Dynamic Density Based Clustering

Junhao Gan
University of Queensland
Brisbane, Australia
j.gan@uq.edu.au

Yufei Tao
University of Queensland
Brisbane, Australia
taoyf@itee.uq.edu.au

ABSTRACT

Dynamic clustering—how to efficiently maintain data clusters along with updates in the underlying dataset—is a difficult topic. This is especially true for *density-based clustering*, where objects are aggregated based on transitivity of proximity, under which deciding the cluster(s) of an object may require the inspection of numerous other objects. The phenomenon is unfortunate, given the popular usage of this clustering approach in many applications demanding data updates.

Motivated by the above, we investigate the algorithmic principles for dynamic clustering by DBSCAN, a successful representative of density-based clustering, and ρ-approximate DBSCAN, proposed to bring down the computational hardness of the former on *static* data. Surprisingly, we prove that the ρ-approximate version *suffers from the very same hardness when the dataset is fully dynamic*, namely, when both insertions and deletions are allowed. We also show that this issue goes away as soon as tiny further relaxation is applied, yet still ensuring the same quality—known as the "sandwich guarantee"—of ρ-approximate DBSCAN. Our algorithms guarantee near-constant update processing, and outperform existing approaches by a factor over two orders of magnitude.

CCS Concepts

•**Theory of computation** → **Data structures and algorithms for data management;**

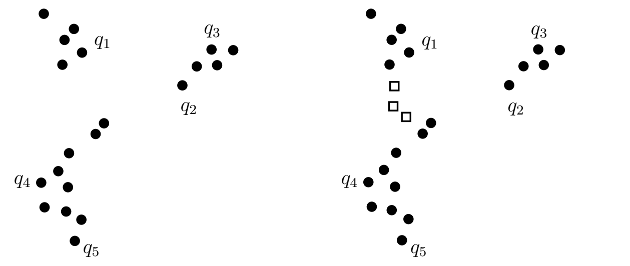
Keywords

Approximate DBSCAN; Dynamic Clustering; Algorithms

1. INTRODUCTION

Clustering is one of the most important topics in data mining and machine learning, and has been very extensively studied (see [13,22] and their bibliographies). An important but notoriously difficult issue is how to *update* the clusters when objects are inserted and deleted from the underlying dataset [4,8,15,17–21]. This is especially true when the clustering problem is *mass-correlated*, namely, the cluster of an object o cannot be decided by looking at o alone, but instead, must also take into account a potentially large number of other objects as well. Adding to the difficulty is the fact that, a single update may even affect more than one cluster: an insertion causing multiple clusters to merge, or a deletion breaking up a cluster into several.

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA.
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3035918.3064050



(a) A dataset of 3 clusters (b) 2 clusters merge after insertions

Figure 1: Dynamic density-based clustering

Density-based clustering, which aggregates objects by "transitivity of proximity", is heavily mass-correlated. A highly successful representative in this category is the *DBSCAN* method by Ester et al. [9]. Figure 1a shows an example dataset, where intuitively there are three clusters—each being a "cloud of points" with an irregular shape. Figure 1b demonstrates the effect of 3 insertions shown in boxes, which merge the left two clusters by creating a "connection path". Deleting those 3 boxes reverses the effect by breaking up a cluster into two.

A New Operation: *Cluster-Group-By* Query. We are interested in algorithms for maintaining density-based clusters on a dynamic set P of d-dimensional points. An immediate question is how to properly approach the problem in the first place. An obvious attempt is to define the problem as: "an algorithm should support fast updates, and in the meantime be prepared to return all the clusters any time upon requested". However, the cluster reporting *itself* already demands $\Omega(n)$ cost, where n is the number of points in P. This is at odds with the conventional database wisdom that "queries" should have response time significantly shorter than $O(n)$.

We eliminate the issue by introducing a novel query type called *cluster-group-by* (C-group-by), which makes the dynamic clustering problem much more interesting:

> Given an arbitrary *subset* Q of P, a C-group-by query groups the points of Q by the clusters they belong to.

Figure 1a shows a query with $Q = \{q_1, q_2, q_3, q_4, q_5\}$, which should return $\{q_1\}$, $\{q_2, q_3\}$, and $\{q_4, q_5\}$ indicating how they should be divided based on the clustering. The same query on Figure 1b returns $\{q_1, q_4, q_5\}$ and $\{q_2, q_3\}$.

method	update	C-group-by query	remark	reference
exact DBSCAN $d = 2$	$\tilde{O}(1)$	$\tilde{O}(\lvert Q \rvert)$	fully dynamic	this paper
exact DBSCAN $d \geq 3$	either $\Omega(n^{1/3})$ insertion or $\Omega(\lvert Q \rvert^{4/3})$ query[†]		even if insertions only	corollary of [10]
ρ-approx. DBSCAN $d \geq 3$	$\tilde{O}(1)$ insertion	$\tilde{O}(\lvert Q \rvert)$	insertions only	this paper
ρ-approx. DBSCAN $d \geq 3$	either $\tilde{\Omega}(n^{1/3})$ update or $\tilde{\Omega}(n^{1/3})$ query[†] even if $\lvert Q \rvert = 2$		fully dynamic	this paper
ρ-double-approx. DBSCAN $d \geq 3$	$\tilde{O}(1)$	$\tilde{O}(\lvert Q \rvert)$	fully dynamic	this paper

[†]subject to the hardness of *unit-spherical emptiness checking* (USEC)

Table 1: Dynamic hardness of DBSCAN variants

By simply setting Q to P, the C-group-by query degenerates into returning all the clusters. In practice, however, a user is rarely interested in the entire dataset. Instead, s/he is much more likely to raise questions regarding selected objects, e.g., "*are stocks X, Y in the same cluster?*", or "*break the 10 stocks by the clusters that their profiles belong to in the entire stock database*." C-group-by queries aim to answer these questions with time *proportional only to* $|Q|$, as opposed to $|P|$.

Hardness of Dynamic DBSCAN and Approximation. Recently, Gan and Tao [10] proved that, when $d \geq 3$, any DBSCAN algorithm must incur $\Omega(n^{4/3})$ worst-case time to cluster n static points (subject to the *USEC hardness* as will be reviewed in Section 2). Unfortunately, this implies that no dynamic DBSCAN algorithm can be fast in both insertions and queries, as explained below.

Suppose, on the contrary, that an algorithm could process an insertion in $\tilde{O}(1)$ time (where notation $\tilde{O}(\cdot)$ hides a polylog factor), and a query in $\tilde{O}(|Q|)$ time. We would be able to solve the *static* DBSCAN problem using the dynamic algorithm by performing n insertions followed by a C-group-by query with $Q = P$. The total cost would be only $\tilde{O}(n)$ which, however, is $o(n^{4/3})$—violating the impossibility result of [10]! To be practically useful, a dynamic algorithm must support an update in $\tilde{O}(1)$ time and a query in $\tilde{O}(|Q|)$ time. The above reduction suggests that no such algorithms can exist for DBSCAN, even if all the updates are insertions.

DBSCAN admits a *ρ-approximate* version [10] that can be settled in only $O(n)$ expected time, and thus avoids the above pitfall. As reviewed in the next section, the approximate version returns provably the same clusters as DBSCAN, unless the DBSCAN clusters are *unstable*: they change even under small perturbation to the clustering parameters. The unstable situation turns out to be the culprit for the hardness of (exact) DBSCAN. Indeed, accepting slightly altered results in those situations allows huge improvement from $\Omega(n^{4/3})$ to $O(n)$ [10].

Our Contributions. Lack of understanding on the computational efficiency of *dynamic* DBSCAN has become a serious issue, given the vast importance of this clustering technique, and the dynamic nature of numerous practical datasets in modern applications. Motivated by this, the current paper presents a comprehensive study on dynamic density-based clustering algorithms. Our contributions can be summarized as follows.

- Fully Dynamic 2D Exact Algorithm: When $d = 2$, we present an algorithm for (exact) DBSCAN that supports each insertion in $\tilde{O}(1)$ amortized time, and answers a C-group-by query in $\tilde{O}(|Q|)$ time.
- Fast Insertion-Only ρ-Approximate Algorithms: A dataset is *semi-dynamic*, if data points are only appended, but never deleted. In this case, we propose a ρ-approximate DBSCAN algorithm that supports each insertion in $\tilde{O}(1)$ amortized time, and answers a C-group-by query in $\tilde{O}(|Q|)$ time. The result holds for any fixed dimensionality d.
- Fully Dynamic ρ-Approximate DBSCAN Is Hard! A dataset is *fully-dynamic*, if data points can be inserted and deleted arbitrarily. We prove that, when $d \geq 3$, no ρ-approximate DBSCAN algorithm can be efficient in both updates and C-group-by queries at the same time! Specifically, such an algorithm must use $\tilde{\Omega}(n^{1/3})$ time either to process an update, or to answer a query—neither complexity is acceptable in practice (notation $\tilde{\Omega}(.)$ hides a polylog factor). This is true even if $|Q| = 2$ for all queries!
- ρ-Double-Approx. DBSCAN and Fully Dynamic: We show how to slightly relax ρ-approximate DBSCAN—into what we call *ρ-double-approximate DBSCAN*—to remove the above computational hardness. The relaxation leads to a fully-dynamic algorithm that processes an update in $\tilde{O}(1)$ amortized time, and answers a C-group-by query in time $\tilde{O}(|Q|)$. The new proposition preserves the clustering quality of (exact) DBSCAN in the same way (known as the *"sandwich guarantee"*) as ρ-approximate DBSCAN! In other words, the double approximation offers an *alternative* way to reach the *same* goal as ρ-approximate DBSCAN, without sharing the latter's deficiencies. The result holds for any fixed dimensionality d.
- Empirical Evaluation: We present experiments with the stringent requirement that ρ-double-approximate DBSCAN should *always* guarantee the same result as the ρ-approximate counterpart. The new algorithms demonstrate excellent running time for both updates and queries, and outperform the state of the art by a factor up to over two orders of magnitude.

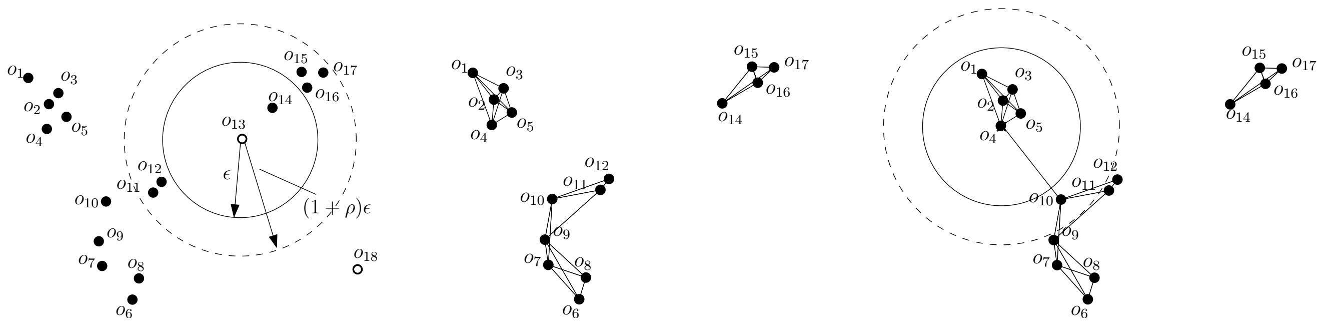
The dynamic hardness of different DBSCAN variants is summarized in Table 1. With these results, the dynamic tractability (i.e., polylog vs. polynomial) in all the fixed dimensionalities and update schemes has become well understood.

The rest of the paper is organized as follows. The next section reviews the basic concepts and properties of DBSCAN and its ρ-approximate version. Then, Section 3 formally defines the dynamic clustering problem studied in this work. Section 4 presents a generic framework that captures all the algorithms proposed in this paper. Section 5 elaborates on our semi-dynamic solutions to ρ-approximate DBSCAN. Section 6 proves our impossibility result for fully dynamic ρ-approximate DBSCAN, and introduces our "double-approximate" version of DBSCAN, for which Section 7 describes fast fully-dynamic algorithms. Section 8 reports the results of our experimental evaluation. Finally, Section 9 concludes the paper with a summary of our findings.

2. PRELIMINARIES

This section paves the foundation for our technical discussion by clarifying the basic concepts and properties of DBSCAN and its ρ-approximate version.

DBSCAN. Let P be a set of points in d-dimensional space $\mathbb{R}^d$. DBSCAN [9] defines a *unique* set of clusters on P based on two parameters: (i) a positive real value ϵ, and (ii) a small positive integer



(a) Dataset (b) Core graph (c) One possible ρ-approximate core graph

Figure 2: Illustration of DBSCAN and ρ-approximate DBSCAN ($\rho = 0.5$, $MinPts = 3$)

$MinPts$, which can be regarded as a constant. Next, we review how the clusters are formed using graph terminology.

Given a point $p \in P$, we use $B(p, r)$ to represent the ball that is centered at p, and has radius r. The point is said to be a *core point* if $B(p, \epsilon)$ covers at least $MinPts$ points of P (including p itself); otherwise, it is a *non-core point*. To illustrate, consider the dataset of 18 points in Figure 2a, where ϵ is the radius of the inner solid circle, and $MinPts = 3$. The core points have been colored black, while the non-core points colored white. The dashed circle can be ignored for the time being.

DBSCAN clusters are defined in two steps. The first one focuses exclusively on the core points, and groups them into preliminary clusters. The second step determines how the non-core points should be assigned to the clusters. Next, we explain the two steps in detail.

Step 1: Clustering Core Points. It will be convenient to imagine an undirected *core graph* G on P—this graph is conceptual and need not be materialized. Specifically, each vertex of G corresponds to a distinct *core* point in P. There is an edge between two core points (a.k.a. vertices) p_1, p_2 if and only if $dist(p_1, p_2) \le \epsilon$, where $dist(\cdot, \cdot)$ represents the Euclidean distance between two points. Figure 2b shows the core graph for the dataset of Figure 2a.

Each connected component (CC) of G constitutes a preliminary cluster. In Figure 2b, there are 3 CCs (a.k.a. preliminary clusters). Note that every core point belongs to *exactly one* preliminary cluster.

Step 2: Non-Core Assignment. This step augments the preliminary clusters with non-core points. For each non-core point p, DBSCAN looks at every core point $p_{core} \in B(p, \epsilon)$, and assigns p to the (only) preliminary cluster containing p_{core}. Note that, in this manner, p may be assigned to zero, one, or more than one preliminary cluster. After all the non-core points have been assigned, the preliminary clusters become final clusters.

It should be clear from the above that the DBSCAN clusters are uniquely defined by the parameters ϵ and $MinPts$, but they are not necessarily disjoint. A non-core point may belong to multiple clusters, while a core point must exist only in a single cluster. It is possible that a non-core point is not in any cluster; such a point is called *noise*.

In Figure 2a, there are two non-core points o_{13} and o_{18}. Since $B(o_{13}, \epsilon)$ covers o_{14}, o_{13} is assigned to the preliminary cluster of o_{14}. $B(o_{18}, \epsilon)$, however, covers no core points, indicating that o_{18} is noise. The final DBSCAN clusters are $\{o_1, o_2, ..., o_5\}$, $\{o_6, o_7, ..., o_{12}\}$, $\{o_{13}, o_{14}, ..., o_{17}\}$.

Remark. DBSCAN can also be defined under the notion of *"density-reachable"*; see [9]. The above graph-based definition is equivalent, perhaps more intuitive, and allows a simple extension to ρ-approximate DBSCAN, as we will see later.

Hardness of DBSCAN and USEC. It is easy to see that the DBSCAN clusters on a set P of n points can be computed in $O(n^2)$ time, noticing that the core graph G has $O(n^2)$ edges. However, clever algorithms should produce the clusters *without* generating all the edges, and thus, avoid the quadratic trap. Indeed, when $d = 2$, the clusters can be computed in only $O(n \log n)$ time [11].

It would be highly desired to find an algorithm of $\tilde{O}(n)$ time for $d \ge 3$, but recently Gan and Tao [10] have essentially dispelled the possibility. They proved an $O(n)$-time reduction from the *unit-spherical emptiness checking* (USEC) *problem* to DBSCAN. In other words, any $T(n)$-time DBSCAN algorithm implies that USEC problem can be solved with $O(T(n))$ time.

In USEC, we are given a set S_{red} of red points and a set S_{blue} of blue points in $\mathbb{R}^d$. All the points have distinct coordinates on every dimension. The objective is to determine whether there exist a red point p_{red} and a blue point p_{blue} such that $dist(p_{red}, p_{blue}) \le 1$ (the distance threshold 1 can be replaced with any positive value by scaling). The problem has a lower bound of $\Omega(n^{4/3})$ for $d \ge 5$ in a broad class of algorithms [6,7]. For $d = 3$ and 4, beating the bound has been a grand open problem in theoretical computer science, and is widely believed [6] to be impossible. By the reduction of [10], no DBSCAN algorithm can have running time $o(n^{4/3})$ in $d \ge 5$; in $d = 3$ and 4, this is also true unless unlikely ground-breaking improvements could be made on 3D USEC.

Approximation and the Sandwich Guarantee. Gan and Tao [10] developed *ρ-approximate DBSCAN*, which returns almost the same clusters as exact DBSCAN by offering a strong *sandwich guarantee* that will be introduced shortly. In contrast to the high time complexity of the latter, the approximate version takes only $O(n)$ expected time to compute for any constant $\rho > 0$.

Besides the parameters ϵ and $MinPts$ inherited from DBSCAN, the approximate version accepts a third parameter ρ, which is a small positive constant less than 1, and controls the clustering precision. Its clusters can also be defined in the same two steps as in exact DBSCAN, as explained below.

Step 1: Clustering Core Points. It will also be convenient to follow a graph-based approach. Let us define an undirected *ρ-approximate core graph* G_ρ on the dataset P—again, this graph is conceptual and need not be materialized. Each vertex of G_ρ corresponds to a distinct core point in P. Given two core points p_1, p_2, whether or not G_ρ has an edge between their vertices is determined as:

- The edge definitely exists if $dist(p_1, p_2) \le \epsilon$.

- The edge definitely does not exist if $dist(p_1, p_2) > (1 + \rho)\epsilon$.
- Don't care, otherwise.

Each preliminary cluster is still a CC, but of G_ρ. Unlike the core graph G, G_ρ may not be unique. This flexibility is the key to the vast improvement in time complexity [10].

To illustrate, consider the dataset of Figure 2a again with the ϵ shown and $MinPts = 3$, but also with $\rho = 0.5$ (the radius of the dashed circle indicates the length of $(1 + \rho)\epsilon$). Figure 2c illustrates a possible ρ-approximate core graph. Attention should be paid to the edge (o_4, o_{10}). Note (from the circles in Figure 2c) that $\epsilon < dist(o_4, o_{10}) < (1 + \rho)\epsilon$—this belongs to the "don't-care" case meaning that there may or may not be an edge (o_4, o_{10}). If the edge exists (as in Figure 2c), there are 2 CCs (i.e., preliminary clusters); otherwise, the ρ-approximate core graph is the same as in Figure 2b, giving 3 preliminary clusters.

Step 2: Non-Core Assignment. Each non-core point p may be assigned to zero, one, or multiple preliminary clusters. Specifically, let S be a CC of G_ρ. Whether p should be added to the preliminary cluster of S is determined as:

- Yes, if S has a core point in $B(p, \epsilon)$.
- No, if S has no core point in $B(p, (1 + \rho)\epsilon)$.
- Don't care, otherwise.

The preliminary clusters after all the assignment constitute the final clusters.

As mentioned, o_{13} and o_{18} are the only two non-core points in Figure 2a. While o_{18} is still a noise point, the case of o_{13} is more interesting. First, it *must* be assigned to the preliminary cluster of o_{14}, just like exact DBSCAN. Second, it *may or may not* be assigned to the preliminary cluster of o_{12} (also the cluster of o_{14}). Either case is regarded as a correct result.

Sandwich Guarantee. Recall that the clusters of exact DBSCAN are uniquely determined by the parameters ϵ and $MinPts$. Now imagine we slightly increase ϵ by an amount no more than $\rho\epsilon$. Have the clusters of DBSCAN changed? If yes, it means that the original choice of ϵ is *unstable*—clusters are susceptible even to a tiny perturbation to ϵ. If no, then the sandwich guarantee asserts that ρ-approximate DBSCAN returns precisely the same clusters as DBSCAN. In Figure 2, the clusters changed because ϵ was deliberately set to be a large value of 0.5. In [10], the recommended value for practical data is actually 0.001.

We refrain from elaborating on the formal description of the sandwich guarantee at this moment. We will come back to this in Section 6 where we prove that our new "double-approximate" DBSCAN offers just the same guarantee.

Remark. Note, interestingly, that when $\rho = 0$, there are no "don't-care" scenarios such that ρ-approximate DBSCAN degenerates into exact DBSCAN. Hence, the former actually subsumes the latter as a special case.

3. PROBLEM DEFINITION AND STATE OF THE ART

The Problem of Dynamic Clustering. We now provide a formal formulation of dynamic clustering, using the C-group-by query as the key stepping stone. Our approach is to define the problem in a way that is orthogonal to the semantics of clusters, so that the problem remains valid regardless of whether we have DBSCAN or any of its approximate versions in mind.

Let P be a set of points in $\mathbb{R}^d$ that is subject to *updates*, each of which inserts a new point to P, or deletes an existing point from P. We are given a *clustering description* which specifies correct ways to cluster P. The description is what distinguishes DBSCAN from, e.g., ρ-approximate DBSCAN.

Suppose that, by the clustering description, $\mathcal{C}(P)$ is a legal set of clusters on the current contents of P. Without loss of generality, assume that $\mathcal{C}(P) = \{C_1, C_2, ..., C_x\}$, where x is the number of clusters, and C_i $(1 \le i \le x)$ is a subset of P. Note that the clusters do not need to be disjoint.

Given an arbitrary subset Q of P, a *cluster-group-by* (C-group-by) *query* must return for every $C_i \in \mathcal{C}(P)$ $(i \in [1, x])$:

- Nothing at all, if $C_i \cap Q = \emptyset$
- $C_i \cap Q$, otherwise.

This definition has several useful properties:

- It breaks *only* the points of Q by how they should appear together in the clusters of P. Points in $P \setminus Q$ are not reported at all, thus avoiding "cheating algorithms" that use "expensive report time" as an excuse for high processing cost.
- When $Q = P$, the query result $Q(P)$ is simply $\mathcal{C}(P)$.
- All the query results must be based on the *same* $\mathcal{C}(P)$. This prevents another form of "cheating" when the clustering description permits multiple legal possibilities of $\mathcal{C}(P)$. Specifically, the algorithm can no longer argue that the results $Q_1(P)$ and $Q_2(P)$ of two queries Q_1 and Q_2 should both be "correct" because $Q_1(P)$ is defined on one possible $\mathcal{C}(P)$, while $Q_2(P)$ is defined on another. Instead, they must be consistently defined on the same $\mathcal{C}(P)$—the one output by the query with $Q = P$.

Our objective is to design an algorithm that is fast in processing both updates and queries. We distinguish two scenarios: (i) *semi-dynamic*: where all the updates are insertions, and (ii) *fully-dynamic*, where the updates can be arbitrary insertions and deletions.

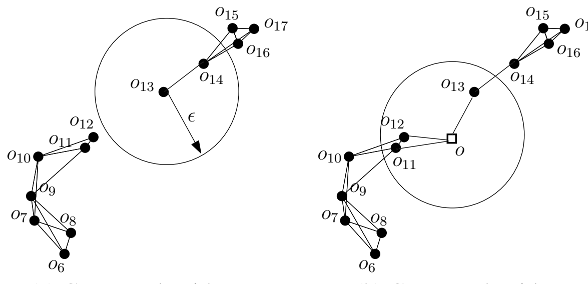
We consider that the dimensionality d is small such that $(\sqrt{d})^d$ is an acceptable hidden constant. All our theoretical results will carry this constant, and hence, are suitable only for low dimensionality. Our experiments run up to $d = 7$.

Dynamic Exact DBSCAN [8]. Dynamic maintenance of density-based clusters has been studied by Ester et al. [8] for exact DBSCAN. Next, we review their method—named *incremental DBSCAN* (*IncDBSCAN*)—assuming $MinPts = 1$ so that all the points of P are core points. This allows us to concentrate on the main ideas without the relatively minor details of handling non-core points.

Insertion. Recall that, for exact DBSCAN, the clusters $\mathcal{C}(P)$ are uniquely determined by the input parameters ϵ and $MinPts$. Given a new point p_{new}, the insertion algorithm retrieves all the points in $B(p_{new}, \epsilon)$, and then merges the clusters of those points into one.

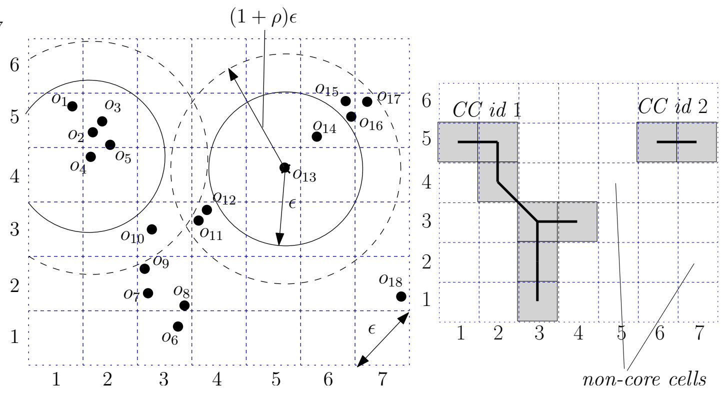
The correctness can be seen from the core graph G, where effectively an edge is added between p_{new} and every other point in $B(p_{new}, \epsilon)$ (remember: this view is conceptual, and G does not need to be materialized). Figure 3a shows the G before the insertion, which has two CCs. To insert point o as in Figure 3b, the algorithm finds the points o_{11}, o_{12}, and o_{13} in $B(p_{new}, \epsilon)$. The two clusters of those points are merged—the newly added edges (o, o_{11}), (o, o_{12}), (o, o_{13}) in Figure 3b connect the two CCs into one.

In merging the clusters, IncDBSCAN does not modify the cluster ids of the points in the affected clusters, which can be prohibitively



(a) Core graph without o (b) Core graph with o

Figure 3: Illustration of the IncDBSCAN method



(a) Grid and cells (b) A grid graph

Figure 4: Our grid-graph framework ($MinPts = 3$)

expensive because the number of such points can be exceedingly high. Instead, it remembers the "merging history" of the cluster ids.

Deletion. The deletion algorithm, in general, reverses the steps of insertion, except for a breadth first search (BFS) that is needed to judge whether (and how) a cluster has split into several.

Let p_{old} be the point being deleted. IncDBSCAN retrieves all the points in $B(p_{old}, \epsilon)$. In (the conceptual) G, all the edges between such points and p_{old} are removed, after which the CC of p_{old} may or may not be broken into separate CCs (e.g., in Figure 3b, the CC is torn into two only if o_{13}, o_{14}, or o is deleted). To find out, the deletion algorithm performs as many threads of BFS on G as the number of points in $B(p_{old}, \epsilon)$. If two threads "meet up", they are combined into one because they must still be in the same CC. As soon as only one thread is left, the algorithm terminates, being sure that no cluster split has taken place. Otherwise, the remaining threads continue until the end, in which case each thread has enumerated all the points in a new cluster that is spawned by the deletion. All those points can now be labeled with the same cluster id.

For example, suppose that we delete o in Figure 3b, which starts three threads of BFS from o_{11}, o_{12} and o_{13}, respectively. The resulting core graph reverts back to Figure 3a. The threads of o_{11} and o_{12} meet up into one, which eventually traverses the entire CC containing o_{11} and o_{12}. Similarly, the other thread traverses the entire CC containing o_{13}.

When a thread of BFS needs the adjacent neighbors of a point p_1 in G, the algorithm finds all the other points $p_2 \in B(p_1, \epsilon)$ through a *range query* [3,12]. Every such p_2 is an adjacent neighbor of p_1. This essentially "fetches" the edge between p_1 and p_2.

Query. The algorithm of [8] can easily answer a C-group-by query Q by grouping the points of Q by their cluster ids (some ids need to be obtained from the merging history).

Drawbacks of IncDBSCAN. Both insertion and deletion start with a range query to extract the points in $B(p, \epsilon)$, which are called the *seed points* [8]. The query is expensive when p falls in a dense region of P where there are many seed points. The issue is more serious in a deletion, because multiple range queries are needed to perform BFS. The worst situation happens in a cluster split, where the number of range queries is simply huge.

4. THE OVERALL FRAMEWORK

All the DBSCAN variants (including the new one to be proposed in Section 6.2) accept parameters ϵ, $MinPts$, and ρ (for exact DBSCAN, $\rho = 0$). This permits us to extract a common structural framework behind all our solutions, as we describe in this section. The components of the framework will be instantiated differently in later sections for individual variants.

4.1 A Grid Graph Approach

The key idea behind our framework is to turn dynamic clustering into the problem of maintaining CCs (connected components) of a special graph.

Grid and Cells. We impose an arbitrary grid $\mathbb{D}$ in the data space $\mathbb{R}^d$, where each cell is a d-dimensional square with side length $\epsilon/\sqrt{d}$ on every dimension. This ensures that any two points in the same cell are within distance at most ϵ from each other.

Given a cell c of $\mathbb{D}$, we denote by $P(c)$ the set of points in P that are covered by c. We call c

- A *non-empty cell* if $P(c)$ contains at least one point.
- A *core cell* if $P(c)$ contains at least one core point.
- A *dense cell* if $|P(c)| \geq MinPts$, and a *sparse cell* if $1 \leq |P(c)| < MinPts$.

Given two cells c_1, c_2, we say that they are *ϵ-close* if the smallest distance between the boundary of c_1 and that of c_2 is at most ϵ.

Consider for instance the grid in Figure 4a, imposed on a set P of 18 points. Again, the radii of the solid and dashed circles indicate ϵ and $(1+\rho)\epsilon$, respectively; and $MinPts = 3$. The only non-core points are o_{13} and o_{18}. The core cells are shaded in Figure 4b. The non-core cells are $(5, 4)$ and $(7, 2)$; note that the minimum distance between the two cells is ϵ—hence, they are ϵ-close.

Grid Graph. In a *grid graph* $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, $\mathbb{V}$ is the set of core cells of $\mathbb{D}$, while $\mathbb{E}$ is a set of edges satisfying the following *CC requirement*:

> Let p_1, p_2 be two core points of P, and c_1 (or c_2, resp.) be the core cell that contains p_1 (or p_2, resp.). Then, p_1 and p_2 are in the same cluster *if and only if* c_1 and c_2 are in the same CC of $\mathbb{G}$.

The above requirement is fulfilled by using the following rules to decide if $\mathbb{E}$ should have an edge between two core cells $c_1, c_2 \in \mathbb{V}$:

- Yes, if there is a pair of core points $(p_1, p_2) \in P(c_1) \times P(c_2)$ satisfying $dist(p_1, p_2) \leq \epsilon$.
- No, if there is *no* pair of core points $(p_1, p_2) \in P(c_1) \times P(c_2)$ satisfying $dist(p_1, p_2) \leq (1+\rho)\epsilon$.
- Don't care, otherwise.

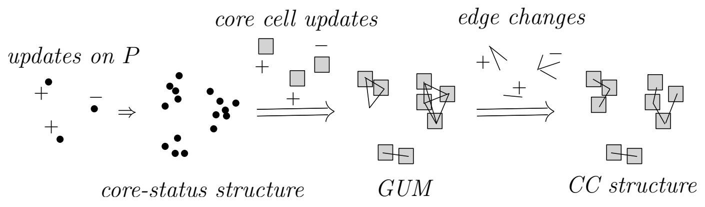


Figure 5: Flow of updates in our structures

$\mathbb{G}$ differs significantly from the core graph G and the ρ-approximate core graph of G_ρ as reviewed in Section 2. $\mathbb{G}$ has at most n vertices because there are at most n non-empty cells. If $\mathbb{G}$ has an edge between core cells c_1, c_2, they must be ϵ-close. A core cell can have $O\Big(\Big(\frac{\epsilon}{\epsilon/\sqrt{d}}\Big)^d\Big) = O((\sqrt{d})^d) = O(1)$ ϵ-close core cells (recall that our target is low dimensionality). Hence, $\mathbb{G}$ can have only $O(n)$ edges, which makes it suitable for materialization when the dimensionality d is low.

Figure 4b demonstrates the grid graph for the dataset in Figure 4a. Note that the edge between cells $(2, 4)$ and $(3, 3)$ fall into the don't-care case, because $\epsilon < dist(o_4, o_{10}) \le (1 + \rho)\epsilon$. That $\mathbb{G}$ satisfies the CC requirement can be seen together with the ρ-approximate core graph in Figure 2c. For example, o_3 and o_6 are in the same cluster, consistent with the fact that cells $(2, 5)$ and $(3, 1)$ are in the same CC of $\mathbb{G}$. Conversely, as cells $(2, 5)$ and $(6, 5)$ are in different CCs of $\mathbb{G}$, we know o_3 and o_{14} must be in different clusters.

4.2 Query Algorithm

All our solutions actually use the same algorithm to answer C-group-by queries. We explain the algorithm in this subsection, as well as the necessary data structures.

Core-Status Structure. We explicitly record whether each point in P is a core or non-core point. This structure maintains these core-status labels under insertions and deletions of the points in P. A semi-dynamic structure only needs to support insertions.

ρ-Approximate ϵ-Emptiness. Given a point q and a core cell c, an *emptiness query* $empty(q, c)$ returns:

- 1, if $P(c)$ has a core point p satisfying $dist(p, q) \le \epsilon$.
- 0, if *no* core point $p \in P(c)$ satisfies $dist(p, q) \le (1 + \rho)\epsilon$.
- 1 or 0 (don't care), otherwise.

As a furthermore requirement, if the output is 1, the query must also return a *proof point* $p \in P(c)$, which is a core point satisfying $dist(q, p) \le (1 + \rho)\epsilon$.

For example, for $q = o_{13}$ and $c =$ cell $(6, 5)$, the emptiness query must return 1 due to the presence of o_{14}. If c changes to cell $(3, 2)$, then the query must return 0. Setting $q = o_4$ and $c =$ cell $(3, 3)$ gives a don't-care case.

We maintain an *emptiness structure* on every core cell c to support (i) such queries efficiently, and (ii) insertions/deletions of core points in $P(c)$. Deletions are not needed if the structure is semi-dynamic.

CC Structure. We maintain a structure on $\mathbb{G}$ to support:

- *EdgeInsert*(c_1, c_2): Add an edge between core cells c_1, c_2 to $\mathbb{G}$.
- *EdgeRemove*(c_1, c_2): Remove the aforementioned edge.
- *CC-Id*(c): Given a core cell c, return a unique id of the CC of $\mathbb{G}$ where c belongs.

If a CC structure is semi-dynamic, it does not need to support *EdgeRemove*.

C-Group-By Query. Next, we clarify how to answer a C-group-by query Q. Divide Q into a set Q_1 of core points, and a set Q_2 of non-core points. This takes $O(|Q|)$ time using the core-status structure. For every core point $q \in Q_1$, we retrieve the core cell c covering q, perform *CC-Id*(c), and set the CC id as the cluster id of q.

A non-core point $q \in Q_2$, on the other hand, is "snapped" to the nearby core cells. Again, obtain the cell c covering q. If c is a core cell, assign the cluster id *CC-Id*(c) to q. In any case, we enumerate all the ϵ-close core cells c' of c. For every c', issue an emptiness query $empty(q, c')$. If the emptiness query returns 1, assign the output of *CC-Id*(c') as a cluster id of q. Note that q may get assigned multiple cluster ids.

We can now group the points in Q by cluster id. A non-core point belongs to as many groups as the number of its distinct cluster ids; if a non-core point has no cluster ids, it is a noise point.

Consider $Q = \{o_{13}, o_{14}, o_8\}$ in the dataset of Figure 4a. Q_1 includes core points o_{14} and o_8, which are in cells $(6, 5)$ and $(3, 2)$, respectively. Invoking *CC-Id* on $(6, 5)$ returns 2 (see Figure 4b), while doing so to $(3, 2)$ returns 1. Q_2 has only a single non-core point, in cell $(5, 4)$, whose ϵ-close core cells are $(4, 3)$, $(3, 3)$, $(3, 2)$, $(6, 5)$, and $(7, 5)$. We perform an emptiness query using o_{13} on each of those 5 cells. Suppose that the emptiness queries on $(4, 3)$ and $(6, 5)$ return 1, while the others 0. We thus assign two distinct CC ids to o_{13}: 1 and 2. The final result of the C-group-by query is therefore $\{o_{14}, o_{13}\}$, and $\{o_8, o_{13}\}$.

4.3 Graph Maintenance

To guarantee correctness, we must keep the grid graph $\mathbb{G}$ up-to-date along with the insertions and deletions on the underlying dataset P. This is accomplished through the collaboration of the core-status structure, GUM (see below), and the CC structure.

Graph Update Module (GUM). This module is responsible for maintaining the vertices and edges in $\mathbb{G}$.

Remark. Figure 5 illustrates the data flow in the internal workings of our update mechanism. The point insertions and deletions in P are fed into the core-status structure, which informs GUM about which cells have turned into core/non-core cells. Utilizing such information, GUM updates $\mathbb{G}$ by generating the necessary edge changes, which are passed to the CC structure for properly maintaining the CCs of $\mathbb{G}$.

Overall, our design will focus on GUM and the core-status structure. CC and emptiness structures have been well-studied in graph theory and computational geometry, respectively; it suffices to plug in the best existing structures suiting our purposes.

5. SEMI-DYNAMIC ALGORITHMS

This section presents maintenance algorithms for exact/approximate DBSCAN clustering when all the updates are insertions. We will do so by specializing the framework of the last section.

The Core-Status Structure. For each non-core point $p \in P$, we remember a *vicinity count* $vincnt(p)$ which equals the number of points of P covered by $B(p, \epsilon)$. By non-core definition, $vincnt(p) < MinPts$. Once $vincnt(p)$ reaches $MinPts$, p becomes a core point, after which we no longer keep track of such a count.

Let us see how to maintain the above information when a new point p_{new} is inserted. Let c_{new} be the cell of $\mathbb{D}$ that contains p_{new}. We start by checking if p_{new} is a core point as follows:

1 If c_{new} is dense, p_{new} must a core point (all the points in c_{new} are within distance ϵ from p_{new}).

2 Otherwise, we simply enumerate all the $O(1)$ ϵ-close cells c of c_{new}, and calculate the distances from p_{new} to all the points in $P(c)$. This way, we obtain the precise number of points in $B(p_{new}, \epsilon)$, noticing that any point within distance ϵ from p_{new} must be in an ϵ-close cell. The core-status of p_{new} can now be decided.

The appearance of p_{new} may increase the vicinity count $vincnt(p)$ of some non-core points p. Such p must be covered in cells c that are (i) sparse, and (ii) ϵ-close to c_{new}. We find all these points by simply visiting the $P(c)$ of all such c.

GUM. In general, whenever we have a new core point p_{core} (it may be p_{new}, or a point p that just has its $vincnt(p)$ increased), $\mathbb{G}$ may need to be updated. Let c_{core} be the cell covering p_{core}. If c_{core} just became a core cell, we add it into $\mathbb{V}$. In any case, new edges are potentially added to $\mathbb{E}$ as follows:

```
1 For every $\epsilon$-close cell $c$ of $c_{core}$ that currently has no edge with $c_{core}$ in $\mathbb{G}$
    1.1 Perform an emptiness query $empty(p_{core}, c)$.
    1.2 If the query returns 1, add $(c, c_{core})$ to $\mathbb{E}$ and call EdgeInsert $(c, c_{core})$.
```

Performance Guarantees. Using the best CC and emptiness structures under the semi-dynamic scheme, we prove in the appendix:

THEOREM 1. *For any fixed dimensionality d and fixed constant $\rho > 0$, there is a semi-dynamic ρ-approximate DBSCAN algorithm that processes each insertion in $\tilde{O}(1)$ amortized time, and answers a C-group-by query Q in $\tilde{O}(|Q|)$ time.*

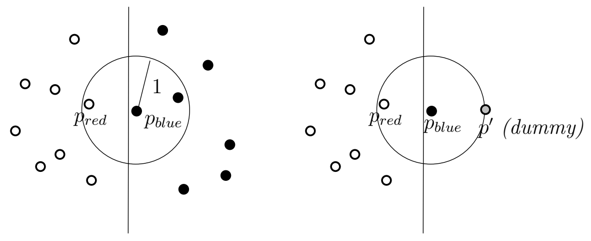
The same insertion and query efficiency can also be achieved in 2D space for exact DBSCAN.

6. DYNAMIC HARDNESS AND DOUBLE APPROXIMATION

We now come to perhaps the most surprising section of the paper. Recall that ρ-approximate DBSCAN was proposed to address the computational hardness of DBSCAN on *static* datasets. In Section 6.1, we will show that the ρ-approximate version suffers from the same hardness on *fully dynamic* datasets. Interestingly, the culprit this time is the definition of core point. This motivates the proposition of *ρ-double-approximate* DBSCAN in Section 6.2, where we also prove that the new proposition has a sandwich guarantee *as strong as* the ρ-approximate version.

6.1 Hardness of Dynamic ρ-Approximation

USEC with Line Separation. Next, we introduce the *USEC with line separation* (USEC-LS) *problem*, which has a subtle connection with ρ-approximate DBSCAN, as shown later.



(a) USEC-LS (b) Reduction to dynamic clustering

Figure 6: Illustration of our hardness proof

In USEC-LS, we are given a set S_{red} of red points and a set S_{blue} of blue points in $\mathbb{R}^d$, which are separated by a d-dimensional plane ℓ perpendicular to the first dimension, such that all the red points are on one side of ℓ, and all the blue points on the other. All the points have distinct coordinates on the first dimension. The objective, as with USEC (see Section 2), is to determine whether there exist $p_{red} \in S_{red}$ and $p_{blue} \in S_{blue}$ such that $dist(p_{red}, p_{blue}) \le 1$. We define $n = |S_{red}| + |S_{blue}|$. Figure 6a shows an example where the answer is "yes".

Recall from Section 2 that USEC is computationally hard. In the appendix, we prove that this is also true for USEC-LS:

LEMMA 1. *If we can solve USEC-LS in $o(n^{4/3})$ time, then we can solve USEC in $o(n^{4/3})$ time.*

Dynamic Hardness. Suppose that we have a ρ-approximate DBSCAN algorithm that handles an update (insertion/deletion) in $T_{upd}(n)$ amortized time, and answers a C-group-by query with $|Q| = 2$ in $T_{qry}(n)$ amortized time. Then:

LEMMA 2. *We can solve the USEC-LS problem in $O(n \cdot (T_{upd}(n) + T_{qry}(n))$ time.*

PROOF. Let x be the coordinate where the separation plane ℓ in USEC-LS intersects dimension 1. Without loss of generality, let us assume that the red points are on the *left* of ℓ, i.e., having coordinates less than x on dimension 1. Conversely, the blue points are on the *right* of ℓ. We solve USEC-LS using the given dynamic ρ-approximate DBSCAN algorithm A as follows.

```
1. Initialize a $\rho$-approximate DBSCAN instance with $\epsilon = 1$, $MinPts = 3$, and an arbitrary $\rho \ge 0$. Let $P$ be the input set, which is empty at this moment.
2. Use $A$ to insert all the red points into $P$.
3. For every blue point $p = (x_1, x_2, ..., x_d)$ (hence, $x_1 > x$), carry out the following steps:
    3.1 Use $A$ to insert $p$ into $P$.
    3.2 Use $A$ to insert a dummy point $p' = (x_1 + 1, x_2, ..., x_d)$ into $P$. That is, $p'$ has the same coordinates as $p$ on all dimensions $i \in [2, d]$, except for the first dimension where $p'$ has coordinate $x_1 + 1$. See Figure 6b for an illustration (where $p = p_{blue}$).
    3.3 Use $A$ to answer a C-group-by query with $Q = \{p, p'\}$. If the query returns the same cluster id for $p$ and $p'$, terminate the algorithm, and return "yes" to the USEC-LS problem.
    3.4 Use $A$ to delete $p'$ and $p$ from $P$.
4. Return "no" to the USEC-LS problem.
```

The running time of the algorithm is $O(n \cdot (T_{upd}(n) + T_{qry}(n))$ because we issue at most $2n$ insertions and $2n$ deletions, as well as n queries, in total. Next, we prove that the algorithm is correct.

Consider Lines 3.1-3.4. A crucial observation is that the dummy point p' must be a non-core point, because $B(p', \epsilon)$ contains only two points p, p'. Therefore, p' and p are placed into the same cluster by ρ-approximate DBSCAN *if and only if* p is a core point. However, p is a core point if and only if $B(p, \epsilon)$ covers at least 3 points, which must include p, p', and at least one point p'' on the other side of ℓ—red point p'' and blue point p are therefore within distance 1.

It is now straightforward to verify that our algorithm always returns the correct answer for USEC-LS. $\square$

THEOREM 2. *For any $\rho \geq 0$ and any dimensionality $d \geq 3$, any ρ-approximate DBSCAN algorithm must incur $\Omega(n^{1/3})$ amortized time either to process an update, or to answer a C-group-by query (even if $|Q| = 2$), unless the USEC problem in $\mathbb{R}^d$ could be solved in $o(n^{4/3})$ time.*

PROOF. Suppose that the algorithm were able to process an update and a query both in $o(n^{1/3})$ amortized time. By Lemma 2, we would solve USEC-LS in $o(n^{4/3})$ time which, by Lemma 1, means that we would solve USEC in $o(n^{4/3})$ time. $\square$

As explained in Section 2, for USEC, a lower bound of $\Omega(n^{4/3})$ is known [7] in $d \geq 5$, whereas beating the $O(n^{4/3})$ bound in $d = 3, 4$ is a major open problem in theoretical computational geometry, and believed to be impossible [6].

This is disappointing because DBSCAN succumbing to the hardness of USEC was what motivated ρ-approximate DBSCAN. Theorem 2 shows that the latter suffers from the same hardness when both insertions and deletions are allowed! Note that the theorem does not apply to the semi-dynamic update scheme because the deletions at Line 3.4 are essential. In fact, Theorem 1 already proved that efficient semi-dynamic algorithms exist for ρ-approximate DBSCAN.

Finally, it is worth pointing out that Theorem 2 holds even for $\rho = 0$, i.e., it is applicable to exact DBSCAN as well.

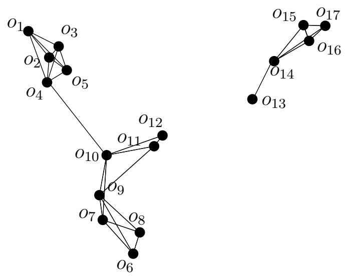
6.2 ρ-Double-Approximate DBSCAN and Sandwich Guarantee

The New Proposition. To enable both (fully-dynamic) update and query efficiency, we propose ρ-double-approximate DBSCAN, which takes the same parameters ϵ, $MinPts$, and ρ as ρ-approximate DBSCAN. Whether a point $p \in P$ is a core point is now decided in a relaxed manner:

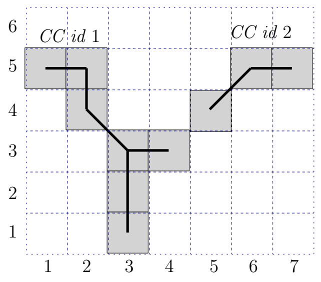
- Definitely a core point if $B(p, \epsilon)$ covers at least $MinPts$ points of P.
- Definitely not a core point if $B(p, (1 + \rho)\epsilon)$ covers less than $MinPts$ points of P.
- Don't care, otherwise.

A good example to illustrate this is point o_{13} in Figure 4a. Since $B(o_{13}, \epsilon)$ covers $2 < MinPts = 3$ points, o_{13} is not a core point under exact or ρ-approximate DBSCAN. Under double approximation, however, it falls into the don't-care case for $\rho = 0.5$, because $B(p, (1 + \rho)\epsilon)$ covers 7 points.

The clusters of ρ-double-approximate DBSCAN are defined by the same two-step approach of ρ-approximate DBSCAN (see Section 2), but with respect to the above core-point semantics. Swaying o_{13} into a core point, Figure 7a shows the ρ-double-approximate core graph (defined precisely as the ρ-approximate version), while Figure 7b gives the corresponding grid graph.



(a) One possible ρ-double-approx core graph



(b) Grid graph

Figure 7: Illustration of ρ-double approximation

Sandwich Guarantee. Recall that this is an attractive feature of ρ-approximate DBSCAN. Next, we prove that ρ-double-approximate DBSCAN provides just the same guarantee. Following the style of [10], we define:

- $\mathscr{C}_1$ as the set of clusters of exact DBSCAN with parameters $(\epsilon, MinPts)$.
- $\mathscr{C}_2$ as the set of clusters of exact DBSCAN with parameters $(\epsilon(1 + \rho), MinPts)$.
- $\mathscr{C}$ as a set of clusters that is a legal result of ρ-double-approximate DBSCAN with parameters ϵ, $MinPts$, and ρ.

Then, the sandwich guarantee is:

THEOREM 3. *The following statements are true: (i) For any cluster $C_1 \in \mathscr{C}_1$, there is a cluster $C \in \mathscr{C}$ such that $C_1 \subseteq C$, and (ii) for any cluster $C \in \mathscr{C}$, there is a cluster $C_2 \in \mathscr{C}_2$ such that $C \subseteq C_2$.*

The proof can be found in the appendix. Note that the theorem is purposely worded exactly the same as Theorem 3 of [10].

7. FULLY DYNAMIC ALGORITHMS

This section presents our algorithms for maintaining ρ-double-approximate DBSCAN clusters under both insertions and deletions (again, exact DBSCAN is captured with $\rho = 0$). We will achieve the purpose by instantiating the general framework in Section 4. The reader is reminded that the core-point definition has changed to the one in Section 6.2.

7.1 Approximate Bichromatic Close Pair

We now take a short break from clustering to discuss a computational geometry problem which we name the *approximate bichromatic close pair* (aBCP) *problem.* In this problem, we have two disjoint axis-parallel squares c_1, c_2 in $\mathbb{R}^d$. There is a set $S(c_1)$ of points in c_1, and a set $S(c_2)$ of points in c_2. The two sets are subject to insertions and deletions. Let ϵ and ρ be positive real values. We are asked to maintain a *witness pair* of (p_1^*, p_2^*) such that

- It may be an empty pair (i.e., p_1^* and p_2^* are null).
- If it is not empty, we must have $dist(p_1^*, p_2^*) \leq (1 + \rho)\epsilon$.
- The pair must not be empty, if there exist a point $p_1 \in S(c_1)$ and a point $p_2 \in S(c_2)$ such that $dist(p_1, p_2) \leq \epsilon$. Note that the pair does *not* have to be (p_1, p_2) though.

We have at our disposal an emptiness structure (as defined in Section 4.2) on each cell, so that an emptiness query (q, c) with $c = c_1$ or c_2 can be answered with cost $\tilde{O}(\tau)$ for some time function τ. The objective is to minimize the cost of (i) finding an initial witness pair, and (ii) maintaining the pair along with updates in $S(c_1)$ and $S(c_2)$.

LEMMA 3. *For the aBCP problem, an initial witness pair can be found in $\tilde{O}(\tau \cdot \min\{|S(c_1)|, |S(c_2)|\})$ time. After that, the pair can be maintained by $\tilde{O}(\tau)$ amortized time when a point is inserted or deleted in $S(c_1)$ or $S(c_2)$.*

The proof can be found in the appendix.

7.2 Edges in the Grid Graph and aBCP

Returning to ρ-double-approximate DBSCAN clustering, let us recall that in the grid graph $\mathbb{G}$, if there is an edge between core cells c_1 and c_2, then the two cells must be ϵ-close. Such an edge may disappear/re-appear as the core points of $P(c_1)$ and $P(c_2)$ are deleted/inserted. Maintaining this edge can be regarded as an instance of the aBCP problem, where $S(c_1)$ is the set of core points in $P(c_1)$, and $S(c_2)$ is the set of core points in $P(c_2)$—the edge exists if and only if the witness pair is not empty!

We run a thread of the aBCP algorithm of Lemma 3 on every pair of ϵ-close core cells c_1 and c_2. Those threads will be referred to as the *aBCP instances* of c_1 (or c_2). Whenever the edge between c_1 and c_2 (re-)appears, we call *EdgeInsert*(c_1, c_2) of the CC structure; whenever it disappears, we call *EdgeRemove*(c_1, c_2).

7.3 The Core-Status Structure

Given a point q, an *approximate range count query* [10] returns an integer k that falls between $|B(q, \epsilon)|$ and $|B(q, (1 + \rho)\epsilon)|$. The query can be answered in $\tilde{O}(1)$ time by a structure that can be updated in $\tilde{O}(1)$ time per insertion and deletion [16]. Under the relaxed core-point definition of ρ-double-approximation, whether a point $p \in P$ is a core point can be decided directly by issuing such a query with p. If the query returns k, we declare p a core point if and only if $k \geq MinPts$.

Leveraging this fact, next we describe how to explicitly update the core-status of all points in P along with insertions and deletions:

- To insert a point p_{new} in cell c_{new}, we first check whether p_{new} itself is a core point. Remember that the insertion may turn some existing non-core points into core points. To identify such points, we look at each of the $O(1)$ ϵ-close sparse cells c of c_{new}. Simply check all the points $p \in P(c)$ to see if p is currently a core point.
- The deletion of a point p_{old} from cell c_{old} may turn some existing core points into non-core points. Following the same idea in insertion, we look at every ϵ-close sparse cell c of c_{old}, and check all the points $p \in P(c)$ for their current core status.

7.4 GUM

When a point p_{core} (say, in cell c_{core}) has turned into a core point, we check whether c_{core} is already in $\mathbb{V}$:

- If so, simply insert p_{core} into every aBCP instance (Lemma 3) of c_{core}—as explained in Section 7.2, this properly maintains the edges of c_{core}.
- Otherwise, it must hold that $|P(c_{core})| \leq MinPts = O(1)$. We add c_{core} to $\mathbb{V}$. Then, for every ϵ-close core cell c of c_{core}, decide whether to create an edge between c and c_{core} by using the algorithm of Lemma 3 to find an initial witness pair (thereby starting the aBCP instance on c_{core} and c).

Consider now the scenario where a core point p in cell c_{core} has turned into a non-core point. If c_{core} is still a core cell, we remove p from all the aBCP instances of c_{core}. Otherwise, we simply remove c_{core} from $\mathbb{V}$, and destroy all its aBCP instances.

7.5 Performance Guarantees

Utilizing the best CC and emptiness structures under the fully-dynamic scheme, we prove in the appendix our last main result:

THEOREM 4. *For any fixed dimensionality d and fixed constant $\rho > 0$, there is a fully-dynamic ρ-double-approximate DBSCAN algorithm that processes each insertion and deletion in $\tilde{O}(1)$ amortized time, and answers a C-group-by query Q in $\tilde{O}(|Q|)$ time.*

The same update and query efficiency can also be achieved in 2D space for exact DBSCAN.

8. EXPERIMENTS

Section 8.1 describes the setup of our empirical evaluation. Then, Sections 8.2 and 8.3 report the results on semi-dynamic and fully-dynamic algorithms, respectively.

8.1 Setup

Workload. We evaluate a clustering algorithm by its efficiency in processing a *workload*, which is a mixed sequence of updates and queries. Each update or query is collectively referred to as an *operation*. A workload is characterized by several parameters:

- N: the total number of updates.
- $\%_{ins}$: the percentage of insertions. In other words, the workload has $N \cdot \%_{ins}$ insertions, and $N(1 - \%_{ins})$ deletions. This parameter is fixed to 1 in semi-dynamic scenarios.
- f_{qry}: the query frequency, which is an integer controlling how often a C-group-by query is issued.

The production of a workload involves 3 steps, as explained below.

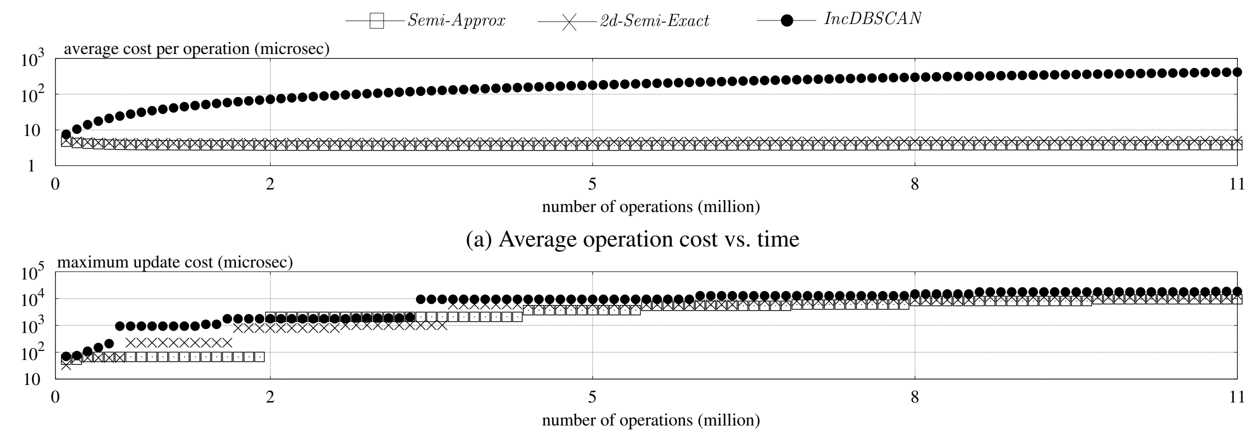
Step 1: Insertions. The sequence of insertions is obtained by first generating a "static dataset" of $I = N \cdot \%_{ins}$ points, and then, randomly permuting these points (i.e., if a point stands at the i-th position, it is the i-th inserted). We generate static datasets whose clusters are the outcome of a "random walk with restart", as was the approach suggested in [10], and will be reviewed shortly. Note that the random permutation mentioned earlier allows the clusters to form up even at an early stage of the workload.

The *data space* is a d-dimensional square that has range $[0, 10^5]$ on each dimension. A static dataset is created using the *seed spreader* technique in [10], which generates around 10 clusters and $0.0001 \cdot I$ noise points as follows. First, place a spreader at a random location p of the data space. At each *time tick*, the spreader adds to the dataset a point that is uniformly distributed in $B(p, 25)$. Whenever the spreader has generated 100 points while stationed at the same p, it is forced to move towards a random direction by a distance of 50. Finally, with probability $10/(0.9999I)$, the spreader "restarts" by jumping to another random location of the data space. Regardless of whether a restart happens, the current time tick finishes, and the next one starts. The spreader works for $0.9999I$ time ticks (thus producing $0.9999I$ points). After that, $0.0001 \cdot I$ random points are added to the dataset as noise.

Step 2: Deletions. First, append to the insertion sequence $D = N - I$ deletion *tokens*, where each token is simply a "place-holder"

parameter	**value**
d	$2, 3, 5, 7$
ϵ	$\boldsymbol{50d}, 100d, 200d, 400d, 800d$
$\%_{ins}$	$\frac{2}{3}, \frac{4}{5}, \mathbf{5/6}, \frac{8}{9}, \frac{10}{11}$
f_{qry}	$0.01N, 0.02N, 0.03N, ..., \boldsymbol{0.1N}$

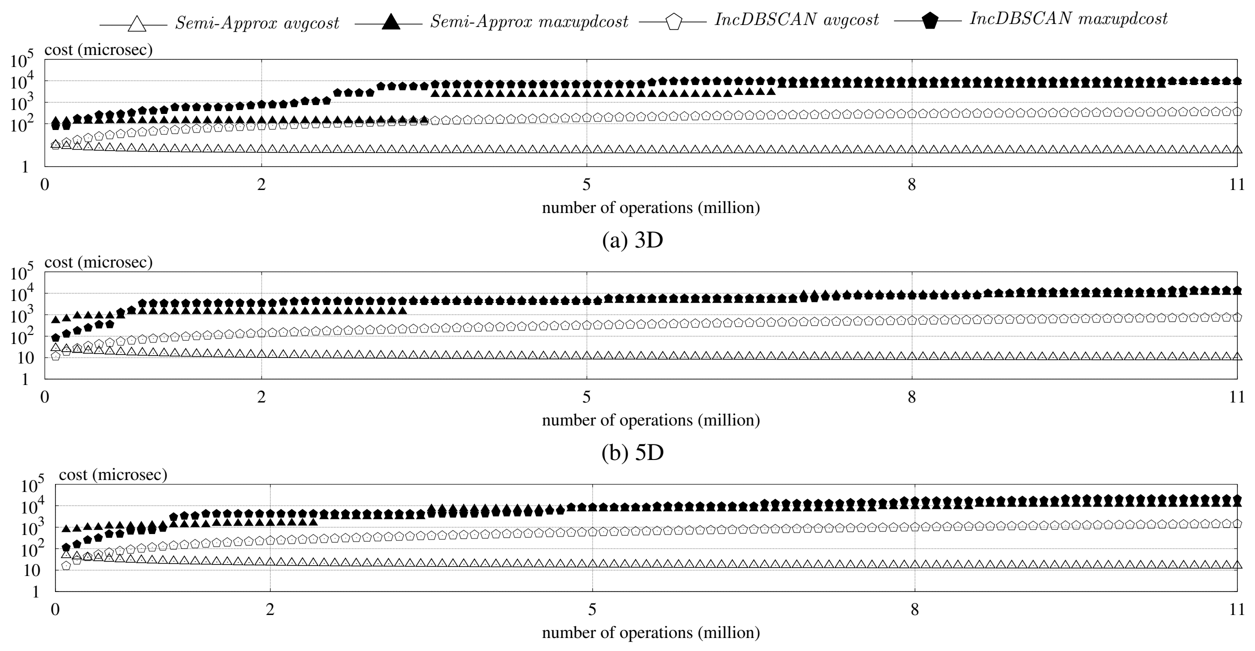
Table 2: Variable parameter values (defaults in bolds)



(a) Average operation cost vs. time

(b) Max update cost vs. time

Figure 8: Performance of semi-dynamic algorithms in 2D



(a) 3D

(b) 5D

(c) 7D

Figure 9: Performance of semi-dynamic algorithms in $d \geq 3$ dimensions

into which we will later fill a concrete point to delete. Then, randomly permute the resulting sequence (which has length N). Check whether the permutation is *bad*, namely, if any of its prefixes has more tokens than insertions. If so, we attempt another random permutation until a *good* one is obtained.

Now we have a good sequence of insertions and deletion tokens. To fill in the tokens, scan down the sequence, and add each inserted point into S, until coming across the first token. Select a random point in S as the one deleted by the token, and then remove the point from S. The scan continues in this fashion until all the tokens have been filled.

Step 3: Queries. We simply insert a C-group-by query after every f_{qry} updates in the sequence. Recall that the query specifies a parameter Q, which is generated as follows. Let S be the set of "alive" points that have been inserted, but not yet deleted before the query. We first decide the value of $|Q|$ by choosing an integer uniformly at random from $[2, 100]$. Then, Q is populated by random sampling $|Q|$ points from S without replacement.

DBSCAN Algorithms. Our experimentation examined:

- *IncDBSCAN [8]*: the state-of-the-art dynamic algorithm for exact DBSCAN, as reviewed in Section 3.
- *2d-Semi-Exact*: our semi-dynamic algorithm in Theorem 1 for exact DBSCAN in 2D space.
- *Semi-Approx*: our semi-dynamic algorithm in Theorem 1 for ρ-approximate DBSCAN in d-dimensional space with $d \geq 2$.
- *2d-Full-Exact*: our fully-dynamic algorithm in Theorem 4 for exact DBSCAN in 2D space.
- *Double-Approx*: our fully-dynamic algorithm in Theorem 4 for ρ-double-approximate DBSCAN in d-dimensional space with $d \geq 2$.

All the algorithms were implemented in C++, and compiled with gcc version 4.8.4.

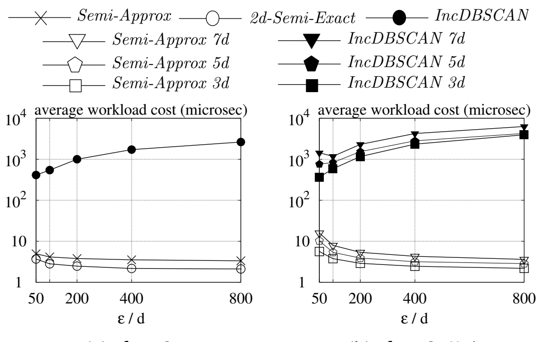


Figure 10: Semi-dynamic performance vs. ϵ

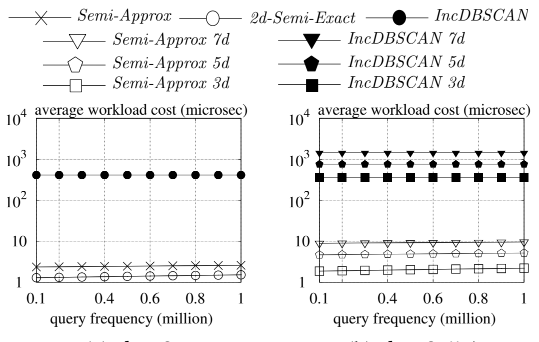


Figure 11: Semi-dynamic performance vs. f_{qry}

Parameters and Machine. We fixed N to 10 million, namely, each workload contains this number of updates. The value of $MinPts$ in all the DBSCAN variants was 10. The value of ρ in the approximate variants was set to 0.001, under which ρ-double-approximate DBSCAN is required to return precisely the same clusters as ρ-approximate DBSCAN.

The other parameters were varied in different experiments. Their values are as shown in Table 2; unless otherwise stated, a parameter was set to its default as shown in bolds. Note that $\%_{ins} = 5/6$ indicates on average 1 deletion every 5 insertions.

Finally, all the experiments were run on a machine equipped with an Intel Core i7-6700 CPU @ 3.40GHz $\times$ 8 and 16GB memory. The operating system was Linux (Ubuntu 14.04.1).

8.2 Semi-Dynamic Results

This subsection will focus on insertion-only workloads. Consider executing an algorithm on such a workload. We define the algorithm's *average cost* as a function of time: $avgcost(t) = \frac{1}{t}\sum_{i=1}^{t} cost[i]$, where $cost[i]$ is the overhead of the algorithm in processing the i-th operation of the workload. Similarly, define the algorithm's *max update cost* as: $maxupdcost(t) = \max_{i=1}^{x} updcost[i]$, where (i) x is the number of updates by the end of the t-th operation, and (ii) $updcost[i]$ is the overhead of the algorithm for the i-th update. Notice that query time is registered in $avgcost$ but not $maxupdcost$.

Focusing on 2D space, Figure 8a plots the average cost of *IncDBSCAN*, *2d-Semi-Exact*, and *Semi-Approx*, whereas Figure 8b plots their max update cost. *2d-Semi-Exact* and *Semi-Approx* finished the workload significantly faster than *IncDBSCAN*, achieving an improvement of two orders of magnitude! Moreover, while the average cost of *IncDBSCAN* deteriorated continuously, the performance of *2d-Semi-Exact* and *Semi-Approx* remained stable throughout the workload. This is expected because *IncDBSCAN* must perform a range query per insertion (see Section 3), which tends to retrieve more data points as time progresses. Our solutions do not suffer from this drawback.

Turning to 3D space—where the competing methods are *IncDBSCAN* and *Semi-Approx*— Figure 9a compares their average cost and max update cost simultaneously. Figures 9b and 9c present the same results for $d = 5$ and 7, respectively. In all dimensionalities, *Semi-Approx* consistently outperformed *IncDBSCAN* by a wide margin even in logarithmic scale.

Interestingly, all the methods exhibited similar behavior when it comes to the $maxupdcost$ metric. We will return to this issue later when we discuss the fully dynamic scenario, where contrasting phenomena will be observed.

We define an algorithm's *average workload cost* as $avgcost(W)$, where W is the total number of operations in the workload of concern. The next experiment demonstrates the effect of ϵ on the cost of cluster maintenance—as discussed in [1,10], an algorithm of density-based clustering should be able to find clusters at different *granularities* of ϵ. Figure 10 shows the average workload cost of each applicable method as a function of ϵ, for $d = 2, 3, 5, 7$, respectively. It is evident that IncDBSCAN became prohibitively expensive as ϵ increases. On the other hand, our solutions actually performed even better for larger ϵ! This is not surprising because a greater ϵ actually *reduces* the number of edges in the grid graph, which in turn leads to substantial cost savings.

We conclude this subsection by giving the average workload cost of all methods as a function of f_{qry} in Figure 11. In general, query cost is negligible compared to update overhead.

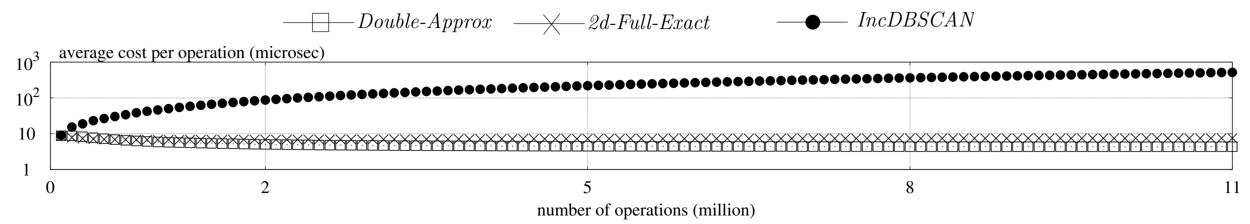
8.3 Fully-Dynamic Results

We now proceed to evaluate the algorithms that can handle both insertions and deletions. Our strategy is similar to that in the previous subsection. *Average cost* and *max update cost* are defined in the same way as before, except that operations/updates obviously also include deletions here.

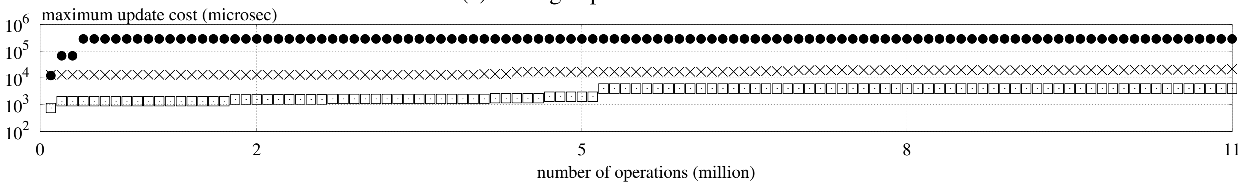
Figure 12 shows the results in experiments corresponding to those in Figure 8, with respect to *IncDBSCAN*, *2d-Full-Exact*, and *Double-Approx*. Similarly, Figure 13 corresponds to Figure 9, with respect to *IncDBSCAN* and *Double-Approx*. As before, our solutions were two orders of magnitude faster than *IncDBSCAN* in average cost. What is new, however, is that they also improved *IncDBSCAN* by nearly 10 times in max update cost as well!

What has triggered the separation in $maxupdcost$? The hardness of deletions! Recall from Section 3 that *IncDBSCAN* requires only one range query (to find the seed objects) in an insertion, whereas in a deletion, it demands multiple—actually perhaps many—such queries to perform BFS. This stands in sharp contrast to *Double-Approx*, which completely gets rid of BFS by novel ideas, in particular, deploying an aBCP algorithm (Lemma 3) to convert cluster maintenance to updating the CCs of the grid graph (which has only $O(n)$ edges). In *all scenarios*, our algorithms ensured processing an update in less than 0.1 seconds! The reader may have noticed that *IncDBSCAN* did not finish the 5D and 7D workloads. Indeed, we terminated it after 3 hours when its deficiencies had become apparent.

Figure 14 presents the results that are the counterparts of Figure 10, confirming that *IncDBSCAN* is essentially inapplicable for large ϵ. Note, again, that this method has no results for $d = 5$ and 7.

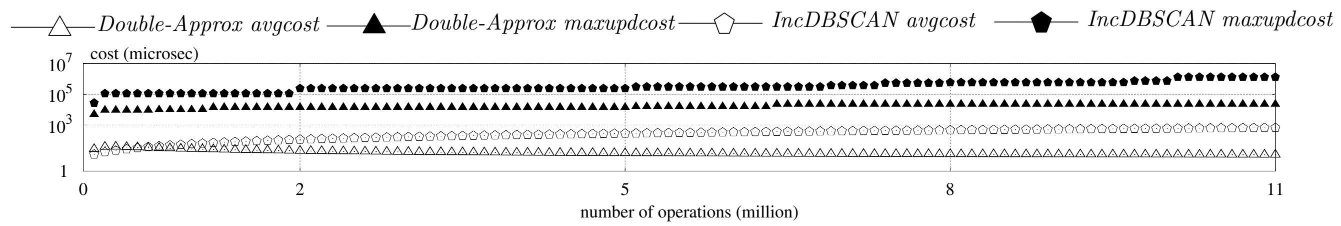


(a) Average operation cost vs. time

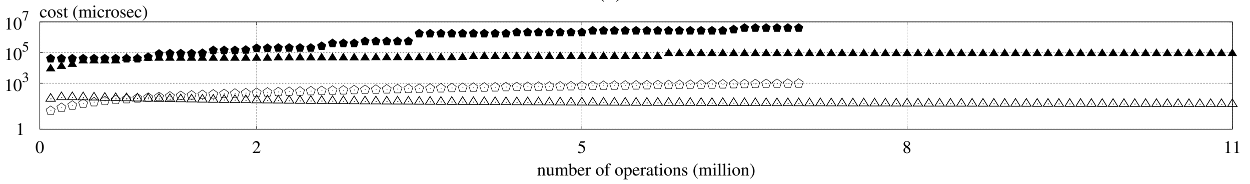


(b) Max update cost vs. time

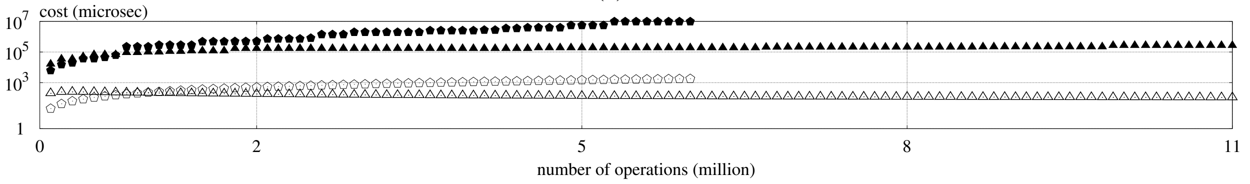
Figure 12: Performance of fully-dynamic algorithms in 2D



(a) 3D



(b) 5D



(c) 7D

Figure 13: Performance of fully-dynamic algorithms in $d \geq 3$ dimensions

The last set of experiments inspected the average workload cost of these algorithms as the insertion percentage increased from $2/3$ to $10/11$. The results are reported in Figure 15. In general, the efficiency of each method improved as insertions accounted for a higher percent of the workload. Our new algorithms were the clear winners in all situations.

9. CONCLUSIONS

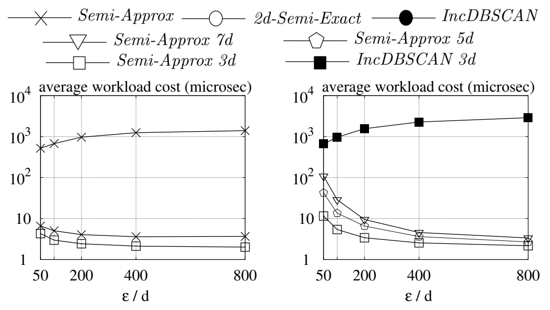
This paper has presented a systematic study on dynamic density based clustering under the theme of DBSCAN. Our findings reveal considerable new insight into the characteristics of the topic, by providing a complete picture of the computational hardness in various update schemes. Perhaps the most surprising result is that ρ-approximate DBSCAN, which was proposed to address the worst-case computational intractability of exact DBSCAN, suffers from the same hardness when both insertions and deletions are allowed. We have also shown how to eliminate the issue elegantly with a tiny relaxation, which has led to the development of ρ-double-approximate DBSCAN. Our algorithmic contributions involve a suite of new algorithms that achieve near-constant update time in cluster maintenance essentially in all the update schemes where this is possible. The practical efficiency of our solutions has also been confirmed with extensive experiments.

10. REFERENCES

[1] M. Ankerst, M. M. Breunig, H. Kriegel, and J. Sander. OPTICS: ordering points to identify the clustering structure. In *SIGMOD*, pages 49–60, 1999.

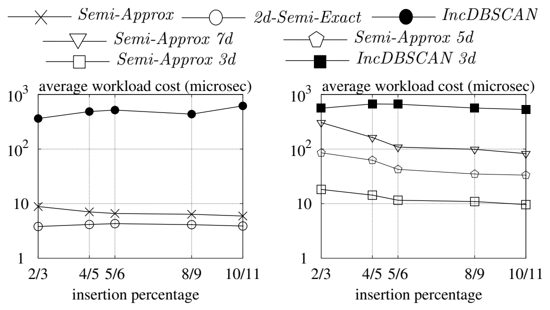
[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *JACM*, 45(6):891–923, 1998.

[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.



(a) $d = 2$ (b) $d = 3, 5, 7$

Figure 14: Fully-dynamic performance vs. ϵ



(a) $d = 2$ (b) $d = 3, 5, 7$

Figure 15: Fully-dynamic performance vs. $\%_{ins}$

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

[5] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *JACM*, 57(3), 2010.

[6] J. Erickson. On the relative complexities of some geometric problems. In *CCCG*, pages 85–90, 1995.

[7] J. Erickson. New lower bounds for Hopcroft's problem. *Disc. & Comp. Geo.*, 16(4):389–418, 1996.

[8] M. Ester, H. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental clustering for mining in a data warehousing environment. In *VLDB*, pages 323–333, 1998.

[9] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *SIGKDD*, pages 226–231, 1996.

[10] J. Gan and Y. Tao. DBSCAN revisited: Mis-claim, un-fixability, and approximation. In *SIGMOD*, pages 519–530, 2015.

[11] A. Gunawan. A faster algorithm for DBSCAN. Master's thesis, Technische University Eindhoven, March 2013.

[12] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[13] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2012.

[14] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *JACM*, 48(4):723–760, 2001.

[15] S. Lühr and M. Lazarescu. Incremental clustering of dynamic data streams using connectivity based representative points. 68(1):1–27, 2009.

[16] D. M. Mount and E. Park. A dynamic data structure for approximate range searching. In *SoCG*, pages 247–256, 2010.

[17] S. Nassar, J. Sander, and C. Cheng. Incremental and effective data summarization for dynamic hierarchical clustering. In *SIGMOD*, pages 467–478, 2004.

[18] S. Nittel, K. T. Leung, and A. Braverman. Scaling clustering algorithms for massive data sets using data streams. In *ICDE*, page 830, 2004.

[19] I. Ntoutsi, A. Zimek, T. Palpanas, P. Kröger, and H. Kriegel. Density-based projected clustering over high dimensional data streams. In *ICDM*, pages 987–998, 2012.

[20] R. G. Pensa, D. Ienco, and R. Meo. Hierarchical co-clustering: off-line and incremental approaches. *Data Min. Knowl. Discov.*, 28(1):31–64, 2014.

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

[22] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Pearson, 2006.

[23] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *JACM*, 22(2):215–225, 1975.

APPENDIX

Proof of Theorem 1

We will prove the theorem first for ρ-approximate DBSCAN, and then for 2D exact DBSCAN.

Implementing the CC-structure as the *union-find* structure of Tarjan [23], we can support both the *EdgeInsert* and *CC-Id* operations in $\tilde{O}(1)$ time amortized. For the emptiness structure of every core cell, we can use the *approximate nearest neighbor* structure of Arya et al. [2], which answers an emptyness query in $\tilde{O}(1)$ time, and can be updated in $\tilde{O}(1)$ time.

Next, we prove that the algorithm processes n insertions in $\tilde{O}(n)$ time, that is, $\tilde{O}(1)$ amortized time per insertion:

- In the core-status structure, Step 1 takes $\tilde{O}(1)$ time per insertion by resorting to a standard dictionary-search structure (e.g., a binary search tree) on the non-empty cells.
- The total cost of Step 2 for the whole update sequence is $O(n)$. To see this, notice that a cell c is involved in this step only if it is an ϵ-close cell of c_{new}. Hence, with respect to the same c_{new}, c can be involved only $MinPts = O(1)$ times (after which c_{new} becomes a core cell, and will not require Step 2 again). As c has $O(1)$ ϵ-close cells, the total number of times that c is involved for *all* the c_{new} is $O(1)$.
- The same reasoning also explains that, the total cost incurred by the execution of the paragraph below Step 2 is $O(n)$ in the whole algorithm.
- In GUM, Steps 1, 1.1, and 1.2 can insert $O(n)$ edges in total, and therefore, entails $\tilde{O}(n)$ cost overall.

The query algorithm performs only $O(|Q|)$ *CC-Id* operations, and therefore, requires only $\tilde{O}(|Q|)$ time.

The above proof holds verbatim also for 2D exact DBSCAN, with the only difference that the structure of [2] should be replaced by the *2D nearest neighbor structure* of Chan [5].

Proof of Lemma 1

Suppose that A is an algorithm settling USEC-LS in $T(n)$ time. We can solve the USEC problem on a set P of n points (each red or blue) using divide and conquer as follows. Divide P using a plane ℓ

orthogonal to dimension 1 into P_1 and P_2 each of which has $n/2$ points. Then, we recursively solve the USEC problem on P_1, and do the same on P_2. If either of these sub-problems returns "yes" (i.e., a red point within distance 1 from a blue point), we return "yes" immediately.

If both sub-problems return "no", we run A twice to solve two instances of USEC-LS. Divide P_1 into the set P_{1red} of red points, and the set P_{1blue} of blue points. Let P_{2red} and P_{2blue} be defined similarly with respect to P_2. The first USEC-LS instance is defined on P_{1red} and P_{2blue}, whereas the second on P_{1blue} and P_{2red}. If either instance returns "yes", we return "yes"; otherwise, we return "no".

Denote by $f(n)$ the running time of our USEC algorithm. The above description shows that

$$f(n) \quad = \quad 2f(n/2) + 2T(n)$$

with $f(n) = O(1)$ when $n = 1$. It is rudimentary to verify that when $T(n) = o(n^{4/3})$, then $f(n) = o(n^{4/3})$.

Proof of Theorem 3

Let G_ϵ be the core graph of $(\epsilon, MinPts)$ exact DBSCAN, and $G_{(1+\rho)\epsilon}$ be the core graph of $((1+\rho)\epsilon, MinPts)$ exact DBSCAN. Also, denote by $G_{\epsilon,\rho}$ the ρ-approximate core graph of ρ-double-approximate DBSCAN with the same ϵ and $MinPts$. A useful observation is that every edge in G_ϵ exists in $G_{\epsilon,\rho}$, and likewise, every edge in $G_{\epsilon,\rho}$ exists in $G_{(1+\rho)\epsilon}$.

Proof of Statement (i). Consider an arbitrary core point p_1 in C_1. Let C be the (only) cluster in $\mathscr{C}$ that contains p_1. Next, we will prove that $C_1 \subseteq C$.

Denote by S_ϵ the CC of G_ϵ containing p_1, and by $S_{\epsilon,\rho}$ the CC of $G_{\epsilon,\rho}$ containing p_1. Clearly, $S_\epsilon \subseteq S_{\epsilon,\rho}$. This means that all the core points of C_1 also belong to C.

Consider now an arbitrary non-core point p_2 in C_1. There must exist a core point $p_3 \in C_1$ such that p_3 is covered by $B(p_2, \epsilon)$. Since $p_3 \in S_\epsilon \subseteq S_{\epsilon,\rho}$, we know that p_2 must also have been assigned to the cluster of $S_{\epsilon,\rho}$, namely, C.

Proof of Statement (ii). Consider an arbitrary core point p_1 in C. Let C_2 be the (only) cluster in $\mathscr{C}_2$ that contains p_1. Next, we will prove that $C \subseteq C_2$.

Denote by $S_{\epsilon,\rho}$ the CC of $G_{\epsilon,\rho}$ containing p_1, and by $S_{(1+\rho)\epsilon}$ the CC of $G_{(1+\rho)\epsilon}$ containing p_1. Clearly, $S_{\epsilon,\rho} \subseteq S_{(1+\rho)\epsilon}$. This means that all the core points of C also belong to C_2.

Consider now an arbitrary non-core point p_2 in C. There must exist a core point $p_3 \in C$ such that p_3 is covered by $B(p_2, (1+\rho)\epsilon)$. Since $p_3 \in S_{\epsilon,\rho} \subseteq S_{(1+\rho)\epsilon}$, we know that p_2 must also have been assigned to the cluster of $S_{(1+\rho)\epsilon}$, namely, C_2.

Proof of Lemma 3

Finding the Initial Pair. Suppose, without loss of generality, that $|S(c_1)| \le |S(c_2)|$. For every point $p_1^* \in S(c_1)$, we run an emptiness query $empty(p_1^*, c_2)$. If the query returns 1 with a proof point p_2^*, we have found a witness pair (p_1^*, p_2^*), and hence, can terminate immediately. The cost is clearly that of $\tilde{O}(|S(c_1)|)$ emptiness queries.

We prove that the algorithm is correct. This is obvious if it finds a pair. Consider, instead, that it does not, in which case we are wrong only if there is a pair $(p_1, p_2) \in S(c_1) \times S(c_2)$ such that $dist(p_1, p_2) \le \epsilon$. However, this means that $empty(p_1, c_2)$ must return 1, thus contradicting the fact that we did not found a pair.

Maintaining the Pair. We store in a list L the points that have been subsequently inserted in $S(c_1) \cup S(c_2)$ (the point ordering can be arbitrary). Each point of L will be *de-listed*—the meaning of which will be clear shortly—once, after which the point is removed from L. Furthermore, we enforce the rule that, if the witness pair is empty, L must be $\emptyset$.

Basic Operation: De-listing. This operation can only be performed when the witness pair is empty—it will attempt to find such a pair by issuing one emptiness query. For this purpose, the operation starts by removing the first point p from L; assume, without loss of generality, that $p \in S(c_1)$. It then issues $empty(p, c_2)$. If the query returns 1 with a proof point $p' \in S(c_2)$, (p, p') is taken as the witness pair. Otherwise, the witness pair remains empty.

Insertion. Consider that a point p has been inserted in $S(c_1)$ (the case with $S(c_2)$ is symmetric). Append p to L. If the witness pair is not empty, we do nothing else. Otherwise, p must now be the only element in L, in which case we perform a de-listing and finish.

Deletion. Consider that a point p has been deleted from $S(c_1)$ (a symmetric algorithm works for $S(c_2)$). Remove from L the entry of p (if found). If the witness pair (p_1^*, p_2^*) is not empty and $p \ne p_1^*$, the deletion does not affect the pair; and we are done. Otherwise, we proceed as follows:

1. Issue $empty(p_2^*, c_1)$. If it returns 1 with a proof point p', set (p', p_2^*) as the new witness pair, and return.
2. Otherwise, do the following until L is empty or the algorithm decides to return:
   2.1 Perform a de-listing.
   2.2 If the de-listing finds a witness pair, return.
3. (Now L is empty) set the witness pair to empty.

Correctness. Our algorithm is always correct if it finds a witness pair Let us look at the case where it does not. This is wrong only if there exists a pair $(p_1, p_2) \in S(c_1) \times S(c_2)$ such that $dist(p_1, p_2) \le \epsilon$. At least one of p_1, p_2 must have been inserted after the initial pair was found. Without loss of generality, assume that p_2 is the one; and if both are, then assume that p_2 was de-listed after p_1.

Consider the moment when our algorithm de-listed p_2 from L. Since p_1 was present in $S(c_1)$, the query $empty(p_2, c_1)$ we issued must have returned a proof point p'. The witness (p', p_2) must have disappeared because p' was deleted. But in this case, our algorithm would immediately issue another $empty(p_2, c_1)$, which (again because p_1 was present in $S(c_1)$) must have returned another proof point. The situation repeats itself, with the consequence that we must be holding a witness pair, thus creating a contradiction.

Efficiency. Clearly, the total number of emptiness queries is at most the number of point insertions and deletions in $S(c_1) \cup S(c_2)$. This concludes the proof of the lemma.

Remark: No Materialization of L. At first glance, it may seem that a point of $S(c_1)$ (or $S(c_2)$) needs to be duplicated in L. Such duplication is space consuming if c_1 is involved in many instances of aBCP simultaneously. In fact, L does not need to be materialized, and instead can be represented using only $O(1)$ memory.

Let us store the points of $S(c_1)$ in a list, sorted by insertion order. We can de-list these points by the sorted order, so that at any moment the points not yet de-listed—namely, those in L—constitute a suffix of the list. The suffix can be identified by remembering only the first point in the suffix, which only needs a single pointer. The same also holds for $S(c_2)$. Thus, two pointers suffice for L. To de-list a point, simply pick the point referenced by either pointer, and then shift the pointer down one position.

It is now evident that, no matter how many instances of aBCP c_1 is involved in, $S(c_1)$ is stored just once, by keeping one pointer for each instance.

Proof of Theorem 4

We prove only the update efficiency because the C-group-by query time is obvious. We will consider first ρ-double-approximate DBSCAN, and then 2D exact DBSCAN.

Implementing the CC-structure as the structure of Holm et al. [14], we can support *EdgeInsert*, *EdgeRemove*, and *CC-Id* all in $\tilde{O}(1)$ amortized time. For the emptiness structure, we can still use the approximate nearest neighbor structure of [2], which answers an emptyness query in $\tilde{O}(1)$ time, and can be updated in $\tilde{O}(1)$ time per insertion and deletion.

Let us now analyze the update cost of the core-status structure. Consider first the insertion of a point p_{new}. Let c_{new} be the cell of p_{new}. In the worst case, we will check all the $O(1)$ ϵ-close sparse cells c of c_{new}. As c has at most $MinPts = O(1)$ points, we need at most $O(1)$ approximate range count queries, whose total overhead is $\tilde{O}(1)$. An analogous argument shows that each deletion entails $\tilde{O}(1)$ time.

To account for the cost of GUM, we analyze how many of the following events may happen during an insertion/deletion on P:

- J_1: the number of aBCP instances created;
- J_2: the number of aBCP instances destroyed;
- K: the number of aBCP insertions/deletions.

An insertion on P can turn at most $O(1)$ points into core points—as mentioned, they must be in c_{new} or the $O(1)$ ϵ-close sparse cells of c_{new}, while each of these cells has at most $MinPts = O(1)$ points. As c has at most $O(1)$ aBCP instances, a new core point in a cell c can trigger at most $O(1)$ new aBCP instances and $O(1)$ aBCP insertions. Similarly, a deletion on P can destroy $O(1)$ aBCP instances and trigger $O(1)$ aBCP core deletions. We thus conclude that J_1, J_2 and K are all bounded by $O(1)$.

The initialization of an aBCP instance takes $\tilde{O}(1)$ time as it requires $O(1)$ emptiness queries by Lemma 3 and the fact that c_{new} has $O(1)$ points. Destroying an aBCP instance also takes only $O(1)$ time because it requires only discarding two pointers (see the remark in the proof of Lemma 3). Furthermore, by Lemma 3, the cost of the aBCP algorithm is proportional to K, now that each emptiness query takes $\tau = \tilde{O}(1)$ time. Thus, J_1, J_2, K all bounded by $O(1)$ indicates that GUM entails $\tilde{O}(1)$ amortized cost in each update.

Finally, the above discussion shows that an update can add/remove $O(1)$ edges of $\mathbb{G}$. Therefore, the cost from the CC-structure is $\tilde{O}(1)$ time amortized per update. We thus conclude the whole proof for ρ-double-approximate DBSCAN.

The above proof holds verbatim also for 2D exact DBSCAN, with the only difference that the structure of [2] should be replaced by the 2D nearest neighbor structure of Chan [5].