and determine their core properties. However, DBSCAN requires all neighborhood queries to be performed, which is expensive. It also does not keep enough information on the cluster structure to 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 serve 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. Users can interact with the algorithm during its execution and terminate it whenever they are satisfied with the current results. This *active clustering* scheme brings up some benefits: (1) IncAnyDBC can produce the same result as DBSCAN with fewer queries, thus enhancing performance; (2) it can be interrupted and resumed at any time to provide good approximate results (or ultimately the exact results of DBSCAN), while most existing techniques can only produce a single approximate or exact result, e.g., [11], [12]. This *anytime* property makes IncAnyDBC useful for applications with limited time constraints or for 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

Corresponding author ()

- Queen's University Belfast, UK.  
Email: {thaison.mai, i.spence, n.tran}@qub.ac.uk
- University of Grenoble Alpes, France.  
Email: sihem.amer-yahia@imag.fr
- Griffith University, Australia.  
Email: quocviethung.nguyen@griffith.edu.au
- Aarhus University, Denmark.  
Email: {ira, jon}@cs.au.dk

Manuscript received xxx; revised xxx.

like state-of-the-art techniques such as [11]–[13]; and (4) the underlying node structure is preserved after the clustering and can be exploited to efficiently update the clusters after insertions or deletions instead of reclustering from scratch.

Second, when there are changes in the data, existing techniques such as [3], [6] update clusters in a *batch* mode (i.e., processing changes one-by-one). This scheme incurs many redundant overheads, especially when the number of changes is large. Our algorithm, in contrast, updates 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 build the final clustering results. Similar to the clustering phase, clusters are rebuilt in an iterative way by letting the algorithm *actively* choose a subset of objects to query at each iteration. Thus, the clusters are finally updated with fewer queries compared to IncDBSCAN [3], thus making it more 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, no existing incremental technique 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 [6].

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 more quickly than state-of-the-art sequential techniques such as DBSCAN [7] and IncDBSCAN [3] using one thread, while scaling very well with the total number of threads. Moreover, the *anytime* property still holds 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 the density-based notions of DBSCAN.

**Summarization.** Our major contributions are as follows:

- We introduce an efficient clustering method for initializing cluster structures before updates. Our algorithm uses much fewer 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, our technique relies on an efficient query pruning scheme and thus it is more efficient than existing techniques like IncDBSCAN. The anytime property is also supported while updating clusters.
- We propose a way of efficiently parallelizing IncAnyDBC on 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 to 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 [7] 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 objects  $p$  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 belongs to only one cluster, while a border object might be shared by multiple clusters.

**Incremental DBSCAN.** When there is a change (insertion or deletion), instead of re-building clusters from scratch, Ester et al. [3] 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.
- **Deletion:** A deleted object may be a core or may change other core objects into a non-core ones. This causes clusters to be split or removed. E.g., the deleted object  $b$  breaks cluster  $C_3$  into two smaller clusters  $C_{31}$  and  $C_{32}$ .

Note that insertion or deletion of an object may merge or split more than two clusters respectively. 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 reclustering from scratch. Each update takes  $O(n^2)$  time complexity.

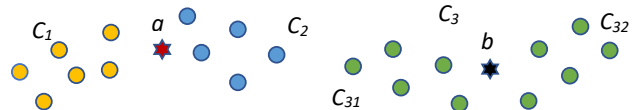


Fig. 1. Incremental clustering: (1) the inserted object  $a$  merges two clusters  $C_1$  and  $C_2$  into a single cluster and (2) the deleted object  $b$  breaks cluster  $C_3$  into two small clusters

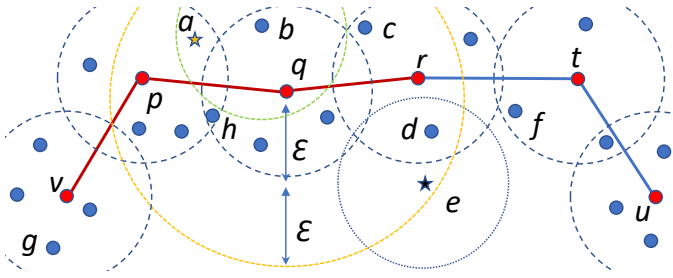


Fig. 2. The general idea of IncAnyDBC

### 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

Figure 2 illustrates general ideas of IncAnyDBC including: summarization, active clustering, block processing, incremental processing in bulks, and parallel processing. Table 3.1 shows some common abbreviations used in the paper.

**Summarization.** IncAnyDBC first summarizes all objects into object nodes using neighborhood queries. Each contains a neighborhood of an object (c.f. Definition 6), e.g., nodes  $v_p$ . 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 serve 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, 2 and 4). This scheme brings up two different benefits: (1) it allows the algorithm to build clusters without having to perform all neighborhood queries, thus significantly improving performance; and (2) it keeps track off necessary cluster structure to efficiently rebuild clusters after data changes (insertions or deletions).

**Active clustering.** Instead of performing all queries like DBSCAN, IncAnyDBC iteratively examines the current node structure and chooses the 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 of 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.

**Block processing.** At each iteration, IncAnyDBC chooses a small set of objects to perform queries instead of a single object. This scheme trades off between the cost of repeatedly examining and choosing 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 inserted object  $a$  (Figure 2) will be absorbed into the

$N_p$	The $\mathcal{E}$ -neighborhood of the object $p$
$v_p$	The object node contains the object $p$ and its neighbors $N_p$
$V_p$	The list of object nodes that contains the object $p$
$G = (V, E)$	The graph $G$ where $V$ is the set of object nodes and $E$ is the set of edges that connect two object nodes $(v_p, v_q)$
$L$	The non-core list contains all processed non-core object $p$
$st(p)$	The state of the object $p$ (c.f., Figure 4)
$st(v_p, v_q)$	The state of an edge $(v_p, v_q)$ (c.f., Figure 5)
$nei(p)$	The current number of neighbors of the object $p$
$usize(v_p)$	The number of unprocessed objects inside the object node $v_p$
$level(p)$	The number of objects when we perform the neighborhood query on the object $p$
$S$	The block of objects that we process at each iteration
$B$	The number of inserted or deleted-objects in bulk
$M_p$	The reverse neighborhood queries of the object $p$ on $B$

TABLE 1  
LIST OF ABBREVIATIONS

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 (e.g.,  $(v_t, v_u)$ ) will not be affected and can be ignored (Lemma 12 and 10). 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.** We propose to process all changes in a bulk scheme instead of a batch 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.

**Parallel processing.** The underlying node structure and block processing scheme allow us to design an efficient parallel technique. The general idea is to process all the queries in a block independently of other blocks. The results are stored in a buffer and then are used for constructing clusters. This scheme reduces the synchronization costs for neighborhood queries and cluster building, thus enhancing the performance (c.f. Section 3.4). Moreover, it reduces the sequential costs while propagating cluster labels among nodes since the number of nodes is much smaller than the number of objects (c.f. Figure 9 in Section 4.1).

#### 3.2 The algorithm IncAnyDBC

The algorithm IncAnyDBC consists of 6 steps. In Step 1, objects are summarized into object nodes of density connected objects. In Step 2, we build a graph  $G = (V, E)$  where each vertex is a node and each edge represents the connectivity status of two 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 are 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 way. 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 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

```

1  function C = IncAnyDBC (O, d, μ, ε, α, β)
2  input:  dataset O, distance function d, parameters μ, ε of DBSCAN,
3         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 S of α untouched objects to query randomly
9      for each object p in S do
10         if |Np| ≥ μ then st(p) = pcore else st(p) = pnoise
11         for each object q in Np do
12             if st(p) = pcore and st(q) = pnoise then st(q) = pborder
13             elseif st(p) = pcore and st(q) = untouched then st(q) = uborder
14             if st(q) = unprocessed then nei(q) = nei(q) + 1
15             if nei(q) ≥ μ and st(q) = uborder then st(q) = ucore
16         put p into V or L based on its core property
17     build a list Vp of nodes for each object p
18 /* Step 2: Build connectivity graph */
19 connect pairs of nodes (vp, vq) if d(p, q) ≤ 3ε (Lemma 3)
20 for each object p in O do
21     if p is a core then set the yes states for edges in Vp (Lemma 2)
22     else set the weak state for edges in Vp (Definition 8)
23 /* Iteratively update clustering results */
24 while true do
25     /* Step 3: Check stopping condition */
26     label nodes via their yes connected components
27     cont = false
28     for each cross-edge (vp, vq) do
29         if st(vp, vq) is unknown, weak, or link then cont = true (Lemma 5)
30     if cont = false then break
31 /* Step 4: Select objects for querying */
32 calculate node degrees for all nodes (Equation 1 and 2)
33 calculate object scores for all unprocessed objects (Equation 3)
34 choose a set S of β top scores objects
35 /* Step 5: Update graphs */
36 perform queries for all objects in S, mark the state, and increase the
37 neighborhood counts for objects as in Step 1
38 for each new core object p do
39     set the yes state for edges in Vp (Lemma 2)
40 for each core object p in S do
41     for each object q in Np do
42         if q is a core then st(Vp[1], Vq[1]) = yes (Lemma 2)
43         else st(Vp[1], Vq[1]) = link (Lemma 2)
44     for each cross-edge (vp, vq) do
45         if st(vp, vq) = weak or unknown then
46             if usize(vp) = 0 ∨ usize(vq) = 0 then st(vp, vq) = no (Lemma 6)
47             else if st(vp, vq) = link then
48                 if usize(vp) = 0 ∧ usize(vq) = 0 then st(vp, vq) = no (Lemma 6)
49 /* Step 6: Check the noise list */
50 for each object p in L do
51     check if p is a border object
52 end
    
```

Fig. 3. Pseudocode for IncAnyDBC

processed-noise (*pnoise*). However, in subsequent iterations, if one of its neighbors is a core,  $p$  is a border (c.f. Definition 2) and its 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  $\alpha$  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 addition, we set 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 6. 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  $p$  and all of its neighbors in  $N_p$ .

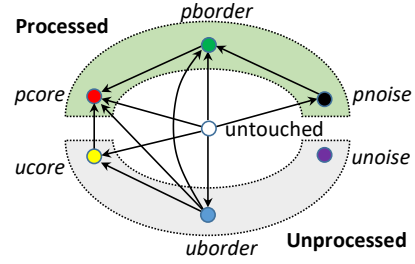


Fig. 4. The state transitions of objects

**Lemma 1.** ([14]) 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  $p$  is a core,  $st(q)$  will be changed to *uborder* or *pborder* if  $st(q)$  is *untouched* or *pnoise*, respectively. Moreover, if  $q$  is not processed, we increase its neighbor count  $nei(q)$  by 1, since  $q$  has  $p$  as its neighbor. If  $nei(q) \geq \mu$  and  $st(q)$  is *uborder*, we set  $st(q) = ucore$ .

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 (Lemma 1).

In the next Steps, IncAnyDBC performs additional queries on *unprocessed* objects to connect nodes into connected components, representing clusters.

**Definition 7.** (Directly-connected). Two object 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  $p \triangleleft x_1 \cdots x_m \triangleright q$ .

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.** Object nodes  $v_p$  and  $v_q$  are directly connected if:

- Case A: they share an object  $a$  where  $st(a) = ucore$  or  $st(a) = pcore$  (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 1.** Case A: We have  $p \triangleleft a$  and  $a \triangleleft q$  (Definition 3). Thus,  $p \triangleleft a \triangleright q$ . Case B: We have  $p \triangleleft a$  and  $b \triangleright q$  (Definition 3). Since  $a$  and  $b$  are core and  $d(a, b) \leq \epsilon$ , we have  $a \bowtie b$ . Thus,  $p \triangleleft a \triangleright b \triangleright q$ . Thus,  $p \Leftrightarrow q$  (Definition 7).

**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 2.** 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  $d(q, b) \leq \epsilon$  (Definition 6), we have  $d(a, b) > \epsilon$ . Thus,  $v_p$  and  $v_q$  will not be directly-connected (Definition 7).

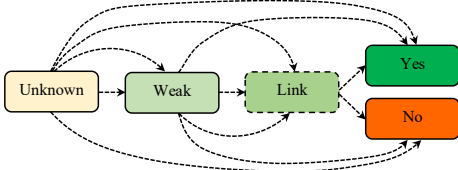


Fig. 5. The state transitions of edges

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

**Lemma 4.** If two core objects  $a$  and  $b$  are density connected in DBSCAN, then 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 3.** 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  $G$  by a path of edges.

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 for representing the connectivity status of two nodes  $v_p$  and  $v_q$ .

**Definition 8.** (Edge state). The state of an edge  $(v_p, v_q)$  ( $st(v_p, v_q)$ ), 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. 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 (in the same cluster). If  $st(v_p, v_q) = no$ ,  $v_p$  and  $v_q$  will not be directly-connected.

During the operation of IncAnyDBC, the states of edges change as summarized in Figure 5. The *link* state 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*, i.e., they are 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 ( $O(|V_p|)$  time complexity). We do not need to change all states of edges among nodes in  $V_p$  since this will takes  $O(|V_p|^2)$  time which is expensive while having unclear performance boost in our experiments.

**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  $st(v_p, v_q) \in \{weak, unknown, link\}$ , the cluster structure may change.

**Proof 4.** Since  $label(v_p) \neq label(v_q)$ ,  $st(v_p, v_q) \neq yes$ . 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 purpose of this step is to select *unprocessed* objects for processing so that the clusters are formed quickly, i.e., more *yes* edges created 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 to rank objects via 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:

$$deg(v_p) = \sum_{v_q \in adj(v_p)} \omega(v_p, v_q) stat(v_q) \quad (1)$$

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*);  $\omega(v_p, v_q)$  is the predefined weight for each edge based on its state (1, 2 and 4 for *unknown*, *link*, and *weak*, resp.); and  $stat(v_p)$  is the current processing score of  $v_p$ :

$$stat(v_p) = \left(1 - \frac{|N_p|}{n}\right) + \frac{usize(v_p)}{|N_p|} + \psi(v_p) \quad (2)$$

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  $deg(v_p)$  measures the uncertainty of  $v_p$  wrt. the current structure. If  $v_p$  is lying closer to borders of many clusters (has larger  $|adj(v_p)|$ ) or contains a *pborder* 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 (Lemma 2) or will break some undetermined edges faster (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 have 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:

$$score(p) = \frac{1}{|V_p|} \sum_{v_q \in V_p} deg(v_q) \quad (3)$$

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 current cluster structure toward the final one.

At the end of Step 4, we choose a set  $S$  of  $\beta$  objects with highest scores for processing in Step 5.

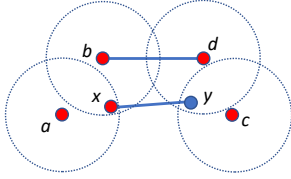


Fig. 6. Illustration of Lemma 7

**Step 5: Update graphs.** In this step, we performing queries on all  $\beta$  selected objects. For each object  $p$ , we mark  $st(p)$  as  $pcore$  if it a core or  $pborder$  otherwise. We set  $lev(p) = n$ . We also increase the number of neighbors  $nei(q)$  for each unprocessed object  $q \in N_p$  and set new states for them 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  $st(v_p, v_q) \in \{weak, unknown, link\}$ , if  $usize(v_p) = 0$  and  $usize(v_q) = 0$ ,  $v_p$  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 5.** Assume that their exists a chain of objects  $x_i \in v_p \cup v_q$  so that  $p \triangleleft x_1 \cdots x_m \triangleright q$ . 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 prove that this scheme guarantees a correct clustering result.

**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, two object nodes  $v_a$  and  $v_c$  will be put in the same cluster as in DBSCAN.

**Proof 6.** (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 two object nodes  $v_a$  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 in Step 3 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 object found,  $p$  is surely a noise.

**Correctness.** When it reaches its final stage, IncAnyDBC produces the identical results as DBSCAN. Shared border objects may be labeled differently in both DBSCAN and IncAnyDBC based on the examining orders of objects.

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

**Proof 7.** (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 not be directly-connected. Consequently,  $label(a) \neq label(b)$  in IncAnyDBC. Step 6 of IncAnyDBC ensures that we do not miss any border objects.

**Monotonicity.** Since IncAnyDBC merges nodes with new queries, the number of clusters decreases at each iteration.

**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

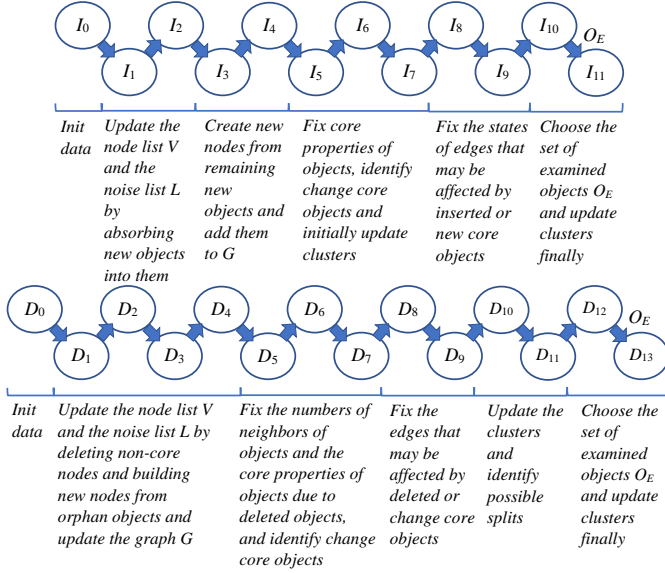


Fig. 7. Different Steps of IncAnyDBC for the insertion (top) and deletion (bottom) cases

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. If  $v$  and  $l$  are  $O(n)$ ,  $e = v^2$  and  $\beta = O(n)$ , 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 *full* 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  $m \ll n$  (where  $n = |O|$ ,  $m = |B|$ ), 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 in Section 2. IncAnyDBC first updates the current noise list  $L$  and the node list  $V$  in Step I1 and I2. Then it creates new nodes for new objects if it is necessary in Step I3. The graph  $G$  is updated for reflexing changes in Step I4 to I10. And the clusters are updated in Step I11 (c.f. Figure 7 (top)).

**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 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  $\alpha$  *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 in the next steps).

**Step I5: Identify change core objects.** 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 neighborhoods of new non-core objects in  $L$ .

**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. E.g., all objects in  $v_u$  and  $v_t$  (Figure 2) will not change their core properties due to the inserted object  $a$ . 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 object, we add their nodes to the list of change core nodes  $V^2$  together with all newly created nodes in Step I1 (since their centers change from noise to 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, its 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  $v_p \notin V^A$  and  $v_q \notin V^A$ ,  $st(v_p, v_q)$  will not be affected by new objects.

**Proof 8.** (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 may 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 contribute to the connectivity. Thus if it becomes a core, it will not cause any change.

E.g., in Figure 2, edges  $(v_t, v_u)$  and  $(v_r, v_t)$  are not affected by the object  $a$ . Following Lemma 10, for updating clusters, an obvious way is resetting 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 by 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  $G$  wrt. 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  $d(a, q) > 2\epsilon$ ,  $a$  itself does not change the state of  $(v_p, v_q)$  directly.

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  $2\epsilon$  (as demonstrated in Figure 2). If  $M_q^{2\epsilon}$  does not contain a core 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 (which will be processed in Step I9)). For each object  $o \in O^{1A}$ , we perform a reverse query on  $o$  to get new neighbors  $M_o$  and use them 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 in Section 3.2.

**Step I9: Fix the links in  $G$  wrt. change core objects.** Similar to new objects, change core ones cause clusters to be merged as in Lemma 2. 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  $d(o, p) > 2\epsilon$ , the object  $o$  does not change  $st(v_p, v_q)$ .

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 cluster structure 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 Lemmas 2 and 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 IncAnyDBC to re-perform queries on these objects to update clusters in Step I11.

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

Concretely, at each iteration, we choose a set of  $\beta$  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.

**Lemma 13.** (Correctness). IncAnyDBC produces identical results to those of DBSCAN after insertions.

**Proof 9.** (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  $b \in v_y$  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. It consumes  $O(vn + v^2 + l\mu + nm)$  overall space complexity.

### 3.3.2 Deletion

In the deletion case, some objects may lose their core property, thus leading to the split of clusters [3]. In Step D1 and D2 (Figure 7), we update the non-core list  $L$  and the node list  $V$  by removing deleted objects out. Due to deleted nodes, orphan objects that are not covered inside any nodes will be grouped again into new nodes or placed into the non-core list in Step D3. Steps D4 to D10 update the connectivity of the graph  $G$  wrt. changes. Step D11 identifies if splits occur in clusters. All objects that may involve to cluster changes are captured in Step D12. We update clusters in Step D13 following the active clustering scheme.

**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* one.

**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  $ptou(p) = 1$ . E.g., the non-core object  $e$  in Figure 2 will be removed since  $e$  is deleted.

**Step D2: Update the node list.** In contrast to the non-core ones, objects inside the node list  $V$  may lose 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* orphan objects.

At each iteration, we choose a set  $A$  of  $\alpha$  *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. Let  $L^1$  be a set of deleted objects inside  $L$  and  $V^1$  be the set of nodes containing deleted objects acquired from Step D2. Let  $O^A$  be the set of objects in  $\cup_{v_p \in V^1} adj(v_p) \vee \cup_{p \in L^1} N_p$ , where  $adj(v_p)$  is the set of adjacent nodes of  $v_p$  including itself.

**Lemma 14.** All object  $o \notin O^A$  will not be affected by deleted objects.

Lemma 14 is directly inferred from the triangular inequality in Lemma 3. Each object  $o \in O^A$  may be inside the neighborhood of a deleted object and thus their number of neighbors  $nei(o)$  may change. Thus, for each object  $o \in O^A$ , we perform a reverse query to get deleted objects  $M_o$  around  $o$ . For each object  $p \in M_o$ , we decrease  $nei(p)$  if  $p \geq level(o)$  and  $o \geq level(p)$ . This ensures that no non-core objects is recognized as cores, causing false *yes* connections and a wrong clustering result consequently.

**Step D6: Identify change core objects.** 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  $st(o) = pcore$  or  $st(o) = ucore$  and  $nei(p) < \mu$ , and we put all nodes  $V_o$  of  $o$  into the set of change core nodes  $V^2$  for further processing.

**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 14, 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 ( $ptou(o) = 0$ ), 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 re-run the clustering algorithm to re-build clusters. However, this still incurs many 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 15.** Given a deleted object  $a \in v_p$ , if  $d(a, q) > 2\epsilon$ ,  $a$  itself does not break the *yes* state of  $(v_p, v_q)$  directly.

E.g., if we delete the object  $a$  in Figure 2, the *yes* state of  $(v_p, v_q)$  remains since  $d(a, v) > 2\epsilon$ . Following Lemma 15, we perform a reverse query on deleted objects for each  $v_q \in V^{1A}$  with a threshold of  $2\epsilon$ . 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 deleted objects.

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

Following Lemma 16, 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  $st(v_p, v_q) = unknown$ , waiting for this edge to be re-updated.

**Step D10: Update cluster structures.** For each object  $o \in O$  and  $o \notin B$ , 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 Steps D8 and D9.

**Step D11: Detect possible splits.** Given two arbitrary object 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 17.** Any cluster  $c \notin C^A$  will not be affected by the deletions.

**Proof 10.** (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 by deleted objects.

For each cluster  $c \in C^A$ , we need to re-cluster it to check if  $c$  is really be splitted. To do so, we 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 splitted 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 17 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 (insertion case).

**Lemma 18.** (Correctness). IncAnyDBC produces identical results to those of DBSCAN after the deletions.

**Proof 11.** (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.

**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. It requires  $O(vn + v^2 + l\mu + nm)$  space complexity.

### 3.4 Parallel processing

Figure 8 shows the parallel model of IncAnyDBC. At each iteration, a block of unprocessed objects are selected from the database for processing 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., threads  $t_1, t_2$ , and  $t_3$  process object  $a, b$ , and  $c$ , respectively. This is more effective than executing each query in parallel, especially

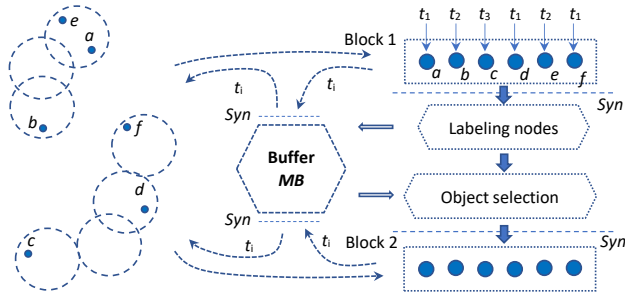


Fig. 8. IncAnyDBC’s parallel processing model on multicore CPUs

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 propose to wait for all queries to be completed before updating the information of objects and connectivity among nodes. Otherwise, many critical sessions (locks) will be required to ensure consistency, thus significantly reducing the scalability. 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. In our preliminary experiment, without using the buffer, the scalability of the algorithm is significantly reduced (1.5-2x). Since the neighborhood sizes of objects vary, we 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 its end.

Instead of propagating labels among 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.

IncAnyDBC needs to hold a list of nodes  $V_p$  for each object  $p$ . Since each object may belong to many different nodes, naively parallelizing this task will lead to performance degradation since many synchronizations need to be used. 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  $p$  in parallel. By this way, we only need to synchronize all threads one time while still having the same workload as the sequential algorithm. Thus, it helps to improve the scalability considerably. However, balancing the workload is still a hard problem since the neighborhood sizes of objects can vary significantly though we use dynamic scheduling to try to balance threads’ workload.

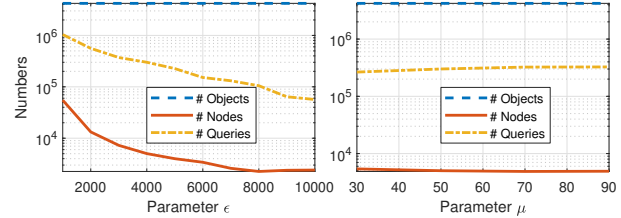


Fig. 9. The numbers of object nodes and queries for GasSensor dataset

## 4 EXPERIMENTS

**Datasets.** We perform experiments on 7 real datasets acquired from different sources including: (1) Farm: contains 3,627,086 objects. Each has 5 VZ-features of a satellite image in SaudiArabia<sup>1</sup> [12]; (2) Household: has 2,049,280 objects with 7-D data for electricity and mortgage expenses in US [15]; (3) Sdss2Mass: contains 1,258,127 8-D objects describing locations and gravities of different galaxies [16]; (4) GasSensor: records values of 16 different sensors exposed to Ethylene-CO with 4,178,504 objects [15]; (5) PAMAP2: describes the physical activities using inertial measurement units and is acquired from the UCI archives [15] with 974,479 39-D objects; (6) Precipitation: contains data of mean monthly surface climate such as precipitations and temperatures over global land areas<sup>2</sup> with 566,268 12-D objects; and (7) OSFP: acquired from the French national registry of sleep apnea<sup>3</sup>. It describes sets of syndromes for 39,252 patients with Obstructive Sleep Apnea (OSA).

**Systems.**<sup>4</sup> Experiments are conducted on Linux server with two 8-core CPUs (Intel Xeon E5-2650 v2 - 2.6 GHz) and 128GB RAM (64 GB/CPU) using g++ 4.9.2 and OpenMP<sup>5</sup>.

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

### 4.1 Clustering performance

Unless otherwise stated, we use default parameters  $\mu = 50$ ,  $\alpha = 512$ , and  $\beta = 4,096$ .

**The pruning power of IncAnyDBC.** Figure 9 shows the numbers of queries and object nodes of IncAnyDBC for the dataset GasSensor with different parameters  $\mu$  and  $\epsilon$ . IncAnyDBC requires much fewer queries to build clusters 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 1,866.1 times). The bigger the values of  $\epsilon$  and  $\mu$  (and the denser and well-separated clusters), the lower the numbers of nodes. The same results are observed for all the datasets. Consequently, IncAnyDBC is much faster than DBSCAN as shown below.

**Performance comparisons.** Figure 10 shows the performance of IncAnyDBC compared to DBSCAN [7] and its fastest variants including  $\rho$ -DBSCANv2 [11] and AnyDBC [14], [17] using different parameters  $\epsilon$  and  $\mu$ . As suggested from [18], [19], we vary the parameter  $\epsilon$  from very small to

1. <https://tinyurl.com/ikonos-tadco-farms>
2. <http://www.cru.uea.ac.uk/>
3. <http://www.osfp.fr>
4. [https://nqvhung.github.io/multicore\\_dbscan/](https://nqvhung.github.io/multicore_dbscan/)
5. <https://www.openmp.org/>

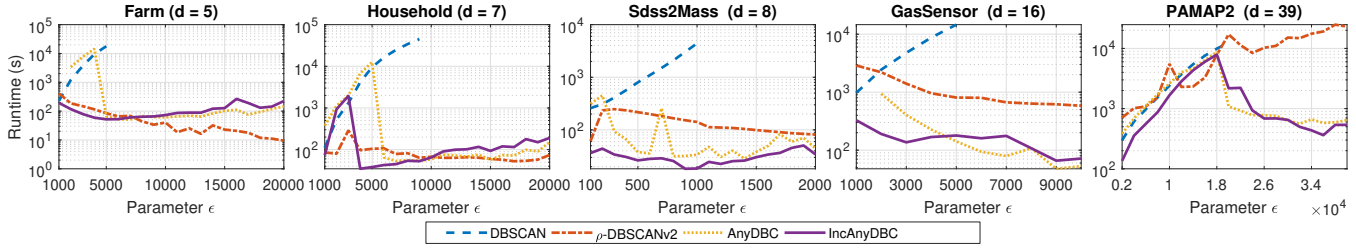


Fig. 10. Clustering performance of IncAnyDBC on real datasets (we only run DBSCAN with some parameters  $\epsilon$  for saving times. AnyDBC ran out of memory when  $\epsilon = 1,000$  for the datasets Farm and GasSensor)

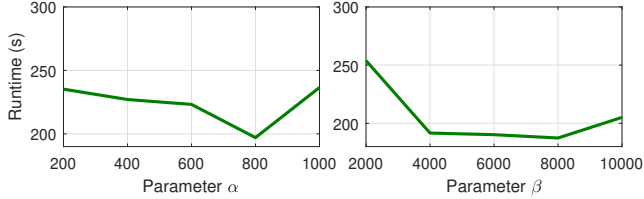


Fig. 11. The effects of parameters  $\alpha$  and  $\beta$  on GasSensor ( $\epsilon = 2,000$ )

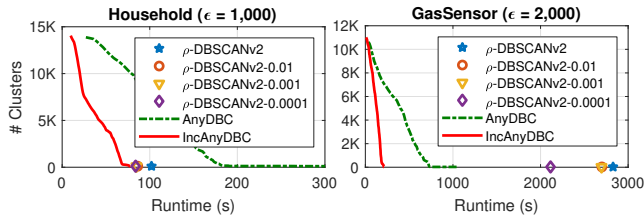


Fig. 12. Anytime properties of IncAnyDBC on the Household and GasSensor datasets

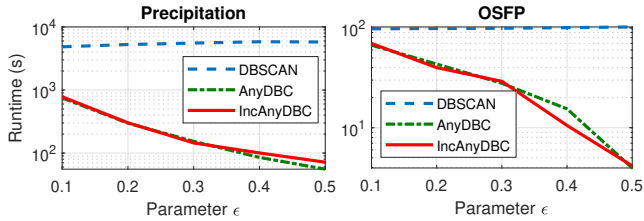


Fig. 13. Performance of IncAnyDBC on the dataset Precipitation (Manhattan distance) and OSFP (Jaccard distance)

very large to study the performance of these algorithms. For example, when  $\epsilon = 100$ , 79% objects are noise for Sdss2Mass. For Household, there is only one cluster containing 99.99% objects when  $\epsilon = 20,000$ . For indexing, we use  $kd$ -tree implementation from [20].

Compared to DBSCAN, IncAnyDBC is much faster in most cases due to its pruning power. E.g., the speedup factor ranges from 7.0 to 238.5 times for Sdss2Mass and from 0.57 to 853.5 times for Household. However, when the number of used queries is too large (e.g., when  $\epsilon = 2,000$  for Household), IncAnyDBC will run slightly slower than DBSCAN due to its active clustering overheads such as object selections in Step 4. When  $\epsilon$  is large, AnyDBC and IncAnyDBC acquires comparable performance on all the datasets. However, when  $\epsilon$  is very small, both IncAnyDBC and AnyDBC need to spend more queries to go to the termination stage as demonstrated in Figure 9. Thus, the overhead increases. However, since IncAnyDBC does not need to merge clusters and queries like AnyDBC, its overhead is much smaller. Thus it is much faster. When the dimension  $d$  of the data is low (e.g., the Farm and Household datasets), the performances of IncAnyDBC and  $\rho$ -DBSCANv2 are comparable. However, when  $d$  is larger, IncAnyDBC runs much faster since it does not rely on

the grid structure like  $\rho$ -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  $\rho$ -DBSCANv2 on the datasets Sdss2Mass, GasSensor, and PAMAP2, respectively.

**Effects of parameter  $\alpha$  and  $\beta$ .** Generally, when  $\alpha$  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. 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. Clarifying these *cross-edges* requires more queries to be performed, thus decreasing performances. The parameter  $\beta$  is used for balancing the overheads of IncAnyDBC and its pruning power. Smaller  $\beta$  means that better objects are selected to build clusters (Steps 3 to 5). Thus, fewer queries are required compared to bigger values of  $\beta$ . However, 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. It typically goes down and up again when increasing  $\alpha$  and  $\beta$ . We suggest to set  $\alpha$  from 400 to 800 and  $\beta$  from 4,000 to 8,000 in our experiments.

**Anytime properties.** One major advantage of IncAnyDBC is that it can be interrupted at any time to provide approximate results, while other methods like [7], [11], [12] can only provide either an exact result or an approximate result as shown in Figure 12. Due to the monotonicity property (c.f. Section 3.2), the number of clusters reduces 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.

**Other distance functions.** While other grid-based methods like [11]–[13] can only work under Euclidean distance, IncAnyDBC can work under arbitrary distance metrics. Figure 13 shows the performance of IncAnyDBC and AnyDBC on Precipitation and OSFP using Manhattan and Jaccard distance metrics [21] ( $\alpha = 128$  and  $\beta = 1,024$ ). IncAnyDBC is comparable to AnyDBC 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 [17], HPDBSCAN [22], and PDSDBSCAN [20] on different datasets using 16 threads. Due to its grid-based scheme, HPDBSCAN can only work on low dimensional datasets Farm and Household

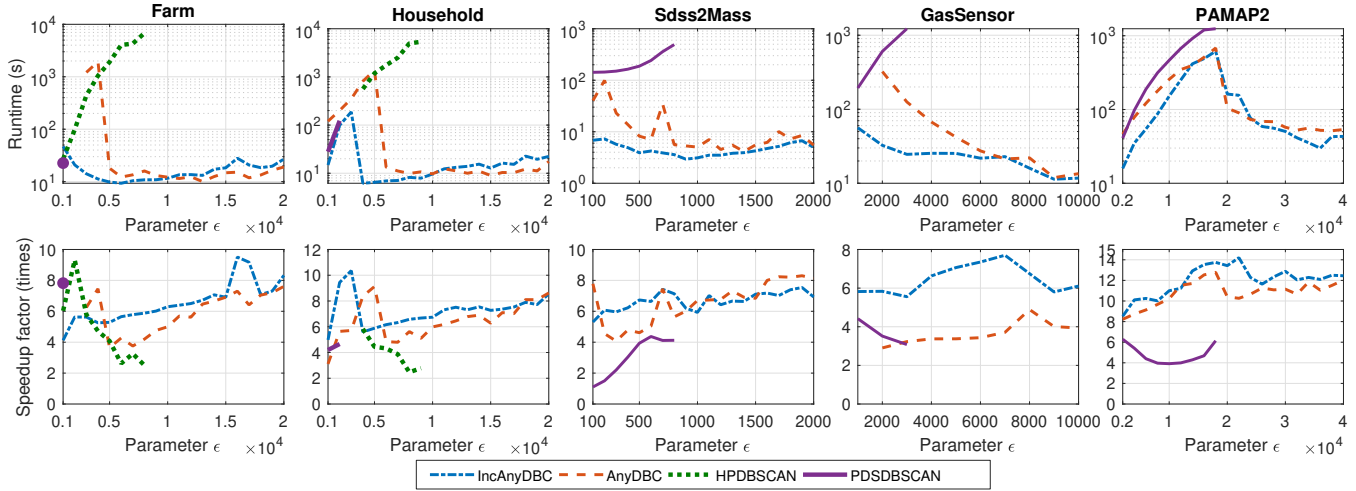


Fig. 14. Runtimes (top) and speedup factors (bottom) of different algorithms using 16 threads. HPDBSCAN only works for the dataset Farm and Household when  $\epsilon$  is large enough. PDSDBSCAN runs out of memory when  $\epsilon$  is large. AnyDBC runs out of memory when  $\epsilon = 1,000$  and  $2,000$  (Farm) and  $\epsilon = 1,000$  (GasSensor)

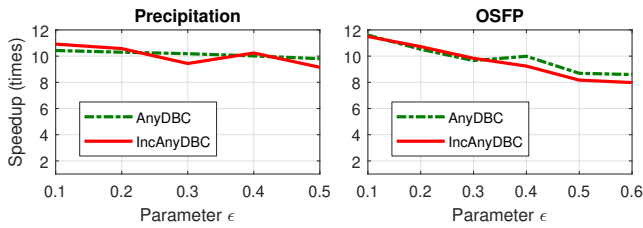


Fig. 15. Scalability of IncAnyDBC on the dataset Precipitation (Manhattan distance) and OSFP (Jaccard distance) using 16 threads

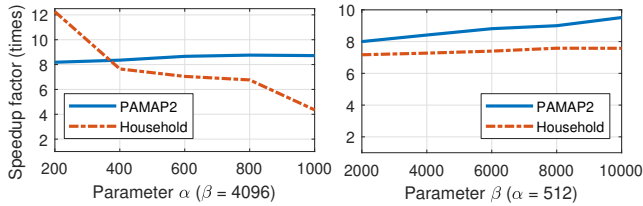


Fig. 16. Effects of parameters  $\alpha$  and  $\beta$  on the scalability of IncAnyDBC using 16 threads for PAMAP2 ( $\epsilon = 2,000$ ) and Household ( $\epsilon = 12,000$ )

with large values of  $\epsilon$  (e.g.,  $\epsilon \geq 4,000$  for Household). PDSDBSCAN, on the other hand, consumes too much memory due to its object storing scheme when the neighborhoods of objects overlap. Thus, when  $\epsilon$  is large enough, it runs out of memory (e.g.,  $\epsilon > 1,000$  for Household). Since both HPDBSCAN and PDSDBSCAN do not focus on workload reduction like IncAnyDBC, their performance is much worse than that of 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 IncAnyDBC on the GasSensor dataset. The bigger the value of  $\epsilon$ , the larger the performance gap. Compared to AnyDBC, IncAnyDBC is faster in most cases<sup>6</sup>, especially when  $\epsilon$  is small, e.g., 55.4 times when  $\epsilon = 2,000$  for GasSensor and 212.4 times when  $\epsilon = 5,000$  for Household. Moreover, since AnyDBC uses bit vectors to merge clusters and queries, it consumes much memory than IncAnyDBC, e.g., up to 19.7 times on Household. In terms of scalability, IncAnyDBC also performs better than AnyDBC in most cases 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

6. We slightly modified AnyDBC to make it faster. However, its scalability becomes slightly worse than the original version [17].

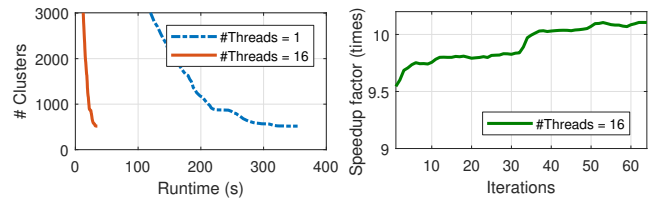


Fig. 17. Performance at each step of IncAnyDBC for the PAMAP2 ( $\epsilon = 4,000$ ) using 16 threads

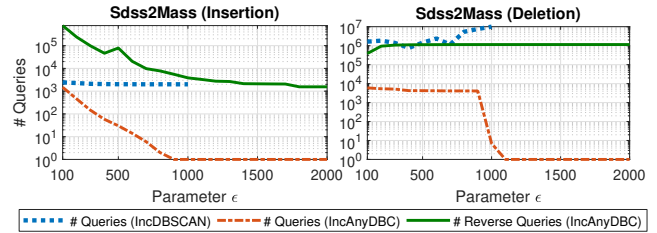


Fig. 18. Numbers of queries and reverse queries of IncAnyDBC and IncDBSCAN for Sdss2Mass

the datasets Farm, Household, Sdss2Mass, GasSensor, and PAMAP2, respectively.

**Other distance functions.** Figure 15 shows the scalability of IncAnyDBC using 16 threads on Precipitation ( $L_1$ ) and OSFP (Jaccard distance). It has very good performance and comparable to AnyDBC. The speedup factors are from 9.1 to 10.9 (Precipitation) and from 8.1 to 11.5 (OSFP). HPDBSCAN [22] and PDSDBSCAN [20] can only work on Euclidean distance and are thus excluded.

**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, using 16 threads, we can acquire the same clustering result at each iteration of IncAnyDBC faster (9.5 to 10.1 times).

**Effects of parameters  $\alpha$  and  $\beta$ .** Figure 16 shows the effects of parameters  $\alpha$  and  $\beta$  on the performance of IncAnyDBC. Increasing  $\beta$  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  $\alpha$ , however, is unclear. On different datasets, it shows different behaviours. E.g., when  $\epsilon$  increases from 200 to 1,000, the speedup factor increases from 8.1 to 8.7 times on PAMAP2 but decreases significantly

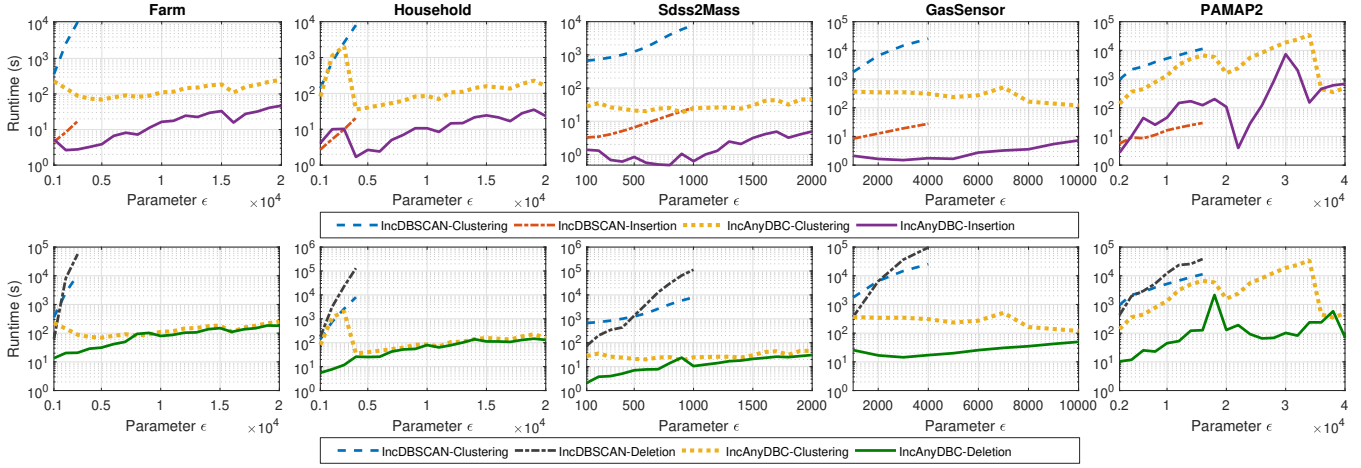


Fig. 19. Performance of IncAnyDBC and IncDBSCAN for various real datasets

from 12.2 to 4.3 times on Household. Unfortunately, there is no known way to predict such behaviors.

### 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  $\mu = 20$ ,  $\alpha = 512$ ,  $\beta = 4,096$ , and  $m = 2,000$ .

**The query pruning scheme of IncAnyDBC.** Figure 18 (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  $\epsilon$ . 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 19. Moreover, when  $\epsilon$  grows bigger, the cluster structure tends to be more stable and there are fewer border objects (that may change to cores). Thus, the number of queries is typically reduced.

The deletion case is much expensive than the insertion case shown in Figure 18 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 (from 180.3 to 1,442,212 times lower than IncDBSCAN). This dramatically improves the performance as shown in Figure 19. The bigger the  $\epsilon$ , the fewer queries it uses typically.

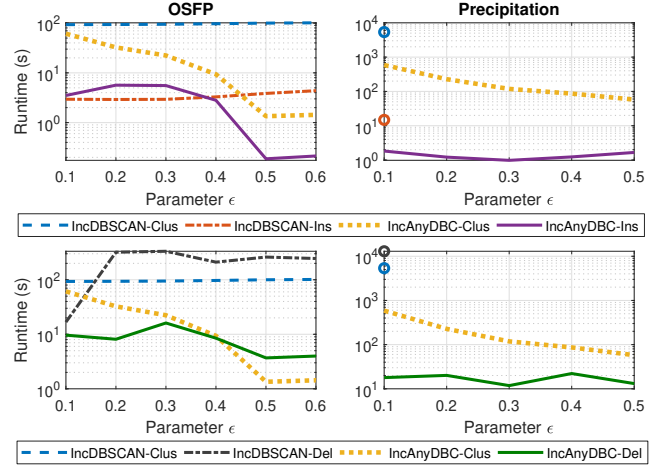


Fig. 20. Performance of IncAnyDBC and IncDBSCAN for datasets Precipitation and OSFP

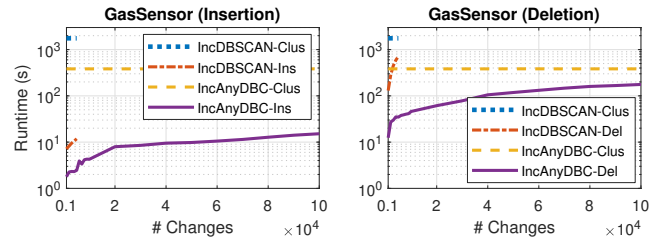


Fig. 21. Performance of IncAnyDBC wrt. the numbers of data changes for GasSensor ( $\epsilon = 1,000$ )

**Performance comparisons.** Figure 19 (top) shows the runtimes of IncAnyDBC and cumulative runtimes IncDBSCAN [3] over 2,000 insertions. When  $\epsilon$  becomes bigger, 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  $\epsilon$ , 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 18 and 19 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

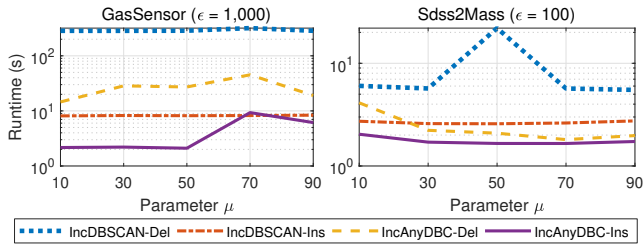


Fig. 22. Effects of the parameter  $\mu$

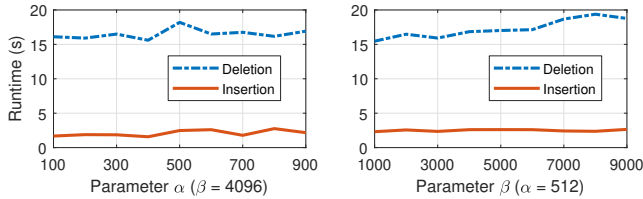


Fig. 23. Effects of the parameter  $\alpha$  and  $\beta$  for Sdss2Mass ( $\epsilon = 500$  and  $m = 10,000$ )

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 19 (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 (both insertions and deletions) fully dominates IncDBSCAN, e.g., from 22.5 to 10,172.8 times for Sdss2Mass. With  $m = 2,000$  changes, updating clusters using IncAnyDBC is also more efficient than re-clustering the whole database using IncAnyDBC (e.g., from 2.1 to 21.7 times on GasSensor) or DBSCAN (e.g., from 13.3 to 5,134.2 times on GasSensor).

**Other distance metrics.** Figure 20 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.

**Scalability wrt. the numbers of object changes?** Figure 21 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  $\epsilon$ , 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 re-clustering 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  $\mu$  and  $\epsilon$ .** Figure 19 and Figure 22 show the effects of parameters  $\mu$  and  $\epsilon$  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  $\epsilon$  and increases with  $\mu$  as seen in Figures 9 and 18.

**Effect of parameters  $\alpha$  and  $\beta$ .** Figure 23 shows the robust-

ness of IncAnyDBC over two parameters  $\beta$  and especially  $\alpha$ . When  $\alpha$  varies from 100 to 900 and  $\beta$  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  $\beta$  due to redundant queries during the re-clustering phases I11 and D13 of IncAnyDBC. Changes caused by  $\alpha$  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 seen in Figure 28. 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 24 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  $\epsilon$ , large values of  $\epsilon$  typically bring up higher speedup factors due to the increasing of the parallelizable 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.

**Scalability for other data.** On OSFP and Precipitation, IncAnyDBC still has very good scalability as seen in Figure 25. However, the speedup factor decreases with  $\epsilon$ . The reason is that we do not use indexing technique. Thus, the query processing time is well-balanced for threads regardless of  $\epsilon$ . In this case, bigger  $\epsilon$  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 26, 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  $\alpha$  and  $\beta$ .** When  $\alpha$  and  $\beta$  increases as demonstrated in Figure 27, the scalability of IncAnyDBC over 16 threads remains quite stable, especially for the deletion case. Theoretically, increasing  $\beta$  will balance workload of threads better, thus leading to better speedup factor as we have seen in Figure 16. 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



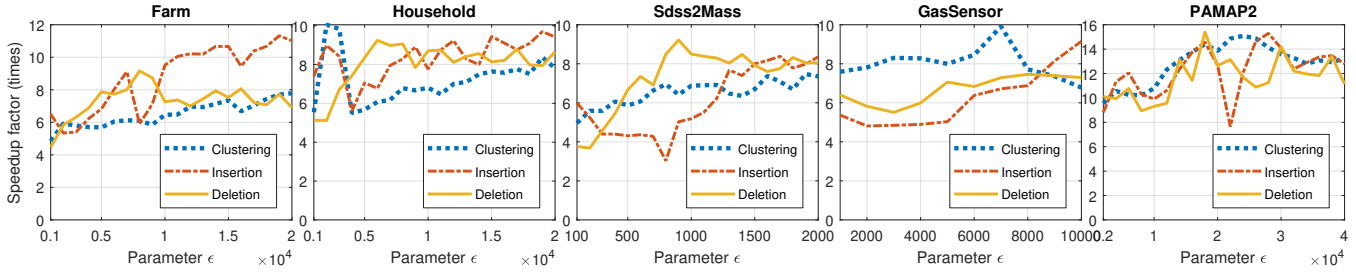


Fig. 24. Scalability of IncAnyDBC over 16 threads for various real datasets

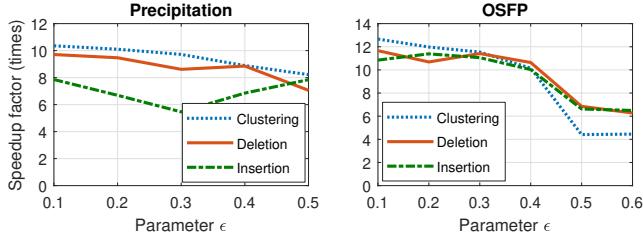


Fig. 25. Scalability over 16 threads of IncAnyDBC for datasets Precipitation and OSFP

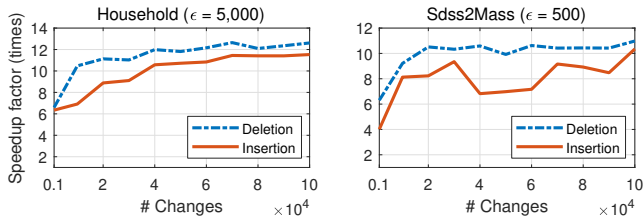


Fig. 26. Scalability of IncAnyDBC using 16 threads for Household and Sdss2Mass

intermediate results faster as shown in Figure 28. 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 has been an important research focus for many years [3], [4], [6], [23]–[27]. Most work focuses on incrementally updating existing clusters when changes occur in the databases instead of reclustering from scratch.

In [3], the locality nature of DBSCAN is exploited to limit the update areas, thus saving computation costs. Gan and Tao [6] 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 [3] and  $\rho$ -DBSCAN [6] 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 can work under arbitrary time constraints and can provide both exact and approximate results of DBSCAN. Besides, it can work with arbitrary distance metrics like IncDBSCAN. Density-based methods for streaming data such as DenStream [28] and [29] has an incremental nature like IncAnyDBC. However,

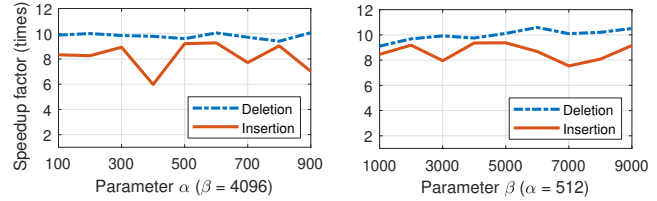


Fig. 27. Effects of the parameter  $\alpha$  and  $\beta$  on the scalability of IncAnyDBC using 16 threads for Sdss2Mass ( $\epsilon = 500$  and  $m = 10,000$ )

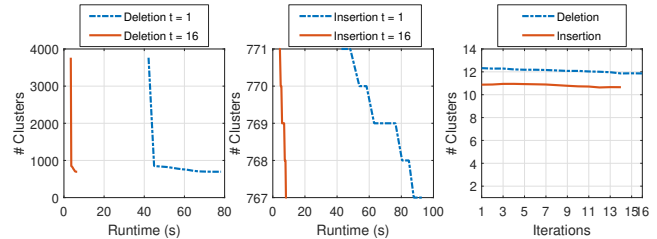


Fig. 28. Runtime, number of clusters, and scalability at each iteration of IncAnyDBC for PAMAP2 ( $\epsilon = 2000$ ,  $m = 100,000$ )

they follow different cluster notions 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.**  $\rho$ -DBSCAN [12] and other methods such as [11], [13], [30], [31] rely on different grid-based schema to improve the performance of DBSCAN. However, since the number of cells grows exponential with the data dimension, they all suffer from performance degradation for high dimension data. Moreover, they can only work with Euclidean distance. Other methods such as [32]–[34] 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 [14] is the closest work to IncAnyDBC. Both of them are based on the active clustering approach for reducing the used queries. However, they 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. Moreover, AnyDBC loses important information on local subgroup connections that can be exploited to efficiently update clusters as presented in IncAnyDBC.

There are many other methods aiming at improving DBSCAN's performance such as BRIDGE [35], IDBSCAN [36] and DBR [37]. However, they can only produce (coarse) approximate results instead of exact results of DBSCAN like IncAnyDBC. On the other hand, HDBSCAN [38], [39] does not focus on runtime improvement but it tries to extract high quality clusters from density-based clustering hierarchy.

**Parallel density-based clustering.** Parallelizing DBSCAN is one of the most active research topics for enhancing DBSCAN with many proposed techniques such as [17], [20], [22], [40]–[50]. Most of them focus on parallelizing DBSCAN on distributed systems including MPI-based [40], [44], [51], MapReduce [45], [46], or Spark [41], [50], [52]. Few algorithms aim at extending DBSCAN on Graphic Processing Units (GPUs) such as [43], [47], [48]. There are some methods that are designed to work with multicore CPUs including [17], [20], [22], [42].

PDSDBSCAN [20] 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 much higher workload than IncAnyDBC. Pardicle [42] 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 [17] 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 in its parallel mode, 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 efficiently 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 fewer queries to build the same clustering results as DBSCAN. Thus, it is orders of magnitude faster than DBSCAN and its variants. IncAnyDBC reserves local cluster structures of data and exploits them to *actively* update clusters when there are changes in the database such as inserted or deleted objects. Thus, it needs much fewer queries than the state-of-the-art method IncDBSCAN for updating results. Moreover, changes are enforced in *bulks* rather than *batches* like existing techniques for reducing overhead. 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 demand. Its *block* processing scheme allows it to be parallelized efficiently on multicore CPUs. Experiments with 16 CPU cores show that IncAnyDBC scales very well with the number of threads (up to  $\approx 15$  times over 16 threads).

## ACKNOWLEDGMENTS

We special thank to anonymous reviewers for their helpful comments that significantly improve the paper. This work is supported by the French National Research Agency (ANR-15-IDEX-02), a Villum postdoc fellowship and a Newton Fund (ID 528154944).

## REFERENCES

- [1] C. C. Aggarwal and C. K. Reddy, Eds., *Data Clustering: Algorithms and Applications*. CRC Press, 2014.
- [2] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques, Third Edition*. Morgan Kaufmann Publishers, 2012.
- [3] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu, "Incremental Clustering for Mining in a Data Warehousing Environment," in *VLDB*, 1998, pp. 323–333.
- [4] P. Bhattacharjee and A. Awekar, "Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for Dynamic Datasets," in *ECIR*, 2017, pp. 568–574.
- [5] S. Singh and A. Awekar, "Incremental shared nearest neighbor density-based clustering," in *CIKM*, 2013, pp. 1533–1536.
- [6] J. Gan and Y. Tao, "Dynamic Density Based Clustering," in *SIGMOD*, 2017, pp. 1493–1507.
- [7] 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*, 1996, pp. 226–231.
- [8] J. Lee, J. Han, and K. Whang, "Trajectory clustering: a partition-and-group framework," in *SIGMOD*, 2007, pp. 593–604.
- [9] S. T. Mai, S. Goebel, and C. Plant, "A Similarity Model and Segmentation Algorithm for White Matter Fiber Tracts," in *ICDM*, 2012, pp. 1014–1019.
- [10] D. Q. Phung, B. Adams, S. Venkatesh, and M. Kumar, "Unsupervised context detection using wireless signals," *Pervasive and Mobile Computing*, vol. 5, no. 6, pp. 714–733, 2009.
- [11] J. Gan and Y. Tao, "On the Hardness and Approximation of Euclidean DBSCAN," *ACM Trans. Database Syst.*, vol. 42, no. 3, pp. 14:1–14:45, 2017.
- [12] —, "DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation," in *SIGMOD*, 2015, pp. 519–530.
- [13] A. Gunawan, "A Faster Algorithm for DBSCAN," Msc Thesis, TU Eindhoven, 2013.
- [14] S. T. Mai, I. Assent, and M. Storgaard, "AnyDBC: An Efficient Anytime Density-based Clustering Algorithm for Very Large Complex Datasets," in *KDD*, 2016, pp. 1025–1034.
- [15] A. Frank and A. Asuncion, "UCI machine learning repository." [Online]. Available: <http://archive.ics.uci.edu/ml>
- [16] G. De Lucia and J. Blaizot, "The hierarchical formation of the brightest cluster galaxies," *Monthly Notices of the Royal Astronomical Society*, vol. 375, no. 1, pp. 2–14, 2007.
- [17] S. T. Mai, I. Assent, J. Jacobsen, and M. S. Dieu, "Anytime parallel density-based clustering," *Data Min. Knowl. Discov.*, vol. 32, no. 4, pp. 1121–1176, 2018.
- [18] 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.*, vol. 42, no. 3, pp. 19:1–19:21, 2017.
- [19] H. Kriegel, E. Schubert, and A. Zimek, "The (black) art of runtime evaluation: Are we comparing algorithms or implementations?" *Knowl. Inf. Syst.*, vol. 52, no. 2, pp. 341–378, 2017.
- [20] 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*, 2012, p. 62.
- [21] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining, (First Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.
- [22] M. Götz, C. Bodenstern, and M. Riedel, "HPDBSCAN: highly parallel DBSCAN," in *MLHPC*, 2015, pp. 2:1–2:10.
- [23] M. Charikar, C. Chekuri, T. Feder, and R. Motwani, "Incremental Clustering and Dynamic Information Retrieval," in *STOC*, 1997, pp. 626–635.
- [24] M. Ackerman and S. Dasgupta, "Incremental Clustering: The Case for Extra Clusters," in *NIPS*, 2014, pp. 307–315.
- [25] J. Lin, M. Vlachos, E. J. Keogh, and D. Gunopulos, "Iterative Incremental Clustering of Time Series," in *EDBT*, 2004, pp. 106–122.

- [26] S. T. Mai, M. S. Dieu, I. Assent, J. Jacobsen, J. Kristensen, and M. S. Birk, "Scalable and Interactive Graph Clustering Algorithm on Multicore CPUs," in *ICDE*, 2017, pp. 349–360.
- [27] 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.*, vol. 31, no. 7, pp. 1239–1252, 2019.
- [28] F. Cao, M. Estert, W. Qian, and A. Zhou, "Density-based clustering over an evolving data stream with noise," in *SDM*. SIAM, 2006, pp. 328–339.
- [29] L. Wan, W. K. Ng, X. H. Dang, P. S. Yu, and K. Zhang, "Density-based clustering of data streams at multiple resolutions," *ACM Trans. Knowl. Discov. Data (TKDD)*, vol. 3, no. 3, p. 14, 2009.
- [30] T. Sakai, K. Tamura, and H. Kitakami, "Cell-Based DBSCAN Algorithm Using Minimum Bounding Rectangle Criteria," in *DASFAA Workshop*, 2017, pp. 133–144.
- [31] T. Li, T. Heinis, and W. Luk, "Hashing-Based Approximate DBSCAN," in *ADBIS*, 2016, pp. 31–45.
- [32] S. Brecheisen, H. Kriegel, and M. Pfeifle, "Efficient Density-Based Clustering of Complex Objects," in *ICDM*, 2004, pp. 43–50.
- [33] C. Böhm, J. Feng, X. He, and S. T. Mai, "Efficient Anytime Density-based Clustering," in *SDM*, 2013, pp. 112–120.
- [34] S. T. Mai, X. He, J. Feng, C. Plant, and C. Böhm, "Anytime density-based clustering of complex data," *Knowl. Inf. Syst.*, vol. 45, no. 2, pp. 319–355, 2015.
- [35] M. Dash, H. Liu, and X. Xu, " $1 + 1 > 2$ : Merging Distance and Density Based Clustering," in *DASFAA*, 2001, pp. 32–39.
- [36] B. Borah and D. K. Bhattacharyya, "An Improved Sampling-Based DBSCAN for Large Spatial Databases," in *ICISIP*, 2004, pp. 92–96.
- [37] X. Wang and H. J. Hamilton, "DBRS: A Density-Based Spatial Clustering Method with Random Sampling," in *PAKDD*, 2003, pp. 563–575.
- [38] R. J. G. B. Campello, D. Moulavi, and J. Sander, "Density-Based Clustering Based on Hierarchical Density Estimates," in *PAKDD*, 2013, pp. 160–172.
- [39] R. J. G. B. Campello, D. Moulavi, A. Zimek, and J. Sander, "Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection," *ACM Trans. Knowl. Discov. Data (TKDD)*, vol. 10, no. 1, pp. 5:1–5:51, 2015.
- [40] S. Brecheisen, H. Kriegel, and M. Pfeifle, "Parallel Density-Based Clustering of Complex Objects," in *PAKDD*, 2006, pp. 179–188.
- [41] A. Lulli, M. Dell'Amico, P. Michiardi, and L. Ricci, "NG-DBSCAN: Scalable Density-Based Clustering for Arbitrary Data," *PVLDB*, vol. 10, no. 3, pp. 157–168, 2016.
- [42] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey, "Pardicle: Parallel Approximate Density-Based Clustering," in *SC*, 2014, pp. 560–571.
- [43] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther, "Density-based Clustering using Graphics Processors," in *CIKM*, 2009, pp. 661–670.
- [44] X. Xu, M. Ester, H.-P. Kriegel, and J. Sander, "A Distribution-based Clustering Algorithm for Mining in Large Spatial Databases," in *ICDE*, 1998, pp. 324–331.
- [45] 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*, 2011, pp. 473–480.
- [46] B.-R. Dai and I.-C. Lin, "Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition," in *CLOUD*, 2012, pp. 59–66.
- [47] 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*, 2013, p. 84.
- [48] 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*, 2013, pp. 369–378.
- [49] X. Hu, J. Huang, and M. Qiu, "A Communication Efficient Parallel DBSCAN Algorithm based on Parameter Server," in *CIKM*, 2017, pp. 2107–2110.
- [50] H. Song and J. Lee, "RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning," in *SIGMOD*, 2018, pp. 1173–1187.
- [51] E. Januzaj, H.-P. Kriegel, and M. Pfeifle, "Scalable Density-Based Distributed Clustering," in *PKDD*, 2004, pp. 231–244.
- [52] D. Han, A. Agrawal, W. Liao, and A. N. Choudhary, "Parallel DBSCAN algorithm using a data partitioning strategy with spark implementation," in *Big Data*, 2018, pp. 305–312.



**Son T. Mai** is an assistant professor at Queen's University Belfast, UK. Before, he was at University of Grenoble Alpes in France, Aarhus University in Denmark and Ludwig-Maximilians-University of Munich (LMU) in Germany. His research focuses on Machine Learning and Data Mining algorithms for large complex data.



**Jon Jacobsen** received the BSc and MSc degree in computer science in 2014 and 2017 respectively from Aarhus University, Denmark. He is currently working as a software developer in Getinge Cetrea's R&D.



**Sihem Amer-Yahia** is a CNRS Research Director. Her interests are at the intersection of large-scale data management and social data exploration. Sihem held positions at QCRI, Yahoo! Research and AT&T Labs. She served on the SIGMOD Executive Board, the VLDB Endowment, and the EDBT Board. She is Editor-in-Chief of the VLDB Journal. Sihem serves as co-chair of PVLDB 2018, WWW 2018 Tutorials and ICDE 2019 Tutorials.



**Ivor Spence** has 30 years experience as an academic member of staff and 5 years industrial experience as a software engineer. His research experience includes: satisfiability solvers and benchmarks; heterogeneous computing; GPU programming and virtualization; and cloud computing. He currently leads the Artificial Intelligence Research Theme within the School of Electronics, Electrical Engineering and Computer Science at Queen's University Belfast.



**Nhat-Phuong Tran** received his B.S. in Software Engineering from Science University, Vietnam, M.S. and Ph.D. degrees in Computer Engineering from Myongji University, Korea. His research interests are performance optimization, parallel algorithms and HPC applications, with special interest in GPU computing. He is working at Nvidia in Belfast, United Kingdom.



**Ira Assent** is full professor of computer science at Aarhus University, Denmark, where she heads the Data-Intensive Systems research group and the Big Data Analysis research at the DIGIT Aarhus University Centre. She holds a PhD from RWTH Aachen University, Germany (2008). Her research interests are in data mining, machine learning and data management.



**Quoc Viet Hung Nguyen** is a senior lecturer and an ARC DECRA Fellow (Australia Discovery Early Career Researcher Award) in Griffith University. He earned his Master and PhD degrees from EPFL (Switzerland). His research focuses on Data Integration, Data Quality, Information Retrieval, Trust Management, Recommender Systems, Machine Learning and Big Data Visualization, with special emphasis on web data, social data and sensor data.