IP.LSH.DBSCAN: Integrated Parallel Density-Based Clustering Through Locality-Sensitive Hashing

Amir Keramatian(✉), Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas

Chalmers University of Technology, Gothenburg, Sweden
{amirke,vincenzo.gulisano,ptrianta,tsigas}@chalmers.se

Abstract. Locality-sensitive hashing (LSH) is an established method for fast data indexing and approximate similarity search, with useful parallelism properties. Although indexes and similarity measures are key for data clustering, little has been investigated on the benefits of LSH in the problem. Our proposition is that LSH can be extremely beneficial for parallelizing high-dimensional density-based clustering e.g., DBSCAN, a versatile method able to detect clusters of different shapes and sizes.

We contribute to fill the gap between the advancements in LSH and density-based clustering. We show how approximate DBSCAN clustering can be *fused* into the process of creating an LSH index, and, through parallelization and fine-grained synchronization, also utilize efficiently available computing capacity. The resulting method, IP.LSH.DBSCAN, can support a wide range of applications with diverse distance functions, as well as data distributions and dimensionality. We analyse its properties and evaluate our prototype implementation on a 36-core machine with 2-way hyper threading on massive data-sets with various numbers of dimensions. Our results show that IP.LSH.DBSCAN effectively complements established state-of-the-art methods by up to several orders of magnitude of speed-up on higher dimensional datasets, with tunable high clustering accuracy through LSH parameters.

1 Introduction

Digitalized applications' datasets are getting larger in size and number of features (i.e., dimensions), posing challenges to established data mining methods such as clustering, an unsupervised data mining tool based on similarity measures. Density-based spatial clustering of applications with noise (DBSCAN) [11] is a prominent method to cluster (possibly) noisy data into arbitrary shapes and sizes, without prior knowledge on the number of clusters, using user-defined similarity metrics (i.e., not limited to the Euclidean one). DBSCAN is used in many applications, including LiDAR [25], object detection [20], and GPS route analysis [33]. DBSCAN and some of its variants have been also used to cluster high dimensional data, e.g., medical images [4], text [32], and audio [10].

© Springer Nature Switzerland AG 2022
J. Cano and P. W. Trinder (Eds.): Euro-Par 2022, LNCS 13440, pp. 268–284, 2022.
https://doi.org/10.1007/978-3-031-12597-3_17

The computational complexity of traditional DBSCAN is in the worst-case quadratic in the input size [26], expensive considering attributes of today's datasets. Nonetheless, indexing and spatial data structures facilitating proximity searches can ease DBSCAN's computational complexity, as shown with KD-trees [5], R-trees [15], M-trees [8], and cover trees [6]. Using such structures is sub-optimal in at least three cases, though: (i) skewed data distributions negatively affect their performance [20], (ii) the *dimensionality curse* results in exact spatial data structures based on deterministic space partitioning being slower than linear scan [34], and (iii) such structures only work for a particular metric (e.g., Euclidean distance). In the literature, the major means for enhancing time-efficiency are those of parallelization [2,3,13,24,33] and approximation [12], studied alone or jointly [20,21,33]. However, state-of-the-art methods target Euclidean distance only and suffer from skewed data distributions and the dimensionality curse.

Locality-sensitive hashing (LSH) is an established approach for approximate similarity search. Based on the idea that if two data points are close using a custom similarity measure, then an appropriate hash function can map them to equal values with high probability [1,9,17], LSH can support applications that tolerate *approximate* answers, close to the accurate ones *with high probability*. LSH-based indexing has been successful (and shown to be the best known method [17]) for finding similar items in large high-dimensional data-sets. With our contribution, the `IP.LSH.DBSCAN` algorithm, we show how the processes of approximate density-based clustering and that of creating an LSH indexing structure can be *fused* to boost parallel data analysis. Our novel fused approach can efficiently cope with high dimensional data, skewed distributions, large number of points, and a wide range of distance functions. We evaluate the algorithms analytically and empirically, showing they complement the landscape of established state-of-the-art methods, by offering up to several orders of magnitude speed-up on higher dimensional datasets, with tunable high clustering accuracy.

Organization: Section 2 reviews the preliminaries. Section 3 and Sect. 4 describe and analyse the proposed `IP.LSH.DBSCAN`. Section 5 covers the empirical evaluation. Related work and conclusions are presented in Sect. 6 and Sect. 7, respectively.

2 Preliminaries

System Model and Problem Description. Let `D` denote an *input* set of `N` points, each a multi-dimensional vector from a domain $\mathcal{D}$, with a unique identifier `ID`. `Dist` is a distance function applicable on $\mathcal{D}$'s elements. The *goal* is to partition `D` into an a priori unknown number of disjoint clusters, based on `Dist` and parameters `minPts` and ϵ: `minPts` specifies a lower threshold for the number of neighbors, within radius ϵ, for points to be clustered together.

We aim for an efficient, scalable parallel solution, trading approximations in the clustering with reduced calculations regarding the density criteria, while targeting high accuracy. Our evaluation metric for efficiency is *completion time*. Accuracy is measured with respect to an exact baseline using *rand index* [31]: given two clusterings of the same dataset, the *rand index* is the ratio of the

number of pairs of elements that are either clustered together or separately in both clusterings, to the total number of pairs of elements. Regarding concurrency guarantees, a common consistency goal is that for every parallel execution, there exists a sequential one producing an equivalent result.

We consider multi-core shared-memory systems executing $\mathtt{K}$ threads, supporting `read`, `write` and `read-modify-write` atomic operations, e.g., `CAS` (Compare-And-Swap), available in all contemporary general purpose processors.

Locality Sensitive Hashing (LSH)

The following defines the *sensitivity* of a family of LSH functions [17,23], i.e., the property that, with high probability, nearby points hash to the same value, and faraway ones hash to different values.

Definition 1. *A family of functions* $\mathcal{H} = \{\mathtt{h} : \mathcal{D} \to \mathtt{U}\}$ *is* $(\mathtt{d_1}, \mathtt{d_2}, \mathtt{p_1}, \mathtt{p_2})$*-sensitive for distance function* $\mathtt{Dist}$ *if for any* $\mathtt{p}$ *and* $\mathtt{q}$ *in* $\mathcal{D}$ *the following conditions hold: (i) if* $\mathtt{Dist}(\mathtt{p},\mathtt{q}) \leq \mathtt{d_1}$*, then* $\Pr_{\mathcal{H}}[\mathtt{h}(\mathtt{p}) = \mathtt{h}(\mathtt{q})] \geq \mathtt{p_1}$ *(ii) if* $\mathtt{Dist}(\mathtt{p},\mathtt{q}) \geq \mathtt{d_2}$*, then* $\Pr_{\mathcal{H}}[\mathtt{h}(\mathtt{p}) = \mathtt{h}(\mathtt{q})] \leq \mathtt{p_2}$*. The probabilities are over the random choices in* $\mathcal{H}$*.*

A family $\mathcal{H}$ is useful when $\mathtt{p_1} > \mathtt{p_2}$ and $\mathtt{d_1} < \mathtt{d_2}$. LSH functions can be *combined*, into more effective (in terms of sensitivity) ones, as follows [23]:

Definition 2. *(i) AND-construction: Given a* $(\mathtt{d_1}, \mathtt{d_2}, \mathtt{p_1}, \mathtt{p_2})$*-sensitive family* $\mathcal{H}$ *and an integer* $\mathtt{M}$*, we can create a new LSH family* $\mathcal{G} = \{\mathtt{g} : \mathcal{D} \to \mathtt{U^M}\}$ *by aggregating/concatenating* $\mathtt{M}$ *LSH functions from* $\mathcal{H}$*, where* $\mathtt{g}(\mathtt{p})$ *and* $\mathtt{g}(\mathtt{q})$ *are equal iff* $\mathtt{h_j}(\mathtt{p})$ *and* $\mathtt{h_j}(\mathtt{q})$ *are equal for all* $\mathtt{j} \in \{1, \cdots, \mathtt{M}\}$*, implying* $\mathcal{G}$ *is* $(\mathtt{d_1}, \mathtt{d_2}, \mathtt{p_1}^{\mathtt{M}}, \mathtt{p_2}^{\mathtt{M}})$*-sensitive; (ii) OR-construction: Given an LSH family* $\mathcal{G}$ *and an integer* $\mathtt{L}$*, we can create a new LSH family* $\mathcal{F}$ *where each* $\mathtt{f} \in \mathcal{F}$ *consists of* $\mathtt{L}$ $\mathtt{g_i}$*s chosen independently and uniformly at random from* $\mathcal{G}$*, where* $\mathtt{f}(\mathtt{p})$ *and* $\mathtt{f}(\mathtt{q})$ *are equal iff* $\mathtt{g_j}(\mathtt{p})$ *and* $\mathtt{g_j}(\mathtt{q})$ *are equal for at least one* $\mathtt{j} \in \{1, \cdots, \mathtt{L}\}$*.* $\mathcal{F}$ *is* $(\mathtt{d_1}, \mathtt{d_2}, 1-(1-\mathtt{p_1}^{\mathtt{M}})^{\mathtt{L}}, 1-(1-\mathtt{p_2}^{\mathtt{M}})^{\mathtt{L}})$*-sensitive assuming* $\mathcal{G}$ *is* $(\mathtt{d_1}, \mathtt{d_2}, \mathtt{p_1}^{\mathtt{M}}, \mathtt{p_2}^{\mathtt{M}})$*-sensitive.*

LSH structure: An instance of family $\mathcal{F}$ is implemented as $\mathtt{L}$ hash tables; the $\mathtt{i}$-th table is constructed by hashing each point in $\mathtt{D}$ using $\mathtt{g_i}$ [9,17]. The resulting data structure associates each *bucket* with the values for the keys mapping to its index. LSH families can associate with various distance functions [23].

LSH for Euclidean distance: Let $\mathtt{u}$ be a randomly chosen unit vector in $\mathcal{D}$. A hash function $\mathtt{h_u}(\mathtt{x})$ in such a family is defined as $\lfloor \frac{\mathtt{x}\cdot\mathtt{u}}{\epsilon} \rfloor$, where $\cdot$ is the inner product operation and ϵ is a constant. The family is applicable for any number of dimensions. In a 2-dimensional domain, it is $(\epsilon/2, 2\epsilon, 1/2, 1/3)$-sensitive.

LSH for angular distance: Let $\mathtt{u}$ be a randomly chosen vector in $\mathcal{D}$. A hash function $\mathtt{h_u}(\mathtt{x})$ in such a family is defined as $\mathtt{sgn}(\mathtt{x}\cdot\mathtt{u})$. The family is $(\theta_1, \theta_2, 1-\frac{\theta_1}{\pi}, 1-\frac{\theta_2}{\pi})$-sensitive, where θ_1 and θ_2 are any two angles (in radians) such that $\theta_1 < \theta_2$.

Related Terms and Algorithms DBSCAN: partitions $\mathtt{D}$ into an a priori unknown number of clusters, each consisting of at least one *core-point* (i.e., one with at least `minPts` points in its ϵ-radius neighbourhood) and the points that are *density-reachable* from it. Point q is density-reachable from p, if q is *directly reachable* from p (i.e., in its ϵ-radius neighbourhood) or from another core-point that is density-reachable from p. Non-core-points that are density-reachable from

some core-point are called *border points*, while others are noise [26]. DBSCAN can utilize any distance function e.g., Euclidean, Jaccard, Hamming, angular [23]. Its worst-case time complexity is $\mathcal{O}(\mathtt{N}^2)$, but in certain cases (e.g., Euclidean distance and low-dimensional datasets) its expected complexity lowers to $\mathcal{O}(\mathtt{N}\log\mathtt{N})$, through indexing structures facilitating range queries to find ϵ neighbours [26].

HP-DBSCAN [13]: Highly Parallel DBSCAN is an OpenMP/MPI algorithm, super-imposing a hyper-grid over the input set. It distributes the points to computing units that do local clusterings. Then, the local clusters that need merging are identified and cluster relabeling rules get broadcasted and applied locally.

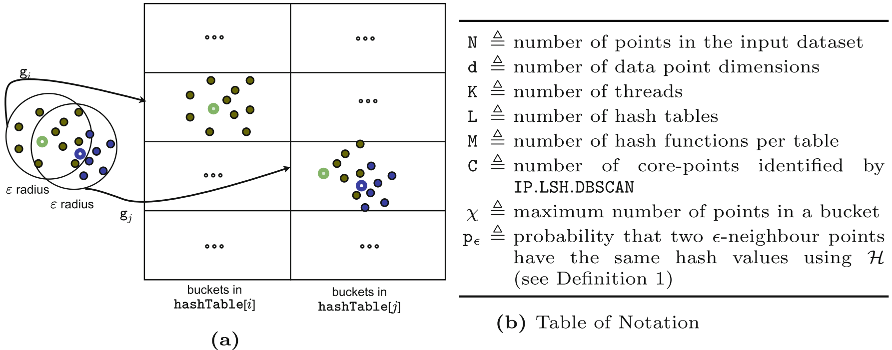
PDS-DBSCAN [24]: An exact parallel version of Euclidean DBSCAN that uses a spatial indexing structure for efficient query ranges. It parallelizes the work by partitioning the points and merging partial clusters, maintained via a *disjoint-set* data structure, also known as *union-find* (a collection of disjoint sets, with the elements in each set connected as a directed tree). Such a data structure facilitates *in-place* `find` and `merge` operations [18] avoiding data copying. Given an element p, `find` retrieves the root (i.e., the *representative*) of the tree in which p resides, while `merge` merges the sets containing two given elements.

Theoretically-Efficient and Practical Parallel DBSCAN [33]: Via a grid-based approach, this algorithm identifies core-cells and utilizes a union-find data structure to merge the neighbouring cells having points within ϵ-radius. It uses spatial indexes to facilitate finding neighbourhood cells and answering range queries.

LSH as index for DBSCAN: LSH's potential led other works ([27,35]) to consider it as a plain means for neighbourhood queries. We refer to them as VLSHDBSCAN.

3 The Proposed IP.LSH.DBSCAN Method

IP.LSH.DBSCAN utilizes the LSH properties, for parallel density-clustering, through efficient fusion of the indexing and clustering formation. On the high level, IP.LSH.DBSCAN hashes each point in D, into multiple hash-tables, in such a way that with a high probability, points within ϵ-distance get hashed to the same bucket at least once across all the tables. E.g., Fig. 1a shows how most nearby points in a subset of D get hashed to the same buckets, in two hash tables. Subsequently, the buckets containing at least `mintPts` elements are examined, to find a set of *candidate core-points* which later will be filtered to identify the real *core-points*, in terms of DBSCAN's definition. In Fig. 1a, the core-points are shown as bold points with a dot inside. The buckets containing core-points are characterized as *core buckets*. Afterwards, with the help of the hash tables, ϵ-neighbour core-points get merged. E.g., the core-bucket in the rightmost hash table in Fig. 1a contains two core-points, indicating the possibility that they



N	≜	number of points in the input dataset
d	≜	number of data point dimensions
K	≜	number of threads
L	≜	number of hash tables
M	≜	number of hash functions per table
C	≜	number of core-points identified by IP.LSH.DBSCAN
χ	≜	maximum number of points in a bucket
$\mathtt{p}_\epsilon$	≜	probability that two ϵ-neighbour points have the same hash values using $\mathcal{H}$ (see Definition 1)

(b) Table of Notation

Fig. 1. 1a shows nearby points get hashed to the same bucket at least once across hash tables, whp. Core-points are the bold ones with a dot inside.

are within each other's ϵ-neighbourhood, in which case they get merged. The merging is done using a forest of union-find data structures, consisting of such core-points, that essentially represent core buckets. As we see later, multiple threads can work in parallel in these steps.

Key Elements and Phases. Similar to an LSH structure (cf. Sect. 2), we utilize L hash tables (hashTable[1], $\cdots$, hashTable[L]), each constructed using M hash functions, chosen according to distance metric Dist and threshold ϵ (see Sect. 2, Sect. 4).

Definition 3. *A bucket in any of the hash tables is called a* candidate core-bucket *if it contains at least* minPts *elements. A* candidate core-point c *in a candidate core-bucket* ccb *is defined to be the closest (using function* Dist*) point in* ccb *to the centroid of all the points in* ccb*; we also say that* c represents ccb*. A candidate core bucket* ccb*, whose candidate core-point* c *has at least* minPts *neighbours within its ϵ-radius in* ccb*, is called a* core-bucket*. A* Core-forest *is a concurrent union-find structure containing core-points representing core buckets.*

Lemma 1. *Given a core bucket, its corresponding core-point* c *is a* true core-point *according to DBSCAN.*

The above follows from the definition of core-point in Definition 3. Next, we present an outline of IP.LSH.DBSCAN's phases, followed by a detailed description of its parallelization and pseudo-code.

In phase I (*hashing and bucketing*), for each i, each point `p` in `D` is hashed using the LSH function $\mathtt{g_i}$ and inserted into `hashTable[i]`. Furthermore, the algorithm keeps track of the buckets containing at least `minPts`, as candidate core buckets. In phase II (*core-point identification*), for each candidate core-bucket, the algorithm identifies a candidate core-point. If at least `minPts` points in a candidate core-bucket fall within the ϵ-neighbourhood of the identified candidate core-point, the latter is identified as a true core-point (see Lemma 1) and inserted into the core-forest as a singleton. In phase III (*merge-task identification and processing*), the algorithm inspects each core-bucket and creates and performs a *merge task* for each pair of core-points that are within each others' ϵ-neighbourhood. Hence, the elements in the core-forest start forming sets according to the merge tasks. In phase IV (*data labeling*), the algorithm labels the points: a core-point gets assigned the same clustering label as all the other core-points with which it forms a set in the core-forest. A border point (i.e., a non-core-point located in the ϵ-radius of a core-point) is labeled the same as a corresponding core-point, and all the other points are considered noise.

Parallelism and Algorithmic Implementation. We here present the parallelization in IP.LSH.DBSCAN (Algorithm 1), targeting speed-up by distributing

Algorithm 1. Outline of IP.LSH.DBSCAN

1: *Input*: dataset `D`, threshold `minPts`, radius ϵ, nr. of hash tables `L`, nr. of hash functions per table `M`, metric `Dist`, nr of threads `K`; *Output*: a clustering label for each point in `D`
2: let `D` be logically partitioned into `S` mutually disjoint batches
3: `hashTable[1]`, $\cdots$,`hashTable[L]` are hash tables supporting concurrent insertions and traversals
4: `candidateCoreBuckets` and `coreBuckets` are empty sets supporting concurrent operations
5: let `hashTasks` be a $\mathtt{S} \times \mathtt{L}$ boolean array initialized to `false`, indicating the status of hash tasks corresponding to the Cartesian product of `S` batches and `L` hash tables
6: let $\mathcal{G} = \{\mathtt{g} : \mathtt{S} \rightarrow \mathtt{U}^{\mathtt{M}}\}$ be an LSH family suitable for metric `Dist`, and let $\mathtt{g_1}, \cdots, \mathtt{g_L}$ be hash functions chosen independently and uniformly at random from $\mathcal{G}$ (Definition 2)
7: **for all** threads **in parallel do**
8:   **phase I:** *hashing and bucketing*
9:     **while** the running thread can book a `task` from `hashTasks` **do**
10:       **for each** point `p` in `task.batch` **do**
11:         let `i` be index of the `hashTable` associated with `task`
12:         $\mathtt{hashTable[i].insert(key} = \mathtt{g_i(p), value} = \mathtt{ptr(p))}$
13:         $\mathtt{bucket{=}hashTable[i].getBucket(key} = \mathtt{g_i(p))}$
14:         **if** $\mathtt{bucket.size()} \geq \mathtt{minPts}$ **then** `candidateCoreBuckets.insert(ptr(bucket))`
15:   **phase II:** *core-point identification* (starts when all threads reach here)
16:     **for each** `ccb` in `candidateCoreBuckets` **do**
17:       let `c` be the closest point in `ccb` to `ccb` points' centroid
18:       **if** $|\{\mathtt{q} \in \mathtt{ccb}$ such that $\mathtt{Dist(c, q)}\}| \geq \mathtt{minPts}$ **then**
19:         $\mathtt{c} \rightarrow$ `corePoint` := `TRUE` and insert `c` into the core-forest
20:         `coreBuckets.insert(ccb)`
21:   **phase III:** *merge-task identification and processing* (starts when all threads reach here)
22:     **while** `cb` := `coreBuckets.pop()` **do**
23:       let `core` be the core-point associated with `cb`
24:       **for** core-point $\mathtt{c} \in \mathtt{cb}$ such that $\mathtt{Dist(core, c)} \leq \epsilon$ **do** `merge(core, c)`
25:   **phase IV:** *data labeling* (starts when a thread reaches here)
26:     **for each** core bucket `cb` **do**
27:       let `core` be the core-point associated with `cb`
28:       **for each** non-labeled point `p` in `cb` **do**
29:         **if** $\mathtt{p} \rightarrow$ `corePoint` **then** `p.idx` = `findRoot(p).ID`
30:         **else** `p.idx` = `findRoot(core).ID`

the work among K threads. We also aim at in-place operations on data points and buckets (i.e., without creating additional copies), hence work with pointers to the relevant data points and buckets in the data structures.

Phase I (*hashing and bucketing*): To parallelize the hashing of the input dataset D into L hash tables, we (logically) partition D into S mutually disjoint batches. Consecutively, we have $S \times L$ *hash tasks*, corresponding to the Cartesian product of the hash tables and the data batches. Threads can *book* a hash task and thus share the workload through hashTasks, which is a boolean $S \times L$ array containing a *status* for each task, initially false. A thread in this phase scans the elements of hashTasks, and if it finds a non-booked task, it tries to *atomically book* the task (e.g., via a CAS to change the status from false to true). The thread that books a hash task $ht_{b,t}$ hashes each data point p in batch b into hashTable[t] using hash function g_t. Particularly, for each point p, a key-value pair consisting of the hashed value of p and a pointer to p is inserted in hashTable[t]. As entries get inserted into the hash tables, pointers to buckets with at least minPts points are added to candidateCoreBuckets. Since the threads operate concurrently, we use hash tables supporting concurrent insertions and traversals. Algorithm 1 l.8–l.14 summarizes Phase I.

Phase II (*core-point identification*): Here the threads identify core-buckets and core-points. Each thread atomically pops a candidate core bucket ccb from candidateCoreBuckets and identifies the closest point to the centroid of the points in ccb, considering it as a candidate core-point, ccp. If there are at least minPts points in ccb within ϵ-radius of ccp, then ccp and ccb become core-point and core-bucket, respectively, and ccp is inserted in the core-forest and the ccb in the coreBucekts set. This phase, shown in Algorithm 1 l.15–l.20, is finished when candidateCoreBuckets becomes empty.

Phase III (*merge-task identification and processing*): The threads here identify and perform merge tasks. For each core-bucket cb that a thread successfully books from the set coreBuckets, the thread merges the sets corresponding to the associated core-point with cb and any other core-point in cb within ϵ distance. For merging, the algorithm uses an established concurrent implementation for disjoint-sets, with *linearizable* and *wait-free* (i.e., the effects of concurrent operations are consistent with the sequential specification, while the threads can make progress independently of each other) find and merge, proposed in [18]. This phase, shown in Algorithm 1 l.21–l.24, completes when coreBucekts is empty.

Phase IV (*data labeling*): Each non-labeled core-point in a core-bucket gets its clustering label after its root ID in the core-forest. All other non-labeled points in a core-bucket are labeled with the root ID of the associated core-point. The process, shown in Algorithm 1 l.25–l.30, is performed concurrently for all core-buckets.

4 Analysis

This section analytically studies IP.LSH.DBSCAN's accuracy, safety and completeness properties, and completion time. We provide sketch proofs to save space, but formal proofs can be found in [19]. Figure 1b summarizes the notations.

Accuracy Analysis

Let $\mathtt{p}_\epsilon$ be the probability that two given points with maximum distance ϵ have the same hash value using $\mathcal{H}$ (see Definition 1). Lemma 1 shows that any point identified as a core-point by IP.LSH.DBSCAN is a true core-point in terms of DBSCAN. The following Lemma provides a lower bound on the probability that IP.LSH.DBSCAN identifies any DBSCAN core-point.

Lemma 2. *Let* c *be a DBSCAN's core-point. The probability that* IP.LSH.DBSCAN *also identifies* c *as a core-points is at least* $1-\left(1-\frac{1}{\chi}\mathtt{p}_\epsilon^{\mathtt{M}(\mathtt{minPts}-1)}\right)^{\mathtt{L}}$*, where* χ *denotes the maximum number of points in any bucket.*

Proof. (sketch) First note that with M hash functions per table (i.e., the AND construction in Definition 2), the probability that two given points with maximum distance ϵ *collide* into the same hash bucket in a fixed hash table is $\mathtt{p}_\epsilon^{\mathtt{M}}$. The probability that c gets identified as a core-point in a fixed hash table is at leaset $\frac{1}{\chi}\mathtt{p}_\epsilon^{\mathtt{M}(\mathtt{minPts}-1)}$ because at least minPts-1 ϵ-neighbours of c must get hashed to b, and c should be the closest point to the centroid of the points in b. Finally, the probability that c gets identified as a core-point in at least one hash table is computed as the complement of the probability that c does not get identified as a core-point in any hash table.

Lemma 2 shows that the probability of identifying any DBSCAN core-point can be made arbirtarily close to 1 by choosing sufficiently large L.

Lemma 3. *Let* $\mathtt{c}_1$ *and* $\mathtt{c}_2$ *be two core-points identified by* IP.LSH.DBSCAN*.*

1. *If* $\mathtt{Dist}(\mathtt{c}_1, \mathtt{c}_2) > \epsilon$*, then* IP.LSH.DBSCAN *does not merge* $\mathtt{c}_1$ *and* $\mathtt{c}_2$*.*
2. *If* $\mathtt{Dist}(\mathtt{c}_1, \mathtt{c}_2) \leq \epsilon$*, then the probability that* IP.LSH.DBSCAN *merges* $\mathtt{c}_1$ *and* $\mathtt{c}_2$ *is at least* $2\mathtt{p}_\epsilon^{\mathtt{M}} - \mathtt{p}_\epsilon^{2\mathtt{M}}$*.*

Proof. (sketch) 1. follows directly from IP.LSH.DBSCAN's algorithmic description (see Algorithm 1 l.24). 2. follows from calculating the probability that $\mathtt{c}_2$ hashes into the same bucket in which $\mathtt{c}_1$ is the representative, and vice-versa.

Safety and Completeness Properties

At the end of phase IV, each set in the core-forest maintained by IP.LSH.DBSCAN contains a subset of density-reachable core-points (as defined in Sect. 2). Two disjoint-set structures ds_1, ds_2 are *equivalent* if there is a one-to-one correspondence between ds_1's and ds_2's sets. The following lemma implies that the outcomes of single-threaded and concurrent executions of IP.LSH.DBSCAN are equivalent.

Lemma 4. *Any pair of concurrent executions of* IP.LSH.DBSCAN *that use the same hash functions, produce equivalent core-forests at the end of phase IV.*

Proof. (sketch) Considering a fixed instance of the problem, any concurrent execution of IP.LSH.DBSCAN identifies the same set of core-points and core-buckets with the same hash functions, hence performing the same set of merge operations. As the concurrent executions of merge operations are linearizable (see Sect. 3)

and merge operation satisfies the associative and commutative properties, the resulting sets in the core-forest are identical for any concurrent execution.

It is worth noting that *border points* (i.e., non-core-points within the vicinity of multiple core-points) can be assigned to any of the neighbouring clusters. The original DBSCAN [11] exhibits the same behaviour as well.

Completion Time Analysis

Lemma 5. *[adapted from Theorem 2 in [18]] The probability that each* findRoot *and each* merge *perform* $\mathcal{O}(\log \mathtt{C})$ *steps is at least* $1 - \frac{1}{\mathtt{C}}$*, where* C *is the number of identified core-points by* IP.LSH.DBSCAN*.*

Corollary 1. *The expected asymptotic time complexity of each* findRoot *and each* merge *is* $\mathcal{O}(\log \mathtt{C})$*.*

Lemma 6. *The expected completion time of phase I is* $\mathcal{O}(\frac{\mathtt{LMNd}}{\mathtt{K}})$*; phase II and phase III is bounded by* $\mathcal{O}(\frac{\mathtt{LN}\log \mathtt{C}}{\mathtt{K}})$*; phase IV is* $\mathcal{O}(\frac{\mathtt{N}\log \mathtt{C}}{\mathtt{K}})$*.*

Theorem 1. *The expected completion time of* IP.LSH.DBSCAN *is* $\mathcal{O}(\frac{\mathtt{LMNd}+\mathtt{LN}\log \mathtt{C}}{\mathtt{K}})$*.*

Theorem 1 is derived by taking the asymptotically dominant terms in Lemma 6. It shows IP.LSH.DBSCAN's expected completion time is inversely proportional to K and grows linearly in N, d, L, and M. In common cases where C is much smaller than N, the expected completion time is $\mathcal{O}(\frac{\mathtt{LMNd}}{\mathtt{K}})$; In the worst-case, where C is $\mathcal{O}(\mathtt{N})$, the expected completion time is $\mathcal{O}(\frac{\mathtt{LMNd}+\mathtt{LN}\log \mathtt{N}}{\mathtt{K}})$. For this to happen, for instance, ϵ and minPts need to be extremely small and L be extremely large. As the density parameters of DBSCAN are chosen to detect meaningful clusters, such choices for ϵ and minPts are in practice avoided.

On the memory use of IP.LSH.DBSCAN: The memory footprint of IP.LSH.DBSCAN is proportional to $(\mathtt{LN}+\mathtt{Nd})$, as it simply needs only one copy of each data point and pointers in the hash tables and this dominates the overhead of all other utilized data structures. Further, in-place operations ensure that data is not copied and transferred unnecessarily, which is a significant factor regarding efficiency. In Sect. 5, the effect of these properties is discussed.

Choice of L and M: For an LSH structure, a plot representing the probability of points hashing into the same bucket as a function of their distance resembles an inverse s-curve (x- and y-axis being the distance, and the probability of hashing to the same bucket, resp.), starting at 1 for the points with distance 0, declining with a significant slope around some threshold, and approaching 0 for far apart points. Choices of L and M directly influence the shape of the associated curve, particularly the location of the threshold and the sharpness of the decline [23]. It is worth noting that steeper declines generally result in more accurate LSH structures at the expense of larger L and M values. Consequently, in IP.LSH.DBSCAN, L and M must be determined to (i) set the location of the threshold at ϵ, and (ii) balance the trade-off between the steepness of the decline and the completion time. In Sect. 5, we study a range of L and M values and their implications on the trade-off between IP.LSH.DBSCAN's accuracy and completion time.

5 Evaluation

We conduct an extensive evaluation of IP.LSH.DBSCAN, comparing with the established state-of-the-art algorithms. Our implementation is publicly available [19]. Complementing Theorem 1, we measure the execution latency with varying number of threads (K), data points (N), dimensions (d), hash tables (L), and hash functions per table (M). We use varying ϵ values, as well as Euclidean and angular distances. We measure IP.LSH.DBSCAN's accuracy against the exact DBSCAN (hence also the baseline state-of-the-art algorithms) using rand index.

Setup: We implemented IP.LSH.DBSCAN in C++, using POSIX threads and the concurrent hash table of Intel's threading building blocks library (TBB). We used a c5.18xlarg AWS machine, with 144 GB of memory and two Intel Xeon Platinum 8124M CPUs, with 36 two-way hyper-threaded cores [33] in total.

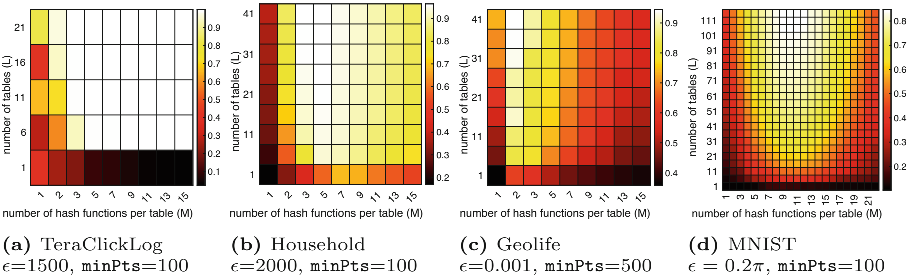
Tested Methods: In addition to IP.LSH.DBSCAN, we benchmark PDSDBSCAN [24], HPDBSCAN [13], and the exact algorithm in [33], for which we use the label TEDBSCAN (Theoretically-Efficient and Practical Parallel DBSCAN). As the approximate algorithms in [33] are generally not faster than their exact counterpart (see Fig. 9 and discussion on p. 13 in [33]), we consider their efficiency represented by the exact TEDBSCAN. We also benchmark VLSHDBSCAN, our version of a single-thread DBSCAN that uses LSH indexing, as we did not find open implementations for [27,35]. Benchmarking VLSHDBSCAN allows a comparison regarding the approximation degree, as well as the efficiency induced by IP.LSH.DBSCAN's "fused" approach. Section 2 covers the aforementioned algorithms.

Evaluation Data and Parameters
Following common practices [13,28,33], we use datasets with different characteristics. We use varying ϵ but fixed minPts, as the sensitivity on the latter is significantly smaller [28]. We also follow earlier works' common practice to abort any execution that exceeds a certain bound, here 9×10^5 sec (more than 24 h). We introduce the datasets and the chosen values for ϵ and minPts as well as the choices for L and M, based on the corresponding discussion in Sect. 4 and also the literature guidelines (e.g., [23] and the reference therein). The *default* ϵ values are shown in *italics.*

TeraClickLog [33]: Each point in this dataset corresponds to a display ad by Criteo with 13 integer and 26 categorical features. We use a subset with over 67 million points, free from missing features. Like [33], we only consider the integer features, and we choose ϵ from $\{\mathit{1500}, 3000, 6000, 12000\}$ and minPts 100.

Household [12]: This is an electricity consumption dataset with over two million points, each being seven-dimensional after removing the date and time features (as suggested in [12]). Following the practice in [12,33], we scale each feature to [0,10000] interval and choose ϵ from $\{1500, \mathit{2000}, 2500, 3000\}$ and minPts 100.

GeoLife [36]: From this GPS trajectory dataset, we choose ca 1.5 million points as selected in [20], containing latitude and longitude with a *highly skewed* distribution. We choose ϵ from $\{\mathit{0.001}, 0.002, 0.004, 0.008\}$ and minPts 500, like [20].



(a) TeraClickLog ϵ=1500, minPts=100 **(b)** Household ϵ=2000, minPts=100 **(c)** Geolife ϵ=0.001, minPts=500 **(d)** MNIST $\epsilon = 0.2\pi$, minPts=100

Fig. 2. Visualizing the rand index accuracy of IP.LSH.DBSCAN as a function of L and M

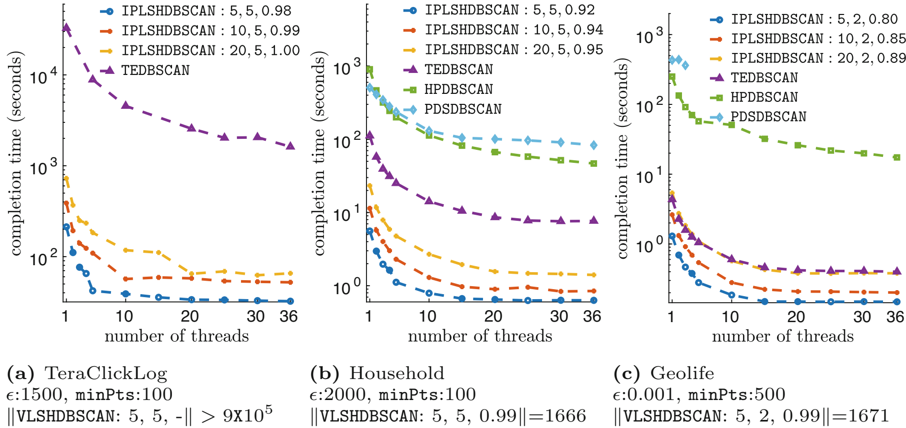
MNIST: This dataset contains 70000 28×28-pixel hand-written and labelled 0–9 digits [22]. We treat each record as a 784-dimensional data point, normalizing each point to have a unit length (similar to [30]). We utilize the angular distance. Following [29], we choose ϵ from $\{0.18\pi, 0.19\pi, \mathit{0.20\pi}, 0.21\pi\}$ and minPts 100.

The heat-maps in Fig. 2a–Fig. 2d visualize IP.LSH.DBSCAN's rand index accuracy as a function of L and M. For TeraClickLog (Fig. 2a), {L = 5, M = 5}, {L = 10, M = 5}, and {L = 20, M = 5} give 0.98, 0.99, and 1 accuracies, respectively. For Household (Fig. 2b), {L = 5, M = 5}, {L = 10, M = 5}, and {L = 20, M = 5} give 0.92, 0.94, and 0.95 accuracies, respectively. For GeoLife (Fig. 2c), {L = 5, M = 2}, {L = 10, M = 2}, and {L = 20, M = 2} give 0.8, 0.85, and 0.89 accuracies, respectively. For (Fig. 2d) dataset, {L = 58, M = 9}, {L = 116, M = 9}, and {L = 230, M = 9} give 0.77, 0.85, and 0.89 accuracy, respectively, computed with respect to the actual labels.

Experiments for the Euclidean Distance

Completion Time with Varying K: Figure 3a, Fig. 3b, and Fig. 3c show the completion time of IP.LSH.DBSCAN and other methods with varying K on TeraClickLog, Household, and Geolife datasets, respectively. PDSDBSCAN runs out of memory on TeraClickLog for all K and on GeoLife for $K \geq 4$, and none of HPDBSCAN's executions terminate within the 9×10^5 sec threshold. For the reference, in Fig. 3, the completion time of single-thread VLSHDBSCAN is provided as a caption for each dataset, except for TeraClickLog as its completion time exceeds 9×10^5 sec. The results indicate the benefits of parallelization for work-load distribution in IP.LSH.DBSCAN, also validating that IP.LSH.DBSCAN's completion time behavior is linear with respect to L, as shown in Theorem 1. For higher dimensionality, challenging the state-of-the-art algorithms, IP.LSH.DBSCAN's completion time is several orders of magnitude shorter.

Completion Time with Varying ϵ: The left Y-axes in Fig. 4a, Fig. 4b, and Fig. 4c show the completion time of IP.LSH.DBSCAN and other tested methods using 36 cores with varying ϵ values on TeraClickLog, Household, and Geolife datasets, respectively. PDSDBSCAN crashes by running out of memory on TeraClickLog and GeoLife for all ϵ, and none of HPDBSCAN's executions terminate within the 9×10^5 sec threshold. The right Y-axes in Fig. 4a, Fig. 4b, and Fig. 4c show

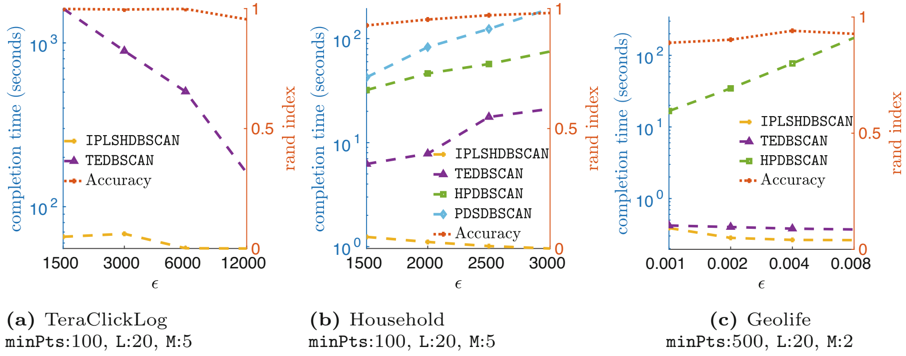


(a) TeraClickLog
ϵ:1500, `minPts`:100
||`VLSHDBSCAN`: 5, 5, -|| > 9X10^5

(b) Household
ϵ:2000, `minPts`:100
||`VLSHDBSCAN`: 5, 5, 0.99||=1666

(c) Geolife
ϵ:0.001, `minPts`:500
||`VLSHDBSCAN`: 5, 2, 0.99||=1671

Fig. 3. *Completion time with varying* `K`. *The comma-separated values corresponding to* `IP.LSH.DBSCAN` *and* `VLSHDBSCAN` *show* `L`, `M`, *and the rand index accuracy, respectively. PDSDBSCAN crashes by running out of memory in 3a for all* `K` *and for* `K` ≥ 4 *in 3c. In 3a no HPDBSCAN executions terminate within the* 9×10^5*-sec threshold.*



(a) TeraClickLog
`minPts`:100, `L`:20, `M`:5

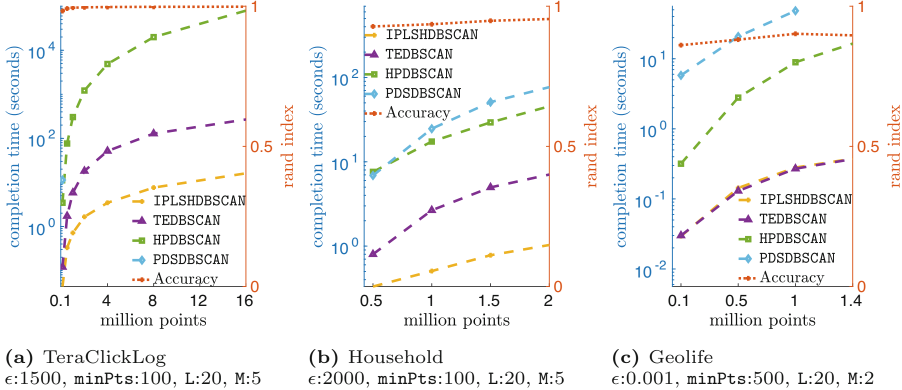
(b) Household
`minPts`:100, `L`:20, `M`:5

(c) Geolife
`minPts`:500, `L`:20, `M`:2

Fig. 4. *Completion time using varying* ϵ *with 36 cores. PDSDBSCAN crashes by running out of memory in 4a and 4c for all* ϵ. *None of HPDBSCAN's executions terminate within the* 9×10^5 *sec threshold in 4a. Right Y-axes show* `IP.LSH.DBSCAN`*'s rand index.*

the corresponding rand index accuracy of `IP.LSH.DBSCAN`. The results show that in general the completion time of `IP.LSH.DBSCAN` decreases by increasing ϵ. Intuitively, hashing points into larger buckets results in lower `merge` workload. Similar benefits, although with higher completion times, are seen for TEDBSCAN. On the other hand, as the results show, completion time of many classical methods (such as HPDBSCAN and PDSDBSCAN) increases with increasing ϵ.

Completion Time with Varying `N`: The left Y-axes in Fig. 5a, Fig. 5b, and Fig. 5c show the completion time of the bench-marked methods using 36 cores on varying size subsets of TeraClickLog, Household, and Geolife datasets, respectively.

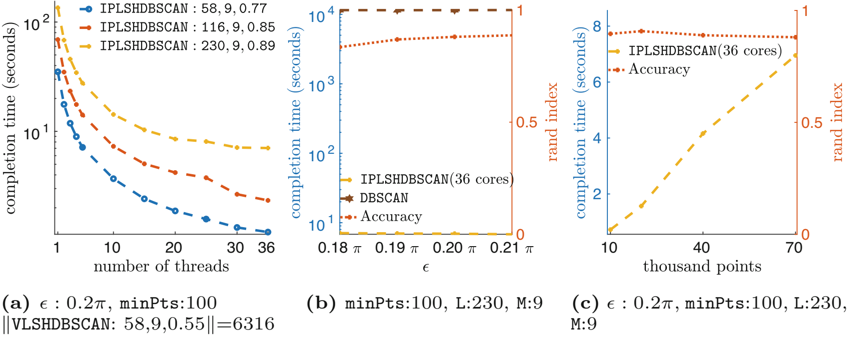


(a) TeraClickLog
ϵ:1500, `minPts`:100, `L`:20, `M`:5

(b) Household
ϵ:2000, `minPts`:100, `L`:20, `M`:5

(c) Geolife
ϵ:0.001, `minPts`:500, `L`:20, `M`:2

Fig. 5. *Completion time with varying* `N` *using 36 cores. PDSDBSCAN runs out of memory in 5a with* `N` $>$ *0.1 million points and with* `N` $>$ *1 million points in 5c.* `IP.LSH.DBSCAN` *and* TEDBSCAN *coincide in 5c. Right Y-axes show* `IP.LSH.DBSCAN`*'s rand index.*



(a) ϵ : 0.2π, `minPts`:100
‖`VLSHDBSCAN`: 58,9,0.55‖=6316

(b) `minPts`:100, `L`:230, `M`:9

(c) ϵ : 0.2π, `minPts`:100, `L`:230, `M`:9

Fig. 6. *MNIST results with the angular distance (only* `IP.LSH.DBSCAN`, `VLSHDBSCAN`, *DBSCAN support the angular distance). 6a shows* `IP.LSH.DBSCAN`*'s completion time with varying* `K`*. The left Y-axes in 6b and 6c respectively show* `IP.LSH.DBSCAN`*'s completion time with varying* ϵ *and* `N`*, using 36 cores. The right Y-axes in 6b and 6c show the associated accuracy, computed with respect to the actual labels.*

PDSDBSCAN runs out of memory on TeraClickLog subsets with `N` $>$ 0.1 million points and GeoLife subsets with `N` $>$ 1 million points. The results empirically validate that completion time of `IP.LSH.DBSCAN` exhibits a linear growth in the number of data points, complementing Theorem 1. The right Y-axes in Fig. 5a, Fig. 5b, and Fig. 5c show the corresponding rand index accuracy of `IP.LSH.DBSCAN`.

Experiments for the Angular Distance. For significantly high number of dimensions, as a side-effect of dimensionality curse, the Euclidean distance among all pairs of points is almost equal [23]. To overcome this, we use angular distance. We only study methods that support such a distance: `IP.LSH.DBSCAN`,

VLSHDBSCAN, and DBSCAN. Here accuracy is calculated against the actual labels. Figure 6a shows completion time with varying K. The left Y-axes in Fig. 6b and Fig. 6c respectively show completion time with varying ϵ and N, using 36 cores. The right Y-axes in Fig. 6b and Fig. 6c show the associated accuracies. Note IP.LSH.DBSCAN's completion time is more than 4 orders of magnitude faster than a sequential DBSCAN and more than 3 orders of magnitude faster than VLSHDBSCAN. Here, too the results align and complement Theorem 1's analysis.

Discussion of Results. IP.LSH.DBSCAN targets high dimensional, memory-efficient clustering for various distance measures. IP.LSH.DBSCAN's completion time is several orders of magnitude shorter than state-of-the-art counterparts, while ensuring approximation with tunable accuracy and showing efficiency for lower dimensional data too. In practice, IP.LSH.DBSCAN's completion time exhibits a linear behaviour with respect to the number of points, even for skewed data distributions and varying density parameters. The benefits of IP.LSH.DBSCAN with respect to other algorithms increase with increasing data dimensionality. IP.LSH.DBSCAN scales both with the size of the input and its dimensionality.

6 Other Related Work

Having compared IP.LSH.DBSCAN with representative state-of-the-art related algorithms in Sect. 5, we focus on related work considering approximation. Gan et al. and Wang et al. in [12,33] proposed approximate DBSCAN clustering for low-dimensional Euclidean distance, with $\mathcal{O}(\mathtt{N}^2)$ complexity if $2^{\mathtt{d}} > \mathtt{N}$ [7]. The PARMA-CC [20] approach is also suitable only for low-dimensional data. VLSHDBSCAN [27,35] uses LSH for neighbourhood queries. However, the LSH index creation in IP.LSH.DBSCAN is embedded into the dynamics of the clusters formation. IP.LSH.DBSCAN iterates over buckets and it applies merges on core-points that represent bigger entities, drastically reducing the search complexity. Also, IP.LSH.DBSCAN is a concurrent rather than a single-thread algorithm. Esfandiari et al. [10] propose an almost linear approximate DBSCAN that identifies core-points by mapping points into hyper-cubes and counting the points in them. It uses LSH to find and merge nearby core-points. IP.LSH.DBSCAN integrates core-point identification and merging in one structure altogether, leading to better efficiency and flexibility in leveraging the desired distance function.

7 Conclusions

IP.LSH.DBSCAN proposes a simple and efficient method combining insights on DBSCAN with features of LSH. It offers approximation with tunable accuracy and high parallelism, avoiding the exponential growth of the search effort with the number of data dimensions, thus scaling both with the size of the input and its dimensionality, and dealing with high skewness in a memory-efficient way. We expect IP.LSH.DBSCAN will support applications in the evolving landscape of

cyberphysical system data pipelines to aggregate information from large, high-dimensional, highly-skewed data sets [14,16]. We also expect that this methodology can be used for partitioning data for other types of graph processing and as such this direction is worth investigating as extension of `IP.LSH.DBSCAN`.

Acknowledgements and Data Availability Statement. Work supported by SSF grant "FiC" (GMT14-0032); VR grants "HARE" (2016-03800), "Relaxed Concurrent Data Structure Semantics for Scalable Data Processing" (2021-05443), "EPITOME" (2021-05424); Chalmers AoA frameworks Energy and Production, proj. INDEED, and WP "Scalability, Big Data and AI". The source code generated for the current study is available in the Figshare repository https://doi.org/10.6084/m9.figshare.19991786 [19].

References

1. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Commun. ACM **51**(1), 117–122 (2008). https://doi.org/10.1145/1327452.1327494
2. Andrade, G., Ramos, G.S., Madeira, D., Oliveira, R.S., Ferreira, R., Rocha, L.: G-DBSCAN: a GPU accelerated algorithm for density-based clustering. In: International Conference on Computational Science. ICCS 2013. Procedia Computer Science, vol. 18, pp. 369–378. Elsevier (2013). https://doi.org/10.1016/j.procs.2013.05.200
3. Arlia, D., Coppola, M.: Experiments in parallel clustering with DBSCAN. In: Sakellariou, R., Gurd, J., Freeman, L., Keane, J. (eds.) Euro-Par 2001. LNCS, vol. 2150, pp. 326–331. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44681-8_46
4. Baselice, F., Coppolino, L., D'Antonio, S., Ferraioli, G., Sgaglione, L.: A DBSCAN based approach for jointly segment and classify brain MR images. In: 37th International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC 2015, pp. 2993–2996. IEEE (2015). https://doi.org/10.1109/EMBC.2015.7319021
5. Bentley, J.L.: K-d trees for semidynamic point sets. In: 6th Symposium on Computational Geometry, pp. 187–197. ACM (1990). https://doi.org/10.1145/98524.98564
6. Beygelzimer, A., Kakade, S., Langford, J.: Cover trees for nearest neighbor. In: 23rd Conference on Machine Learning. ICML 2006, pp. 97–104. ACM (2006). https://doi.org/10.1145/1143844.1143857
7. Chen, Y., Tang, S., Bouguila, N., Wang, C., Du, J., Li, H.: A fast clustering algorithm based on pruning unnecessary distance computations in DBSCAN for high-dimensional data. Pattern Recogn. **83**, 375–387 (2018). https://doi.org/10.1016/j.patcog.2018.05.030
8. Ciaccia, P., Patella, M., Zezula, P.: M-tree: an efficient access method for similarity search in metric spaces. In: VLDB 1997, 23rd International Conference on Very Large Data Bases, pp. 426–435. M. Kaufmann (1997). http://www.vldb.org/conf/1997/P426.PDF
9. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: 20th Symposium on Computational Geometry. SCG 2004, pp. 253–262. ACM (2004). http://doi.acm.org/10.1145/997817.997857

10. Esfandiari, H., Mirrokni, V.S., Zhong, P.: Almost linear time density level set estimation via DBSCAN. In: 35th AAAI Conference on Artificial Intelligence AAAI 2021, pp. 7349–7357. AAAI Press (2021). https://ojs.aaai.org/index.php/AAAI/article/view/16902
11. Ester, M., Kriegel, H., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: 2nd Conference on Knowledge Discovery and Data Mining (KDD-96), pp. 226–231. AAAI Press (1996). http://www.aaai.org/Library/KDD/1996/kdd96-037.php
12. Gan, J., Tao, Y.: On the hardness and approximation of Euclidean DBSCAN. ACM Trans. Database Syst. **42**(3), 14:1–14:45 (2017). https://doi.org/10.1145/3083897
13. Götz, M., Bodenstein, C., Riedel, M.: HPDBSCAN: highly parallel DBSCAN. In: Workshop on Machine Learning in High-Performance Computing Environments, MLHPC 2015, pp. 2:1–2:10. ACM (2015). https://doi.org/10.1145/2834892.2834894
14. Gulisano, V., Nikolakopoulos, Y., Cederman, D., Papatriantafilou, M., Tsigas, P.: Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types. ACM Trans. Parallel Comput. (TOPC) **4**(2), 1–28 (2017)
15. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: 1984 SIGMOD International Conference on Management of Data, pp. 47–57. ACM Press (1984). https://doi.org/10.1145/602259.602266
16. Havers, B., Duvignau, R., Najdataei, H., Gulisano, V., Koppisetty, A.C., Papatriantafilou, M.: Driven: a framework for efficient data retrieval and clustering in vehicular networks. In: 35th International Conference on Data Engineering (ICDE), pp. 1850–1861. IEEE (2019)
17. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: 30th ACM Symposium on the Theory of Computation, pp. 604–613. ACM (1998). https://doi.org/10.1145/276698.276876
18. Jayanti, S.V., Tarjan, R.E.: A randomized concurrent algorithm for disjoint set union. In: 2016 ACM Symposium on Principles of Distributed Computating ACM (2016). https://doi.org/10.1145/2933057.2933108
19. Keramatian, A., Gulisano, V., Papatriantafilou, M., Tsigas, P.: Artifact and instructions to generate experimental results for the Euro-Par 2022 paper: "IP.LSH.DBSCAN: Integrated Parallel Density-Based Clustering Through Locality-Sensitive Hashing" (2022). https://doi.org/10.6084/m9.figshare.19991786
20. Keramatian, A., Gulisano, V., Papatriantafilou, M., Tsigas, P.: PARMA-CC: parallel multiphase approximate cluster combining. In: 21st International Conference on Distributed Computing and Networking, pp. 20:1–20:10. ACM (2020). https://doi.org/10.1145/3369740.3369785
21. Keramatian, A., Gulisano, V., Papatriantafilou, M., Tsigas, P.: MAD-C: multi-stage approximate distributed cluster-combining for obstacle detection and localization. J. Parallel Distrib. Comput. **147**, 248–267 (2021)
22. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998). https://doi.org/10.1109/5.726791
23. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets, 2nd edn. Cambridge University Press (2014). http://www.mmds.org/
24. Patwary, M.M.A., Palsetia, D., Agrawal, A., Liao, W., Manne, F., Choudhary, A.N.: A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In: SC Conference on High Performance Computing Networking, Storage and Analysis, SC 2012, p. 62. IEEE/ACM (2012). https://doi.org/10.1109/SC.2012.9

25. Rusu, R.B., Cousins, S.: 3D is here: point cloud library (PCL). In: IEEE International Conference on Robotics and Automation, ICRA. IEEE (2011). https://doi.org/10.1109/ICRA.2011.5980567
26. Schubert, E., Sander, J., Ester, M., Kriegel, H.P., Xu, X.: DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. ACM Trans. Database Syst. **42**(3), 19:1–19:21 (2017). http://doi.acm.org/10.1145/3068335
27. Shiqiu, Y., Qingsheng, Z.: DBSCAN clustering algorithm based on locality sensitive hashing. J. Phys. Conf. Series **1314**, 012177 (2019). https://doi.org/10.1088/1742-6596/1314/1/012177
28. Song, H., Lee, J.: RP-DBSCAN: a superfast parallel DBSCAN algorithm based on random partitioning. In: 2018 SIGMOD International Conference on Management of Data, pp. 1173–1187. ACM (2018). https://doi.org/10.1145/3183713.3196887
29. Starczewski, A., Goetzen, P., Er, M.J.: A new method for automatic determining DBSCAN parameters. J. Artif. Intell. Soft Comput. Res. **10**(3), 209–221 (2020). https://doi.org/10.2478/jaiscr-2020-0014
30. Sundaram, N., et al.: Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. VLDB Endow. **6**(14), 1930–1941 (2013). http://www.vldb.org/pvldb/vol6/p1930-sundaram.pdf
31. Wagner, S., Wagner, D.: Comparing clusterings- an overview (2007)
32. Wang, X., Zhang, L., Zhang, X., Xie, K.: Application of improved DBSCAN clustering algorithm on industrial fault text data. In: 18th IEEE International Conference on Industrial Information, INDIN, pp. 461–468. IEEE (2020). https://doi.org/10.1109/INDIN45582.2020.9442093
33. Wang, Y., Gu, Y., Shun, J.: Theoretically-efficient and practical parallel DBSCAN. In: 2020 SIGMOD International Conference on Management of Data, pp. 2555–2571. ACM (2020). https://doi.org/10.1145/3318464.3380582
34. Weber, R., Schek, H., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: VLDB 1998, 24rd International Conference on Very Large Data Bases, pp. 194–205. M. Kaufmann (1998). http://www.vldb.org/conf/1998/p194.pdf
35. Wu, Y.P., Guo, J.J., Zhang, X.J.: A linear DBSCAN algorithm based on LSH. In: International Conference on ML and Cybernetics, vol. 5, pp. 2608–2614 (2007). https://doi.org/10.1109/ICMLC.2007.4370588
36. Zheng, Y., Xie, X., Ma, W.: GeoLife: a collaborative social networking service among user, location and trajectory. IEEE Data Eng. Bull. **33**(2), 32–39 (2010). http://sites.computer.org/debull/A10june/geolife.pdf