DISC: Density-Based Incremental Clustering by Striding over Streaming Data

Bogyeong Kim Kyoseung Koo Juhun Kim Bongki Moon
Department of Computer Science and Engineering
Seoul National University, Seoul, Korea
{bgkim, koo, johnjhkim}@dbs.snu.ac.kr, bkmoon@snu.ac.kr

***Abstract*—Given the prevalence of mobile and IoT devices, continuous clustering against streaming data has become an essential tool of increasing importance for data analytics. Among many clustering approaches, the density-based clustering has garnered much attention due to its unique advantages. The main drawback is, however, the limited scalability attributed to its relatively high computational cost, which is further aggravated when it has to update clusters continuously along with evolving data. In this paper, we present a new incremental density-based clustering algorithm called *DISC* optimized for the sliding window model. *DISC* is capable of producing exactly the same clustering results as existing methods such as Incremental DBSCAN for streaming data much more quickly and efficiently.**

I. Introduction

Clustering is one of the common methods of unsupervised learning, which discovers the natural groupings in the unlabeled data. Since K-means [1] was proposed more than a half century ago, clustering has been studied extensively (publishing thousands of clustering algorithms in the literature [2]) and applied widely to many data analysis tasks in various fields. Recently, the interest in clustering has been revived particularly for data streams [3]–[8]. Given the prevalence of IoT devices, for example, continuous clustering against streaming data has become an essential tool of increasing importance for data analytics such as traffic monitoring [9], [10], community tracking over social networks [11], and outlier detection in network communication [12].

Among many clustering approaches, the density-based clustering, pioneered by Ester *et al.* [13], has garnered much attention due to some of its unique advantages. Unlike K-means (and another well known BIRCH algorithm [14] as well for that matter) that discovers spherical clusters, the density-based approach can identify clusters of an arbitrary shape without requiring the pre-set number of labels, and can determine which cluster each non-noise data object belongs to.

The hardly unnoticeable drawback of the density-based clustering, however, is the limited scalability attributed to its relatively high computational cost. While clustering is already a challenging problem, the scalability concern becomes further aggravated when the clusters need to be updated continuously along with an evolving input data set. The main reason is that updating the clusters even for a single data object inserted to or deleted from the data set may require exploring a large number of its surrounding objects.

Since the incremental version of the DBSCAN algorithm was proposed [15], many studies have been conducted to address the problem of updating clusters efficiently. Most of them attempt to address it by taking approximation or summarization one way or another [4], [6], [16]–[18]. These clustering methods lower the computational complexity of density-based clustering and hence may be well poised to deal with frequent updates. However, they may not achieve the desired level of resolution, for example, to avoid falsely detecting the congested roads among those located in close proximity. They could adopt a minuscule distance threshold for high resolution but, as is demonstrated by Schubert *et al.* [19], that could actually make them run much more slowly, which will be confirmed once again in this paper.

The goal of this work is to address the limitation of density-based clustering squarely so that the clustering tasks for streaming data can be carried out in a timely manner without compromising the quality of clustering results or consuming an excessive amount of computational resources. Achieving this goal is not an easy challenge to tackle, which we believe has been confirmed repeatedly by the previous studies [9], [15], [17], [20]. Therefore, we focus our effort narrowly on developing a new incremental clustering method optimized under the *sliding window* model. We adopt the sliding window model as a general framework for processing streaming data because it is an effective tool for capturing the recent state of streaming data, which cannot be stored in its entirety by virtue of their sheer volume.

The density-based clustering algorithm we present in this paper is called *Density-based Incremental Striding Cluster* (*DISC* in short). It is capable of producing exactly the same clustering results as existing methods such as Incremental DBSCAN much more quickly and efficiently. The contributions of this work are summarized as follows.

- The elaborate design of *DISC*, based on the novel ideas for the *minimal bonding cores* of *ex-cores* and *neo-cores*, enables it to avoid a considerable amount of redundant work by checking the density-connectedness only for the minimal bonding cores.
- The *MS-BFS* strategy and the epoch-based R-tree index probing method are proposed to further reduce the cost

This work was partly supported by the National Research Foundation of Korea (2020R1A2C1010358 and 2016M3C4A7952633). The authors assume all responsibility for the content of the paper.

of checking density-connectedness.

- Through an extensive evaluation carried out under various configurations, we have demonstrated that $\mathcal{DISC}$ is highly effective especially when clusters need to be updated frequently with a small stride. In most practical settings, $\mathcal{DISC}$ outperformed all the *exact* clustering methods in comparison.
- For detecting clusters of high resolution, $\mathcal{DISC}$ outperformed significantly all the *approximate* clustering methods in comparison in both speed and quality.

This paper is organized as follows. Section II briefly covers the background of the density-based clustering and the sliding window model. Section III presents the *minimal bonding cores* and the detailed description of the $\mathcal{DISC}$ algorithm. Further optimizations for checking density-reachability are presented subsequently in Section IV. Section V describes the procedure for updating labels. Finally, we evaluate the $\mathcal{DISC}$ algorithm with real datasets in Section VI, and summarize the contributions of this paper in Section VIII.

II. Background

This section provides readers with the background information of density-based clustering methods and the key characteristics of sliding window models commonly adopted for streaming data processing. The $\mathcal{DISC}$ algorithm proposed in this paper leverages the sliding window model innovatively and overcomes the critical weaknesses of the existing density-based clustering methods.

A. Density-Based Clustering

The density-based clustering was pioneered by Ester *et al.* more than two decades ago [13]. In this seminal work, the density of a point is defined by the number of neighbors that are within a given distance threshold denoted by a parameter ϵ. It is the density that determines the status of a point as one of *core*, *border*, and *noise*. (See Figure 1 for illustration.) Such classification of points is done by introducing another parameter called *MinPts*. If a point has its density no less than *MinPts*, it is labeled as *core*. If a point has its density less than *MinPts* but is within the threshold distance ϵ from at least one other *core* point, it is labeled as *border*. Otherwise, the point is labeled as *noise*. For example, in Figure 1, points X, Y, Z are a core point, a border point and a noise, respectively, assuming the density threshold is four.

Ester *et al.* 's DBSCAN algorithm defines a cluster as a set of core and border points that are *density-reachable* from an arbitrary core point of the cluster. Let $N_\epsilon(p)$ denote a set of points within the threshold distance ϵ from p. A point q is said to be *directly density-reachable* from p if $q \in N_\epsilon(p)$ and p is a core. Note that q does not have to be a core point. The direct density-reachability is a symmetric relation for core points, although it is not when a border point is involved. In general, a point q is said to be *density-reachable* from p if there is a chain of directly density-reachable cores from p to q. In Figure 1, border Y is directly density-reachable from core X, but not vice versa. Cores A and B are density-reachable from each other, whereas cores A and X are not. The density-reachability is a transitive but asymmetric relation.

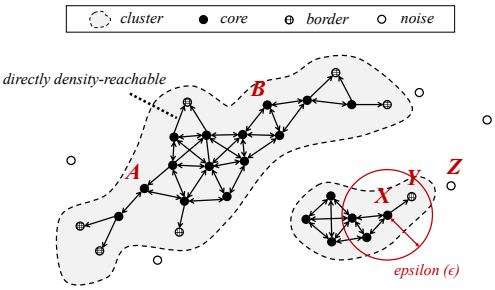


Fig. 1: Density-based clustering

Based on the density-reachability relation of cores and borders, the DBSCAN algorithm attempts to find density-based clusters. Specifically, for each core point p found in the seeding phase, a singleton cluster, say $\mathcal{C}$, containing p is created. Then, in the growing phase, all the directly density-reachable points from any $q \in \mathcal{C}$ are added to $\mathcal{C}$. This process is repeated until $\mathcal{C}$ does not grow any more. Therefore, when it terminates, the DBSCAN algorithm returns density-based clusters, each of which is a *maximally connected component* of core points and border points.

B. Sliding Windows

The fundamental premise of computations over data streams is that the streaming data cannot be stored and processed in its entirety by virtue of their sheer volume. One of the popular models for streaming data analytics is the *sliding window* model, which is typically characterized by two parameters known as $window$ and $stride$ [21]–[25]. The size of the $window$ defines the range of streaming data to be analyzed, and the $stride$ defines the interval at which the result of the analysis is updated.

In this model, the one end of the window is assumed to be anchored to the current time or the current data item. This model thus allows us to analyze and understand the most recent data within the current window. Whenever the sliding window advances, some of the existing older data objects (in the oldest stride) will leave the window while newer data objects (in the new stride) will enter it.

From the computational point of view, it is important to understand that a multitude of data objects enter or leave the window at once when the window advances and there is no particular order of processing among the data points in the same stride. Furthermore, unlike a *decaying data* model [26] where the influence of each data object wanes over time, all the data objects in the current window are assumed to carry the same influence or weight.

Note that the sliding window model can be either *time-based* or *count-based* depending on how the two parameters, $window$ and $stride$, are interpreted. While the parameters are measured in time duration under the former model, they are measured in the number of data objects under the latter. The

Symbol	Description
ϵ	distance threshold
τ	density threshold
W_{curr}	points in the current window
W_{prev}	points in the previous window
Δ_{in}	points entering the window ($W_{curr} - W_{prev}$)
Δ_{out}	points exiting the window ($W_{prev} - W_{curr}$)
$N_\epsilon(p)$	points within ϵ distance from p
$n_\epsilon(p)$	cardinality of $N_\epsilon(p)$
$l(p)$	category label of p
$p \overset{\sim}{\rightsquigarrow} q$	q is *retro-reachable* from p
$p \overset{+}{\rightsquigarrow} q$	q is *nascent-reachable* from p
$\mathcal{M}^-$	minimal bonding cores for *ex-cores*
$\mathcal{M}^+$	minimal bonding cores for *neo-cores*

TABLE I: Notations

clustering algorithm proposed in this paper is not subject to how those parameters are measured and will work with either of the two model types.

III. The DISC Algorithm

This section presents a new incremental clustering algorithm, DISC, we propose to deal with the problem of clustering large-scale streaming data under the sliding window model. We first give an overview of the algorithm and then describe its two primary steps, COLLECT and CLUSTER.

A. Overview of DISC

DISC is no different from DBSCAN in that it assigns each individual data point to one of the three categories, *core*, *border*, and *noise*. Besides, by the time clustering is completed, a cluster id (or `cid` in short) will have been assigned to every data point except for those in the *noise* category.

Let $N_\epsilon(p)$ denote a set of data points within the threshold distance ϵ from p. The cardinality of $N_\epsilon(p)$, denoted by $n_\epsilon(p)$, is maintained up-to-date for each data point p. Whenever the sliding window advances by a stride, new data points may enter the window while some of the existing ones may leave it. DISC will then take the changes in the data population within the sliding window into account and will bring the clusters up to date by updating the $n_\epsilon(p)$ value and the category label, denoted by $l(p)$, of each point p in the current window. (Refer to Table I for more symbols adopted to describe DISC in this paper.) Apparently, recomputing $n_\epsilon(p)$ and $l(p)$ values for every point p in the current window is the primary task for DISC to update clusters. This will be carried out in two separate steps called COLLECT and CLUSTER, which are summarized in Figure 2.

The COLLECT step updates $n_\epsilon(p)$ for every point p in the current window, and resets or initializes $l(q)$ of every point q leaving or entering the sliding window. It then identifies *ex-cores* and *neo-cores* among the points that remain in the current window, and updates a spatial R-tree index accordingly for the changes. The notions of *ex-cores* and *neo-cores* are the cornerstone of DISC and will be defined in Section III-B.

The CLUSTER step finds the *minimal bonding cores* of each *ex-core* and each *neo-core*, determines the types of cluster evolution by checking reachability, and finally recomputes the cluster labels for every point in the current window.

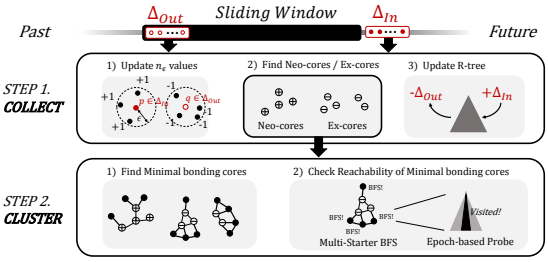


Fig. 2: Overview of DISC

The *minimal bonding cores* are the key idea that enables DISC to update clusters efficiently. They will be defined in Section III-C.

B. COLLECT

When a point p enters or exits the sliding window, it changes the number of ϵ-*neighbors* for all the ϵ-*neighbors* of p. In other words, for any point $q \in N_\epsilon(p)$ in the current window, the $n_\epsilon(q)$ value needs to be updated. Let Δ_{out} and Δ_{in} denote the set of points exiting the window and the set of points entering the window, respectively. Then, for any point $q \in N_\epsilon(p)$, $n_\epsilon(q)$ will decrease if $p \in \Delta_{out}$ (Line 6 of Algorithm 1), and it will increase if $p \in \Delta_{in}$ (Line 12). At the end of the COLLECT step, every data point in the current window will have an up-to-date n_ϵ value.

Algorithm 1: COLLECT (Δ_{in}, Δ_{out})

```
1  C_out ← ∅;                                   // C_out : ex-cores in Δ_out
2  foreach p ∈ Δ_out do
3      if l(p) = core then C_out ← C_out ∪ {p}
4      else delete p from the R-tree index
5      foreach q ∈ N_ε(p) do
6          if l(q) ≠ deleted then n_ε(q)--
7      l(p) ← deleted, n_ε(p) ← 0
8  foreach p ∈ Δ_in do
9      Insert p into the R-tree index
10     l(p) ← unclassified, n_ε(p) ← 1
11     foreach q ∈ N_ε(p) do
12         if l(q) ≠ deleted then n_ε(q)++, n_ε(p)++
13 Compute the sets {ex-cores} and {neo-cores}
14 return ({ex-cores}, {neo-cores}, C_out)
```

Another major work to be done in this step is to identify a set of *ex-cores* and a set of *neo-cores* defined below. Let W_{curr} denote the set of points in the current window, and let W_{prev} denote the set of points in the previous window.

***Definition** 1:* **(Ex-core)** A data point p that was a core in the previous window is called an *ex-core* if it already exited the current window (*i.e.*, $p \in \Delta_{out}$) or it is still in the current window but no longer a core (*i.e.*, $p \in W_{prev} \cap W_{curr}$). $\lhd$

***Definition** 2:* **(Neo-core)** A data point that is a core in the current window is called a *neo-core* if it just entered the current

window (*i.e.*, $p \in \Delta_{in}$) or it was not a core in the previous window (*i.e.*, $p \in W_{prev} \cap W_{curr}$). ◁

In the next CLUSTER step, these two mutually exclusive sets of *ex-cores* and *neo-cores* will play a critical role in determining the types of cluster evolution such as *split* and *merger* among others.

Note that the COLLECT algorithm uses an R-tree index to facilitate the retrieval of ϵ-$neighbors$ of a given point. Obviously, whenever the sliding window advances, it has to maintain the R-tree index up to date by adding and removing entries as data points enter and leave the window. However, the *ex-cores* in Δ_{out} will not be removed from the R-tree index until both the COLLECT and CLUSTER steps are completed. This is because the CLUSTER step will have to access *ex-cores* in Δ_{out} as well as those in $W_{prev} \cap W_{curr}$. For the reason, all the *ex-cores* that exited the window are collected (Line 3 of Algorithm 1) and passed to the CLUSTER step in a set denoted by C_{out}. Note that the set C_{out} is equivalent to $\{ex\text{-}cores\} \cap \Delta_{out}$.

Algorithm 2: CLUSTER (*ex-cores*, *neo-cores*, C_{out})

```
   // {ex-cores}, {neo-cores}, C_out from COLLECT
1  E ← {ex-cores}
2  while E ≠ ∅ do
3      Compute R⁻(p) and M⁻(p) for p ∈ E
4      ncc ← MS-BFS(M⁻(p))
         // ncc: # of connected components in M⁻(p)
5      if ncc > 1 then a cluster splits
6      else a cluster shrinks or dissipates
7      E ← E − R⁻(p)          // Avoid redundant work
8  Remove C_out from the R-tree index
9  N ← {neo-cores}
10 foreach p ∈ N do
11     if M⁺(p) is disconnected then clusters merge
12     else a cluster grows or emerges
13     N ← N − R⁺(p)          // Avoid redundant work
```

C. CLUSTER

The *ex-cores* and *neo-cores* defined in the previous section determine collectively whether a cluster should be split and whether clusters should be merged. Besides, the other types of cluster evolution such as *emergence*, *dissipation*, *expansion*, and *shrink* can also be determined solely by the *ex-cores* and *neo-cores*.

The CLUSTER step presented in this section provides a sophisticated but highly efficient procedure to expedite the processing of cluster evolution. The high-level description of the procedure is given in Algorithm 2. As can be seen in the pseudocode, *ex-cores* are used to process splitting clusters while *neo-cores* are used to process merging clusters. Between these two main operations, splitting a cluster is computationally much more intensive for updating clusters incrementally. Each of the sub-procedures of the algorithm will be described in detail in this section.

Splitting a Cluster

A cluster split involves a breakup of density-reachability between core points. When a core point loses its status to become an *ex-core*, it may cut a density-reachable path between the cores in the same cluster, which in turn may contribute to a cluster split event. Essentially, a cluster can only be split when *ex-cores* break up a density-reachable path between two core points in the same cluster and there is no more path left between them.

In fact, we can expedite splitting a cluster by consolidating all the *ex-cores* turned up when the sliding window advances. We can avoid a considerable amount of redundant work by checking the density connectedness only for the *minimal bonding cores* (that will be defined below) and by minimizing the number of range searches required.

In an attempt to clearly specify the minimal set of cores to examine, we define below the notions of *retro-reachability* (Definition 3) and *minimal bonding cores* of an *ex-core* (Definition 4).

***Definition** 3:* For a pair of *ex-cores* p and q, p is *directly retro-reachable* to q if p was directly density-reachable to q with respect to the previous window W_{prev}. More generally, p is *retro-reachable* to q (denoted by $p \dashrightarrow q$) if there is a chain of *directly retro-reachable ex-cores* from p to q. ◁

Unlike density-reachability, the retro-reachability is transitive and symmetric because this relation is defined for *ex-cores* only. That is, $p \dashrightarrow q$ is equivalent to $q \dashrightarrow p$. For an *ex-core* p, let $\mathcal{R}^-(p)$ denote a set of *ex-cores* that are retro-reachable from p. Formally,

$$\mathcal{R}^-(p) = \{q \in W_{prev} \mid p \dashrightarrow q\}.$$

Note that $p \in \mathcal{R}^-(p)$ and retro-reachability is reflexive.

Now we define the *minimal bonding cores* of an *ex-core*, which will be used to pose a necessary condition for the *ex-core* to trigger splitting a cluster. Note that not every *ex-core* necessarily splits a cluster.

***Definition** 4:* For an *ex-core* p, the set $\mathcal{M}^-(p)$ of its *minimal bonding cores* is defined to be

$$\mathcal{M}^-(p) = \{q \mid (q \text{ is a core in both } W_{prev} \text{ and } W_{curr}) \wedge (q \in N_\epsilon(r) \text{ for some } r \in \mathcal{R}^-(p))\}$$ ◁

The second half of the condition, namely "$q \in N_\epsilon(r)$ for some $r \in \mathcal{R}^-(p)$" requires that q must be among the ϵ-$neighbors$ of a certain *ex-core* that is retro-reachable from p. It is this condition which the minimality of $\mathcal{M}^-(p)$ comes from. Among the cores that are density-reachable to the *ex-cores* in $\mathcal{R}^-(p)$, only those *directly* density-reachable to some *ex-core* are included in $\mathcal{M}^-(p)$.

For example, in Figure 3, border P_1 and core P_2 are about to exit the current window. Exiting P_1 turns its adjacent cores, B and K, to *ex-cores*. Exiting P_2 also turns its adjacent cores, D and F, as well as itself to *ex-cores*. The set of *ex-cores* that are retro-reachable from B is $\mathcal{R}^-(B) = \{B, D, F, K, P_2\}$, and the *minimal bonding cores* of B is $\mathcal{M}^-(B) = \{A, C, E, G, H, J\}$.

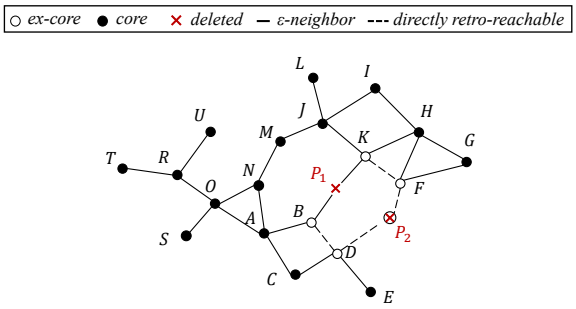


Fig. 3: Cluster evolution by sliding window

Combined with the minimality of $\mathcal{M}^-(p)$, the following lemmas and theorem allow us to focus on the minimal set of core points when determining whether any cluster would be split by an *ex-core* p or any of its retro-reachable *ex-cores*.

***Lemma** 1:* For *ex-cores* p and q, $\mathcal{M}^-(p) = \mathcal{M}^-(q)$ if $\mathcal{R}^-(p) = \mathcal{R}^-(q)$.

Proof. For $\forall x \in \mathcal{M}^-(p)$, x was and is a core in W_{prev} and W_{curr} such that $x \in N_\epsilon(r)$ for some $r \in \mathcal{R}^-(p)$. If $\mathcal{R}^-(p) = \mathcal{R}^-(q)$, then it holds that $x \in \mathcal{M}^-(q)$. Therefore, $\mathcal{M}^-(p) \subseteq \mathcal{M}^-(q)$. It can be shown that $\mathcal{M}^-(q) \subseteq \mathcal{M}^-(p)$ in the same way. □

***Lemma** 2:* An *ex-core* p does not split the cluster it belongs to if $\mathcal{M}^-(p)$ is *density-connected*.

Proof. (By contradiction.) Suppose a cluster $\mathcal{C}$ containing p in the previous window is being split to two non-empty separate clusters $\mathcal{C}_1$ and $\mathcal{C}_2$. Any point $x \in \mathcal{C}_1$ was density-reachable to p in the previous window because both x and p were in $\mathcal{C}$. So there must exist $x' \in \mathcal{C}_1$ that was on the density-reachable path and closest to p. This implies that x' is an ϵ-neighbor of p or one of $\mathcal{R}^-(p)$. Thus, by definition, $x' \in \mathcal{M}^-(p)$. Similarly, there must exist $y' \in \mathcal{C}_2$ such that $y' \in \mathcal{M}^-(p)$. Since $\mathcal{M}^-(p)$ is density-connected, x' and y' are density-reachable from each other. This is a contraction to the assumption that $\mathcal{C}_1$ and $\mathcal{C}_2$ are two separate clusters. □

***Theorem** 1:* For an *ex-core* p, if $\mathcal{M}^-(p)$ is *density-connected*, none of the *ex-cores* in $\mathcal{R}^-(p)$ splits a cluster.

Proof. For any *ex-core* $x \in \mathcal{R}^-(p)$, $\mathcal{R}^-(x) = \mathcal{R}^-(p)$ because the retro-reachability is symmetric and transitive. Then, it follows that $\mathcal{M}^-(x) = \mathcal{M}^-(p)$ by Lemma 1. Therefore, by Lemma 2, x does not split the cluster it belonged to in the previous window. □

The implication of Theorem 1 is that examining any one of the *ex-cores* in $\mathcal{R}^-(p)$ will obviate the need for all the other *ex-cores* in $\mathcal{R}^-(p)$ (Line 7 of Algorithm 2). This will let us avoid redundant work and reduce the number of range searches significantly.

Now let us turn our attention to cluster evolution caused by *ex-cores*. For each *ex-core* p, all *ex-cores* in $\mathcal{R}^-(p)$ can be discovered by executing $|\mathcal{R}^-(p)|$ range searches. Since all the cores in $\mathcal{M}^-(p)$ are ϵ-*neighbors* of an *ex-core* in $\mathcal{R}^-(p)$, they will also be discovered with no additional search. Once the set $\mathcal{M}^-(p)$ of minimal bonding cores is computed for each *ex-core* p, we are ready to determine the types of cluster evolution caused by them. If $\mathcal{M}^-(p)$ is not density-connected (*i.e.*, there is more than one connected component), then the cluster from the previous window will be split in the current window (Line 5 of Algorithm 2). If $\mathcal{M}^-(p)$ is density-connected, the cluster will be simply $shrunk$ in size (Line 6). If $\mathcal{M}^-(p)$ is empty, then the cluster will be $dissipated$ completely.

Checking the connectedness of $\mathcal{M}^-(p)$ can be done by a BFS traversal, which requires executing a number of range searches against the R-tree index. When $\mathcal{M}^-(p)$ is large, this overhead may become significant and warrant careful coordination and optimization. In order to do it efficiently, we propose a variant of breadth-first search called *Multi-Starter BFS* (invoked in Line 4 of Algorithm 2) and an *Epoch-Based* probing method for the R-tree index. The number of range searches to execute can be considerably reduced by the former, and the individual range searches can be performed more quickly by the latter. They will be described in detail in Section IV.

The following examples illustrate how DBSCAN and the proposed $\mathcal{DISC}$ algorithm deal with cluster evolution caused by *ex-cores* and compare these methods with respect to the minimum number of range searches required by each method.

***Example** 1:* Consider the evolving cluster shown in Figure 3. DBSCAN performs the clustering procedure from scratch. Specifically, when each of P_1 and P_2 is excluded from the window, a BFS traversal is performed for every point in the window. At least 19 range searches are required, and the number of range searches would be higher if $noise$ and $border$ points were taken into account. ◊

***Example** 2:* Consider again the same scenario given in Example 1. After the exclusion of P_1 and P_2, there are five *ex-cores*, B, D, F, K, and P_2, which turned up in the current window. The `CLUSTER` algorithm of $\mathcal{DISC}$ finds $\mathcal{R}^-(p)$ and $\mathcal{M}^-(p)$ for each $p \in \{B, D, F, K, P_2\}$ by executing five range searches. Then, a BFS traversal is performed for each of the five minimal bonding core sets. Although more work appears to be required with an increased number of BFS traversals, the opposite is true. This is because all the five *ex-cores* are retro-reachable from one another, and hence

$$\mathcal{R}^-(B) = \mathcal{R}^-(D) = \mathcal{R}^-(F) = \mathcal{R}^-(K) = \mathcal{R}^-(P_2).$$

Thus, once $\mathcal{M}^-(B)$ is processed and all the *ex-cores* in $\mathcal{R}^-(B)$ are excluded from further consideration (by Line 7 of Algorithm 2 that will make the set E empty in this case), there will be no more *ex-cores* left to process the minimal bonding core sets for. A BFS traversal will only be performed for $\mathcal{M}^-(B) = \{A, C, E, G, H, J\}$ by executing no more than six range searches. Therefore, the minimum number of range searches is reduced further down to eleven. ◊

Merging Clusters

After all *ex-cores* are processed, the `CLUSTER` algorithm starts examining *neo-cores* to see whether existing clusters

would have to be merged (Lines 9-13 of Algorithm 2). Clusters are merged when cores from different clusters become density-reachable. Since only a *neo-core* can contribute to creating a new density-reachable path, existing clusters can be merged only when an existing point gains the core status or a new core enters the current window.

Much the similar way done for *ex-cores*, below we define the *nascent-reachability* (Definition 5) and the *minimal bonding cores* of a *neo-core* (Definition 6).

***Definition** 5:* For a pair of *neo-cores* p and q, p is *directly nascent-reachable* to q if p is directly density-reachable to q with respect to the current window W_{curr}. In general, p is *nascent-reachable* to q (denoted by $p \not\rightarrow q$) if there is a chain of *directly nascent-reachable neo-cores* from p to q. $\triangleleft$

For a *neo-core* p, let $\mathcal{R}^+(p)$ denote a set of *neo-cores* that are nascent-reachable from p. Formally,

$$\mathcal{R}^+(p) = \{q \in W_{curr} \mid p \not\rightarrow q\}.$$

Like the retro-reachability, the nascent-reachability is reflexive, symmetric and transitive. Therefore, $p \not\rightarrow q$ is equivalent to $q \not\rightarrow p$, and $p \in \mathcal{R}^+(p)$.

***Definition** 6:* For a *neo-core* p, the set $\mathcal{M}^+(p)$ of its *minimal bonding cores* is defined to be

$$\mathcal{M}^+(p) = \{q \mid (q \textit{ is a core in both } W_{prev} \textit{ and } W_{curr}) \wedge (q \in N_\epsilon(r) \text{ for some } r \in \mathcal{R}^+(p))\}$$ $\triangleleft$

The minimal bonding cores of *neo-cores* defined above enable us to determine the types of cluster evolution caused by them. For a *neo-core* p, if $\mathcal{M}^+(p)$ is empty, then a new cluster $emerges$, which consists solely of the *neo-cores* in $\mathcal{R}^+(p)$. If all the cores in $\mathcal{M}^+(p)$ belong to one cluster, then all the *neo-cores* in $\mathcal{R}^+(p)$ are added to that cluster and let it grow in size (*expansion*). If the cores in $\mathcal{M}^+(p)$ are spread over more one cluster, say $\mathcal{C}_1, \ldots, \mathcal{C}_k$, then all the cores in $\mathcal{C}_1, \ldots, \mathcal{C}_k$ are merged into a single cluster together with the *neo-cores* in $\mathcal{R}^+(p)$.

Despite all the similarities between $\mathcal{M}^-(q)$ of an *ex-core* q and $\mathcal{M}^+(p)$ of a *neo-core* p, there is a striking difference between them. While the connectedness of the cores in $\mathcal{M}^-(q)$ must be checked for each *ex-core* q by executing a number of range searches (Line 4 of Algorithm 2), it is not necessary to do that for the cores in $\mathcal{M}^+(p)$ of any *neo-core* p. We have only to find out whether $\mathcal{M}^+(p)$ is empty or how many clusters the cores in $\mathcal{M}^+(p)$ are spread over (Line 11), which can be done quickly just by examining the labels of the cores. Therefore, the cluster evolution caused by *neo-cores* will be handled with much more ease than the cluster evolution caused by *ex-cores*.

IV. Checking Reachability

Whether a cluster is split by an *ex-core* is determined by the density-reachability among the minimal bonding cores of the *ex-core*. For a given pair of cores, the density-reachability can be checked by executing a series of range searches against the R-tree index starting from either core. Only when the search encounters the other core before exhausting all reachable cores, the pair will be declared density-reachable. This procedure is essentially a variant of breadth-first search (BFS) commonly used for graph traversal. Considering the potentially high cost of reachability checks requiring a number of range searches, we propose *Multi-Starter BFS* and *Epoch-Based* probing strategy for the R-tree index.

Note that range searches against the R-tree index could be avoided entirely if the *ϵ-neighbor* relations between cores were materialized in a graph. Then the reachability checks could be done more quickly by traversing the materialized graph. However, we choose not to do that because the $\mathcal{O}(n^2)$ cost of maintaining a materialized graph can be too high with n being the number of cores in the graph.

A. Multi-Starter BFS

In order to check the density-connectedness of $\mathcal{M}^-(p)$ for an *ex-core* p efficiently, we have developed a new search procedure called *Multi-Starter Breadth-First Search (MS-BFS)*. This is an extension of the traditional breadth-first search.

Algorithm 3: *MS-BFS* ($\mathcal{M}^-(p)$)

```
1  ncc ← 0                          // # of connected components
2  M ← M⁻(p)
3  Q_{s∈M} ← EmptyQueue
4  foreach s ∈ M do Q_s.enque(s)
5  while |M| > 1 do
       // Run BFS_s for each s ∈ M simultaneously
6      if Q_s is empty then ncc++, M ← M − {s}
7      else
8          r ← Q_s.deque
9          foreach core x ∈ N_ε(r) unvisited by BFS_s do
10             if x is visited by BFS_t then
11                 Q_s ← Q_s ∪ Q_t, M ← M − {t}
12             else Q_s.enque(x)
13 return ncc
```

Imagine a (non-materialized) graph $G(V, E)$ whose vertex set V consists of core points and whose edge set E consists of pairs of cores that are ϵ-$neighbors$ to each other. The *MS-BFS* initiates a breadth-first search starting from each vertex in $\mathcal{M}^-(p)$ of G simultaneously. When two searches meet at a certain vertex, they merge their queues of vertices into one and restart as a single search with the merged queue (Line 11 of Algorithm 3). If all those searches are combined into one, then the graph G is connected, which indicates that all the cores in $\mathcal{M}^-(p)$ are density-connected. Otherwise, the graph G has more than one connected component and $\mathcal{M}^-(p)$ is not density-connected. Specifically, as shown in Line 6, if one of the queues becomes empty before all the vertices in G are visited, that thread of the *MS-BFS* terminates with its own connected component. In this case, the connected component does not cover the entire set of vertices in G. Thus the graph

G is not connected, and neither is the set $\mathcal{M}^-(p)$ of minimal bonding cores. That is, a cluster split happens.

It should be noted that the *MS-BFS* presented in this paper is completely different from Then *et al.* 's Multi-Source BFS [27]. The Multi-Source BFS executes multiple *independent* BFS traversals over the same graph simultaneously and focuses on reducing the memory accesses when every vertex is visited multiple times. On the other hand, our *MS-BFS* aims at reducing the scope of exploration by starting BFSs from multiple starters concurrently thereby reducing the number of range searches made against the R-tree index.

B. Epoch-Based Probing of R-tree Index

In the conventional BFS graph traversal, an array of Boolean flags is used to separate visited vertices from unvisited ones. Such an array of Boolean flags, however, does not help us reduce the cost of checking density-reachability because those flags will be referenced only after the $\epsilon\text{-}neighbors$ of a certain core are identified by a complete range search. Consequently, the cost of avoiding an already visited core (as much as visiting an unvisited one) would remain as high as $\Omega(d)$, where d is the depth of the R-tree index.

An easy fix to this problem is to store the Boolean flags in the R-tree index itself instead of a separate array. If an entry in a leaf node is marked as visited, then the corresponding core will be ignored. If an entry in an internal node is marked as visited, then all the cores indexed in the subtree rooted at the entry will be ignored altogether. Unfortunately, however, this approach introduces another problem. Whenever another density-reachability checking *MS-BFS* is initiated, all the Boolean flags of the R-tree index must be reset beforehand, and this overhead may not be trivial.

Algorithm 4: $Epoch\_Based\_Probe(range, node, tick)$

```
1 foreach entry in node do
2     if range covers entry and entry.epoch < tick then
3         if node is a leaf then entry.epoch ← tick
4         else
              Epoch_Based_Probe(range, entry, tick)
5 node.epoch ← min(entries.epoch)
```

We address this concern by adopting an epoch-based method that stores *epochs of a visiting history* rather than just Boolean visited-or-not flags in the R-tree index. This method can be implemented efficiently with a monotonically increasing counter. When a density-reachability checking *MS-BFS* begins anew, a *tick* value is assigned from the counter so that each individual *MS-BFS* instance is given a distinct tick value.

An entry in a leaf node takes the current *tick* value as its *epoch* when the entry (and its core) is visited (Line 3 of Algorithm 4). Thus, an epoch value smaller than the current tick implies that the core referenced by the leaf entry has not been visited by the current instance of *MS-BFS*. On the backtracking, the range search adjusts the epoch of a parent entry such that it is always equal to the *minimum* of all epochs in its child entries (Lines 5). Thus, the epoch of an internal entry smaller than the current tick implies that there exists at least one child entry that has not been visited by the current instance of *MS-BFS*. The checking procedure can ignore a core or a group of cores altogether if an index entry encountered has an epoch equal to the current tick.

Note that even with this epoch-based method, the cost of finding unvisited $\epsilon\text{-}neighbors$ will remain as $\Omega(d)$. Nonetheless, this method can reduce the cost of probing the R-tree index quite considerably by pruning out any unnecessary portion of the index each range search has to probe.

V. Updating Labels

The ultimate goal of $\mathcal{DISC}$ is to label each core or border point in the current window correctly with the id of the cluster (or *cid*) it belongs to. Since the cluster membership of points may change as the sliding window advances, the `CLUSTER` algorithm of $\mathcal{DISC}$ handles it by updating the labels for *ex-cores*, *neo-cores* and any point which is affected by *ex-cores* and *neo-cores*.

The labels of *ex-cores* may change to $border$ or $noise$, and the labels of $cores$ affected by *ex-cores* may change to a different cid due to the splits caused by *ex-cores*. These labels of *ex-cores* and $cores$ are updated when a $\mathcal{M}^-$ set is processed by the *MS-BFS* procedure. Similarly, the labels of *neo-cores* and $cores$ affected by them are updated so that they have the same cid when a $\mathcal{M}^+$ set is processed. Besides, non-core points near *ex-cores* and *neo-cores* can also change their labels. Labels of these points are instantly updated if they are visited while *minimal bonding cores* are processed. Otherwise, they will be updated later by examining labels of their $\epsilon\text{-}neighbors$. Eventually, all the $core$ and $border$ points of the same connected component will share the same cid, and this guarantees a set of clusters identical to what DBSCAN would produce.

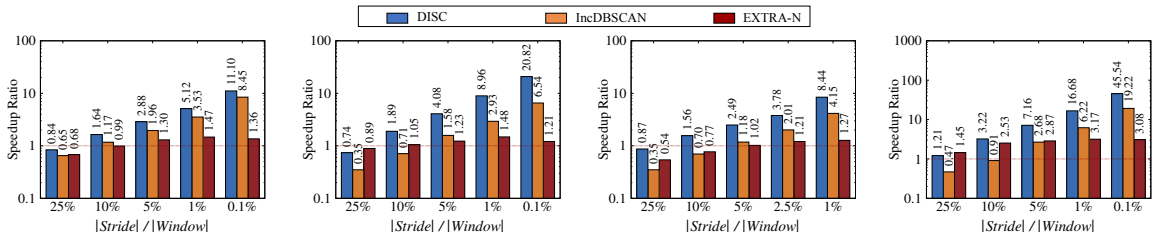
VI. Evaluation

We analyze the performance characteristics of $\mathcal{DISC}$ in comparison with existing density-based clustering methods, DBSCAN [13], IncDBSCAN [15], and EXTRA-N [9]. All of them produce the same clustering results without any approximation. The EXTRA-N method is included because it takes a unique approach based on consolidating multiple sub-windows to avoid costly range searches. In addition, we also compare $\mathcal{DISC}$ with DBSTREAM [8], EDMSTREAM [7] and ρ-double-approximate DBSCAN [17], which produce approximate or summarized clustering results.

A. Experimental Settings

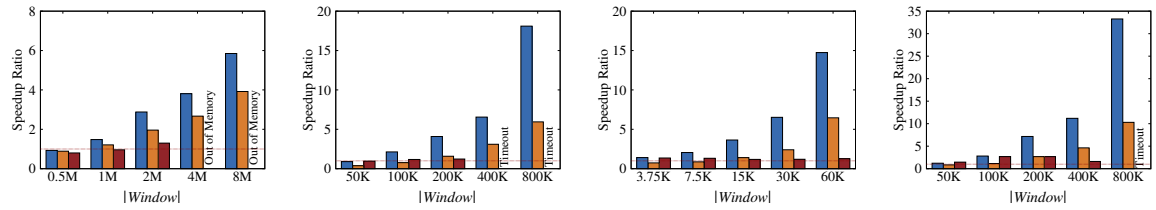
Except for EDMSTREAM[1], we implemented all the aforementioned clustering algorithms as well as an in-memory version of the R-tree index in Java with JDK 1.8.0-121. All the experiments were carried out on an AMD-based stand-alone

[1]The Java code is available in https://github.com/ShufengGong/EDMStream.



(a) DTG (b) GeoLife (c) COVID-19 (d) IRIS

Fig. 4: Relative speedup over DBSCAN with a varying size of stride



(a) DTG ($|Stride| = 100K$) (b) GeoLife ($|Stride| = 10K$) (c) COVID-19 ($|Stride| = 750$) (d) IRIS ($|Stride| = 10K$)

Fig. 5: Relative speedup over DBSCAN with a varying size of window

machine with Ryzen 7 1700 8-Core Processor, 64GB RAM, and a 256GB solid-state drive, running Ubuntu 18.04 LTS.

Throughout the experimental evaluation, we adopted the *count-based* sliding window model where its parameters, *window size* and *stride*, are measured in terms of the number of data points rather than time duration. This model enables us to control the amount of workload and calibrate the experimental settings with more ease. The ingestion order of data points still follows strictly the time stamp of the data records.

Each input dataset was preloaded into memory-resident data structures so that no disk activity would affect the performance of any clustering method. Elapsed times were measured by the `System.nanoTime` function. Each of all the measurements presented in this section is the average of five runs.

B. Real-World Datasets

We used four real-world datasets to evaluate $\mathcal{DISC}$ in comparison with existing density-based clustering algorithms. The datasets used in the experiments are briefly depicted below.

DTG is a dataset obtained from digital tachograph devices attached to commercial vehicles in a metropolitan city [28]. Each record includes the time, location, speed and acceleration of a vehicle. The 2D coordinates (p_{lat}, p_{lon}) were used to denote the spatial location of each record, where p_{lat} and p_{lon} are the latitude and longitude fields, respectively. The total number of records is 300 million.

GeoLife is a set of GPS trajectories obtained from 182 user over a period of four years [29]. Each record includes the time and location of each user. The 3D normalized coordinates $(p_{lat}, p_{lon}, p_{alt}/300,000)$ were used to denote the spatial location, where p_{alt} is the altitude field. The total number of records is 24.8 million.

COVID-19 consists of geo-tagged tweets about the novel *coronavirus* from March to September 2020 [30]. Each record includes the time and location of a tweet around the world. The 2D coordinates (p_{lat}, p_{lon}) were used to denote the spatial location. The total number of records is 210 thousand.

IRIS is a dataset of earthquake events around the world from 1960 to 2019 [31]. The 4D normalized coordinates $(p_{lat}, p_{lon}, p_{dep}/10, p_{mag} \times 10)$ were used to denote the spatial location, where p_{dep} and p_{mag} are the depth and magnitude fields, respectively. The total number of records is 1.8 million.

C. Baseline Evaluation

This section compares the overall performance of $\mathcal{DISC}$ with such exact clustering methods as DBSCAN, IncDBSCAN, and EXTRA-N with respect to elapsed time. The IncDBSCAN code we implemented ran with our *MS-BFS* algorithm in its own favor. Since DBSCAN is a clustering algorithm designed for a static database, we used it as the baseline method rather than a target of direct comparison in our experiments, and measured the performance of the other methods in relation to that of DBSCAN.

Figures 4 and 5 show the relative speedup of $\mathcal{DISC}$, IncDBSCAN and EXTRA-N over DBSCAN for the four real-world datasets, with a varying size of stride and window, respectively. (As for the absolute performance measurements, the average elapsed times taken by DBSCAN were 102s, 523s, 496ms, and 533s for the four datasets, respectively, in Figure 4.) Table II summarizes the threshold values and the default window sizes chosen for each dataset. For the DTG dataset, we adopted the ground traffic monitoring example to set the distance (ϵ) and density (τ) thresholds. The distance threshold was set to be small enough to distinguish roads in close proximity, and the density threshold was set to the average number of points within the distance threshold. For the other datasets, we adopted the parameter settings used by the previous work based on a K-distance graph [13], [19]. The

sliding window sizes were set to a fraction of each dataset, roughly corresponding to a chosen time duration.

Dataset	density (τ)	distance (ϵ)	$\|window\|$
DTG	372	0.002	2M (∼10 min)
$GeoLife$	7	0.01	200K (∼fortnight)
$COVID$-19	5	1.2	15K (∼fortnight)
$IRIS$	9	2	200K (∼decade)

TABLE II: Threshold values and window sizes

For each of the four clustering methods including DBSCAN, we measured the average elapsed time taken to update clusters when the sliding window advanced by a single stride. The execution time of DBSCAN remained unaffected by a varying ratio of stride to window, because it recomputed clusters from scratch whenever the sliding window advanced. In contrast, the execution time of the other three methods was affected significantly by the ratio. In particular, IncDBSCAN and $\mathcal{DISC}$ updated clusters incrementally focusing on the data points leaving and entering the sliding window. Consequently, their execution time tended to decrease as the stride shrank smaller.

In the case of EXTRA-N, however, its speedup over DBSCAN started being saturated earlier as the stride shrank smaller or the window grew larger. As is shown in Figure 5, when the sliding window was large, EXTRA-N exceeded the memory capacity or was terminated forcefully after ten hour execution. This is because EXTRA-N maintains as many sub-windows as the number of strides fitting in a single window so that it can keep track of the local neighbors of individual data points. Thus, when the ratio of stride to window was too high, it suffered from the steep increase of memory consumption, and the cost of maintaining too many sub-windows outweighed the benefit from avoiding range searches.

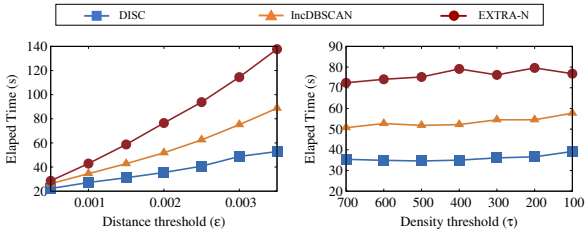
When the stride was no larger than 10 percent of the sliding window, $\mathcal{DISC}$ was the best performer among all the clustering methods compared including DBSCAN. In Figure 4, for example, the speedup of $\mathcal{DISC}$ over the second-best performer ranged from 27% (in IRIS with a 10% stride) to 318% (in GeoLife with a 0.1% stride). There was no clear second-best performer in this range of stride sizes for the datasets.

The most noteworthy feature of $\mathcal{DISC}$ observed from this set of experiments was that its benefit is amplified particularly when applied to *finer-grained* incremental clustering with the sliding window advancing frequently in a small stride. On the other hand, when the stride was as large as 25 percent of the sliding window, all the three incremental methods performed poorly. Their execution times were comparable to that of DBSCAN at best or even worse than that. Given that an efficient method is desired more for finer-grained incremental clustering, it clearly attests that $\mathcal{DISC}$ is an effective tool for the task of clustering fast evolving streaming data.

D. Drilled-Down Evaluation

Having presented the baseline evaluation of $\mathcal{DISC}$, we provide further analyses to understand its performance characteristics in more detail under various parameter settings and the effects of the proposed optimization techniques. Unless stated otherwise, the experiments were carried out under the same default settings as shown in Table II.

1) Effects of threshold values: The density-based clustering is governed by two thresholds, density (τ) and distance (ϵ), as they determine which points are cores. Therefore, these parameters inherently have a critical influence over not only the quality of clustering results but also the cost of cluster discovery.

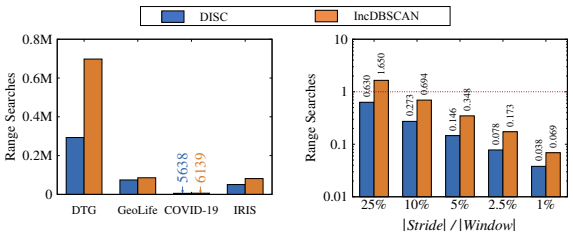


(a) τ fixed to 372 (b) ϵ fixed to 0.002

Fig. 6: Threshold effects : distance (ϵ) and density (τ)

Figures 6(a) and 6(b) show the elapsed times taken by the three incremental clustering methods for the DTG dataset with a varying distance threshold (ϵ) and with a varying density threshold (τ), respectively. The stride size was fixed to 5% of the window size. The elapsed times of all three methods were elongated as the value of ϵ increased or the value of τ decreased. This is because a longer distance threshold allowed data points to have more ϵ-$neighbors$ and a lower density threshold produced a larger population of core points. However, the impact of τ on the elapsed time was not as significant as we anticipated. Note that the performance of $\mathcal{DISC}$ was much more stable and efficient than the others over the entire spectrum of ϵ values and τ values tested. The same trend was observed from the other datasets as well.

2) Range searches: We counted the number of range searches executed by IncDBSCAN and $\mathcal{DISC}$ in order to understand how much the clustering algorithms were affected by the costly search operations. Unlike DBSCAN, as a static approach, that always invokes as many range searches as the number of data points in the current sliding window, the number of range searches required by IncDBSCAN and $\mathcal{DISC}$ is dependent on the stride sizes.



(a) $|Stride|$ / $|Window|$ = 5% (b) DTG with varying ratio

Fig. 7: Range searches executed

Figure 7(a) shows the number of range searches carried out by $\mathcal{DISC}$ and IncDBSCAN with the ratio of stride to window fixed to 5%. $\mathcal{DISC}$ invoked a smaller number of range searches than IncDBSCAN across all the four datasets. Figure 7(b) compares the two methods relatively in comparison with DBSCAN for the DTG dataset. $\mathcal{DISC}$ was superior to IncDBSCAN as well as DBSCAN in the number of range search invocations consistently across all the stride-to-window ratios tested. Figure 7(b) coupled with Figure 4(a) clearly indicates that the number of range searches has a direct impact on the performance of DBSCAN, IncDBSCAN, and $\mathcal{DISC}$.

3) MS-BFS and Epoch-based probing: The MS-BFS and epoch-based probing presented in Section IV are optimization techniques proposed for $\mathcal{DISC}$ so that the cost of checking the density-connectedness of minimal bonding cores can be further reduced. These two techniques can be applied independently of each other, and so can their effect be evaluated separately. Figure 8 shows the elapsed times taken by $\mathcal{DISC}$ for each dataset, when neither optimization was applied, only the epoch-based probing was applied, only the MS-BFS was applied, and both were applied. The elapsed times were measured with the stride size fixed to 5% of the window size.

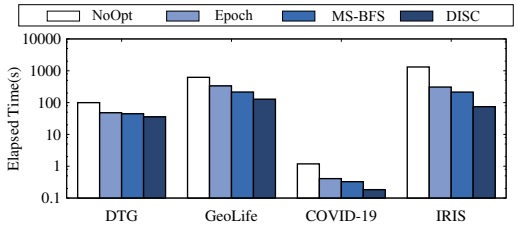


Fig. 8: Effects of optimizations

Each of the two optimization techniques achieved substantial reduction in the elapsed times even when they are applied alone separately. Between the two, the MS-BFS was slightly more effective than the epoch-based probing consistently over all the datasets. Apparently, the best performance was attained when both the optimization techniques were applied together, yielding more than an order of magnitude speedup in the case of the IRIS dataset.

E. Comparison with Summarization/Approximation-Based

DBSTREAM [8] and EDMSTREAM [7] are the state of the art summarization-based methods known for the low latency and high quality of clustering. The ρ-double-approximate DBSCAN (or ρ_2-DBSCAN in short) is another approximate clustering method, which is the dynamic version of ρ-approximate DBSCAN [17], [32]. We compared $\mathcal{DISC}$ with these three clustering methods to evaluate the trade-off between the processing speed and the quality of clustering. We focused on their capability of capturing the detailed shape of clusters by adopting rather small values for the distance threshold ϵ.

Two datasets, DTG and Maze, were used for this evaluation. For the real dataset DTG, we used the clustering results from

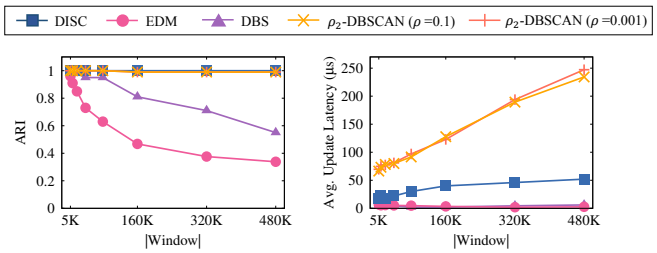


Fig. 9: Maze: ARI and Update Latency

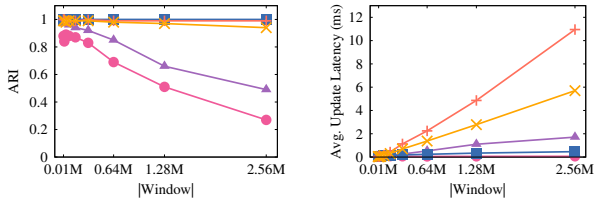


Fig. 10: DTG: ARI and Update Latency

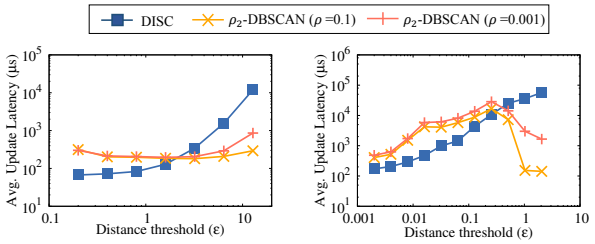
DBSCAN as the true labels. The synthetic dataset Maze was created by placing 100 random seeds in the 2-dimensional space. They spread out over time such that the trajectory of each seed was mapped to a single cluster. When the window size increased, trajectories became longer and closer to one another, and consequently the shape of clusters grew more complicated. We manually labeled each point in the Maze dataset so that each trajectory could be identified clearly as a separate cluster.

To evaluate the quality of clustering results, we measured the Adjusted Rand Index (ARI) [33] with a varying size of sliding window. The ARI measures how close the clustering results from different methods are to the true labels, and the measurements are in the range of -1 (lowest) to 1 (highest).

Figure 9 shows the quality measurements and the per-point update latency observed in the Maze dataset. The stride was 5% of the window size. (Since no deletion was supported by the summarization-based methods, only the insertion latency was measured for DBSTREAM and EDMSTREAM.) DBSTREAM and EDMSTREAM were evaluated with parameter settings that helped them achieve the best ARI. The same thresholds, τ and ϵ, were used for both ρ_2-DBSCAN and $\mathcal{DISC}$. The approximation parameter (ρ) of ρ_2-DBSCAN was set to 0.1 and 0.001 for low and high accuracy, respectively.

The summarization-based methods, EDMSTREAM and DBSTREAM, were much faster than the others but their clustering quality (measured in ARI) deteriorated very quickly as the sliding window grew larger. To achieve high ARI for a large window, summarization-based methods need to connect micro-clusters correctly.[2] EDMSTREAM connected them well when it dealt with a small number of large micro-clusters, but it did not do so well for a large number of small micro-clusters. DBSTREAM achieved better ARI than EDMSTREAM by utilizing additional information about the connectivity among micro-clusters, although that was not enough to sustain its ARI level. Both ρ_2-DBSCAN and $\mathcal{DISC}$ were able to detect

[2]Micro-cluster is a summarized representation of a set of adjacent points.



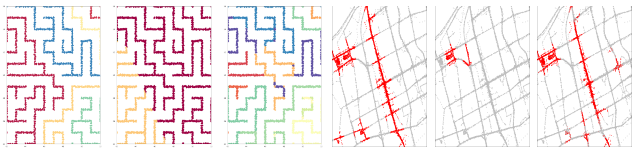
(a) Maze ($|Window| = 480K$) (b) DTG ($|Window| = 160K$)

Fig. 11: Update Latency with varying ϵ

accurate clusters but ρ_2-DBSCAN was up to five times slower than $\mathcal{DISC}$.

Similar trends were observed in the DTG dataset as shown in Figure 10 except for DBSTREAM, which was considerably slower than $\mathcal{DISC}$ across all the window sizes tested. This is because it has to manage a large number of micro-clusters to catch the details of fine-grained clusters. Although ρ_2-DBSCAN yielded high ARI comparable with that of $\mathcal{DISC}$, ρ_2-DBSCAN was much slower than all the other methods. With a larger approximation parameter (ρ), it ran faster but it was still slower than all the other methods.

Figure 11 compares $\mathcal{DISC}$ and ρ_2-DBSCAN with a varying distance threshold ϵ. We measured the cluster update latency when the stride was 5% of the window size. The overall trends were similar to those reported by Schubert *et al.* [19] about the static version of ρ_2-DBSCAN. For both datasets, $\mathcal{DISC}$ outperformed ρ_2-DBSCAN considerably with smaller ϵ values. $\mathcal{DISC}$ was outperformed by ρ_2-DBSCAN only when $\epsilon \geq 3.2$ for Maze and $\epsilon \geq 0.512$ for DTG. Beyond those crossover points, however, the clustering results were completely meaningless. Those distance thresholds were simply too large and only one huge cluster was detected covering all or almost all the data points in the window.



(a) DISC (b) EDM (c) DBS (d) DISC (e) EDM (f) DBS

Maze ($|W|$=480K) DTG ($|W|$=2.56M)

Fig. 12: Illustration of clusters found in Maze and DTG

Figure 12 illustrates the clusters discovered by different methods. Since ρ_2-DBSCAN and $\mathcal{DISC}$ produced the same (or almost the same) clusters, we omit the results from ρ_2-DBSCAN in the figure. Figures 12 (a)-(c) show clusters (in different colors) found in Maze by $\mathcal{DISC}$, EDMSTREAM, and DBSTREAM, respectively. Only $\mathcal{DISC}$ detected a connected component as a separate cluster correctly. Figures 12 (d)-(f) show clusters (marked in red color) found in DTG by the three methods. Only $\mathcal{DISC}$ detected the same clusters that matched those found by DBSCAN.

The experiments above confirm that $\mathcal{DISC}$ can detect clusters of high resolution with relatively low cost. Although the summarization-based methods can process streaming data at high speed, the quality of clustering results deteriorates significantly when the sliding window becomes large. The experiments also demonstrate that ρ_2-DBSCAN, the dynamic version of approximate DBSCAN, consumes an exceedingly high amount of computing time to detect clusters of high resolution, which was confirmed for its static version by the previous work [19]. ρ_2-DBSCAN was outperformed by $\mathcal{DISC}$ significantly for any practically useful range of distance thresholds.

VII. Related Work

Although a plethora of research has been conducted for various aspects of density-based clustering in the past (*e.g.*, parallel processing [34], [35], and parameter estimation [36]), this section focuses on briefly reviewing the existing work on density-based clustering for time-evolving data.

A. Incremental Approaches

Since the DBSCAN algorithm was proposed more than two decades ago, much research has been conducted to make the density-based clustering a viable option even for time-evolving or streaming data. The first one to note is Incremental DBSCAN [15], which updates existing clusters upon each individual data point being inserted to or deleted from the database.

The EXTRA-N method was proposed to address the *slow deletion* problem of density-based clustering [9]. It maintains multiple sub-windows to avoid processing a large number of range searches required for dealing with deleted data points. Maintaining many sub-windows, however, incurs high memory consumption and high computational overhead. This will lead to serious performance degradation when the clusters within the sliding window are updated frequently with a relatively small stride.

Recently, approximate versions of DBSCAN were proposed for both static and dynamic datasets [17], [32]. The approximation strategy enables them to update clusters incrementally. As is demonstrated in Section VI-E, however, these approximate algorithms consume computing resources excessively for detecting clusters of high resolution. In contrast, $\mathcal{DISC}$ can produce *exact* clusters of high resolution incrementally and more efficiently than the *approximate* algorithms.

B. Summarization-Based Approaches

Given the relatively high cost of density-based clustering for both static and streaming datasets, many summarization-based approaches have been proposed to expedite the continuous re-discovery of clusters for streaming data [6], [8], [16], [37]. In the summarization phase, they compute micro-clusters (or summarized objects) from streaming data with decaying density. Then, in the clustering phase, a clustering algorithm (*e.g.*, DBSCAN) is applied to the micro-clusters. Among them, DBSTREAM [8] achieves the better performance and

higher quality by considering shared density among micro-clusters [38]. EDMStream [7] is another summarization-based method. It detects splitting clusters incrementally by constructing a tree of micro-clusters with the density peaks [39].

VIII. Conclusions

In this paper, we study the problem of incremental clustering for streaming data and propose a new density-based incremental strategy called $\mathcal{DISC}$ optimized under the sliding window model. $\mathcal{DISC}$ is an elaborate algorithm based on novel ideas we define such as *ex-cores*, *neo-cores*, and their *minimal bonding cores*. The experimental evaluation corroborates the effectiveness of $\mathcal{DISC}$ in alleviating the computational burden of updating clusters frequently even in the presence of many deletions. This is a significant contribution given that the slow deletion problem has been a notoriously difficult challenge for density-based clustering algorithms since DBSCAN was introduced a quarter century ago.

References

[1] J. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations," in *the 5th Berkeley Symposium on Mathematical Statistics and Probability Data Mining*, Berkeley, CA, USA, 1965, pp. 281–297.

[2] A. K. Jain, "Data clustering: 50 Years Beyond K-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651 – 666, 2010.

[3] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A Framework for Clustering Evolving Data Streams," in *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 2003, pp. 81–92.

[4] J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. P. L. F. d. Carvalho, and J. a. Gama, "Data Stream Clustering: A Survey," *ACM Comput. Surv.*, vol. 46, no. 1, pp. 13:1–13:31, 2013.

[5] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan, "Clustering Data Streams: Theory and Practice," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 3, pp. 515–528, 2003.

[6] F. Cao, M. Ester, W. Qian, and A. Zhou, "Density-Based Clustering over an Evolving Data Stream with Noise," in *Proceedings of the 2006 SIAM International Conference on Data Mining*, Bethesda, MD, USA, 2006, pp. 328–339.

[7] S. Gong, Y. Zhang, and G. Yu, "Clustering Stream Data by Exploring the Evolution of Density Mountain," *Proc. VLDB Endow.*, vol. 11, no. 4, pp. 393–405, 2017.

[8] M. Hahsler and M. Bolaños, "Clustering Data Streams Based on Shared Density between Micro-Clusters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 6, pp. 1449–1461, 2016.

[9] D. Yang, E. A. Rundensteiner, and M. O. Ward, "Neighbor-based Pattern Detection for Windows over Streaming Data," in *Proceedings of the 12th International Conference on Extending Database Technology*, Saint Petersburg, Russia, 2009, pp. 529–540.

[10] Yang, Di and Rundensteiner, Elke A. and Ward, Matthew O., "Summarization and Matching of Density-based Clusters in Streaming Environments," *Proc. VLDB Endow.*, vol. 5, no. 2, pp. 121–132, 2011.

[11] P. Lee, L. V. S. Lakshmanan, and E. E. Milios, "Incremental Cluster Evolution Tracking from Highly Dynamic Network Data," in *30th IEEE ICDE Conference*, Chicago, IL, USA, 2014, pp. 3–14.

[12] J. Dromard, G. Roudiere, and P. Owezarski, "Online and Scalable Unsupervised Network Anomaly Detection Method," *IEEE Transactions on Network and Service Management*, vol. 14, no. 1, pp. 34–47, 2017.

[13] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *Proceedings of the 2nd KDD Conference*, Portland, Oregon, 1996, pp. 226–231.

[14] T. Zhang, B. Ramakrishnan, and M. Livny, "BIRCH: An Efficient Data Clustering Method for Very Large Databases," in *Proceedings of the 1996 ACM SIGMOD Conference*, Montreal, Canada, 1996, pp. 103–114.

[15] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu, "Incremental Clustering for Mining in a Data Warehousing Environment," in *Proceedings of the 24th VLDB Conference*, San Francisco, CA, USA, 1998, pp. 323–333.

[16] Y. Chen and L. Tu, "Density-based Clustering for Real-time Stream Data," in *Proceedings of the 13th ACM SIGKDD Conference*, San Jose, California, USA, 2007, pp. 133–142.

[17] J. Gan and Y. Tao, "Dynamic Density Based Clustering," in *Proceedings of the 2017 ACM SIGMOD Conference*, Chicago, Illinois, USA, 2017, pp. 1493–1507.

[18] S. Lühr and M. Lazarescu, "Incremental Clustering of Dynamic Data Streams Using Connectivity Based Representative Points," *Data & Knowledge Engineering*, vol. 68, no. 1, pp. 1 – 27, 2009.

[19] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN," *ACM Transactions on Database Systems*, vol. 42, no. 3, Jul. 2017.

[20] S. Venkatasubramanian, "Clustering on Streams," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer, 2009, pp. 378–383.

[21] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," in *Proceedings of the 21st ACM PODS Conference*, Madison, Wisconsin, 2002, pp. 1–16.

[22] J. Gama, *Knowledge Discovery from Data Streams*, 1st ed. Porto, Portugal: Chapman & Hall/CRC, 2010.

[23] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General Incremental Sliding-window Aggregation," *Proc. VLDB Endow.*, vol. 8, no. 7, pp. 702–713, 2015.

[24] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining Stream Statistics over Sliding Windows (Extended Abstract)," in *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, California, 2002, pp. 635–644.

[25] Y. Zhu and D. Shasha, "StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time," in *Proceedings of the 28th VLDB Conference*, Hong Kong, China, 2002, pp. 358–369.

[26] E. Cohen and M. J. Strauss, "Maintaining Time-Decaying Stream Aggregates," in *Proceedings of the 22nd ACM PODS Conference*, San Diego, California, 2003, p. 223–233.

[27] M. Then, M. Kaufmann, F. S. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo, "The More the Merrier: Efficient Multi-Source Graph Traversal," *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 449–460, 2014.

[28] S. Optac, "How to Use a Digital Tachograph," https://www.optac.info/uk/digital-tachograph/, Stoneridge.

[29] Y. Zheng, H. Fu, X. Xie, W.-Y. Ma, and Q. Li, *Geolife GPS trajectory dataset - User Guide*, Microsoft, July 2011, https://www.microsoft.com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide/.

[30] R. Lamsal, "Coronavirus (COVID-19) Geo-tagged Tweets Dataset," http://dx.doi.org/10.21227/fpsb-jz61, 2020.

[31] *IRIS*, Incorporated Research Institutions for Seismology, http://service.iris.edu/fdsnws/event/1/.

[32] J. Gan and Y. Tao, "DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation," in *Proceedings of the 2015 ACM SIGMOD Conference*, Melbourne, Victoria, Australia, 2015, pp. 519–530.

[33] L. Hubert and P. Arabie, "Comparing Partitions," *Journal of Classification*, vol. 1, pp. 193–218, 1985.

[34] H. Song and J.-G. Lee, "RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning," in *Proceedings of the 2018 ACM SIGMOD Conference*, Houston, TX, USA, 2018, pp. 1173–1187.

[35] Y. Wang, Y. Gu, and J. Shun, "Theoretically-Efficient and Practical Parallel DBSCAN," in *Proceedings of the 2020 ACM SIGMOD Conference*, Portland, OR, USA, 2020, p. 2555–2571.

[36] J. Hou, H. Gao, and X. Li, "DSets-DBSCAN: A Parameter-Free Clustering Algorithm," *IEEE Transactions on Image Processing*, vol. 25, no. 7, pp. 3182–3193, 2016.

[37] I. Ntoutsi, A. Zimek, T. Palpanas, P. Kröger, and H.-P. Kriegel, "Density-based Projected Clustering over High Dimensional Data Streams," in *Proceedings of the 2012 SIAM International Conference on Data Mining*, München, Germany, 2012, pp. 987–998.

[38] M. Carnein, D. Assenmacher, and H. Trautmann, "An Empirical Comparison of Stream Clustering Algorithms," in *Proceedings of the Computing Frontiers Conference*, Siena, Italy, 2017, p. 361–366.

[39] A. Rodriguez and A. Laio, "Clustering by fast search and find of density peaks," *Science*, vol. 344, no. 6191, pp. 1492–1496, 2014.