IP.LSH.DBSCAN: Integrated parallel density-based clustering by locality-sensitive hashing

Discrete Applied Mathematics 382 (2026) 183–196

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

IP.LSH.DBSCAN: Integrated parallel density-based clustering by locality-sensitive hashing

Amir Keramatian, Vincenzo Gulisano, Marina Papatriantafilou *, Philippas Tsigas

Department of Computer Science and Engineering, Chalmers University of Technology and Gothenburg University, Sweden

ARTICLE INFO

Article history:
Received 31 March 2025
Received in revised form 15 November 2025
Accepted 19 November 2025

Dataset link: https://doi.org/10.6084/m9.figshare.19991786

Keywords:
Density-based clustering
Similarity-based clustering
Approximation algorithms
Data summarization
High-dimension data analytics

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 multifaceted benefits of LSH in the problem. 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 algorithm, IP.LSH.DBSCAN, described in this article, can support a wide range of applications with diverse distance functions, as well as data distributions and dimensionality. We analyse the algorithm's asymptotic completion time and provide an open-source prototype implementation. We also conduct a detailed evaluation measuring latency and accuracy metrics of IP.LSH.DBSCAN, on a 36-core machine with 2-way hyper threading on massive data-sets with various numbers of dimensions. The analysis and the empirical study of IP.LSH.DBSCAN show how it complements 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.

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 [36] point-cloud analysis, object detection [24], and GPS route analysis [44]. DBSCAN and some of its variants have been also used to cluster high dimensional data, e.g., medical images [3], text [48], and audio [10].

The computational complexity of traditional DBSCAN is in the worst-case quadratic in the input size [37], 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 [4], R-trees [18], M-trees [7], and cover trees [5]. However, the majority of commonly used indexing data structures (including KD-trees [4], Octrees [9], and voxel grids [35]), depending on the data distribution, dimensionality and implementation, could exhibit a super-linear growth in size with respect to the size of the dataset. Furthermore, the cost of common queries and operations can become

* Corresponding author.
E-mail address: ptrianta@chalmers.se (M. Papatriantafilou).

https://doi.org/10.1016/j.dam.2025.11.047

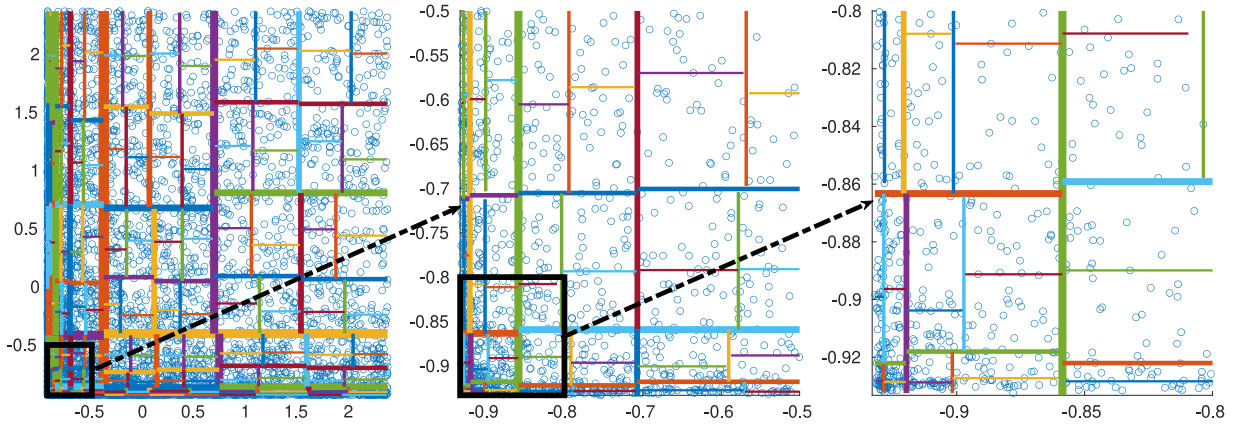


Fig. 1. The structure of a KD-tree on a highly skewed two-dimensional data distribution. The distribution peaks at the bottom left corner, where the figure zooms in twice. The majority of points are located in small-sized cells. Therefore, most range and neighbourhood queries require to examine a substantial portion of the points, rendering the efficiency of the structure similar to a brute-force approach.

expensive (e.g., super linear in the number of data points). Skewed data distributions can also critically challenge the performance of such data structures [15,20,28,46].

For example, Fig. 1 visualizes the structure of a balanced KD-tree (where each partition divides the data into two equal parts) on a highly skewed two-dimensional data distribution. The distribution peaks at the bottom left corner, where the figure zooms in twice. As the figure illustrates, the cell sizes vary vastly across the structure, but most points are located in very small-sized (in width and height) cells. Therefore, most range and neighbourhood queries require examining a substantial portion of the points, making the efficiency of the KD-tree structure comparable to a brute-force approach. Moreover, since data in different applications can exhibit varying levels of spatio-temporal locality, specialized methods tailored to different levels of spatio-temporal locality are necessary for improved efficiency. Additionally, one of the most detrimental effects of high-dimensional spaces is the concentration effect of L_p norms [1], which "*is the surprising characteristic of all points in a high dimensional space to be at almost the same distance to all other points in that space*" [12]. Being a special case of L_p norms when $p = 2$, the Euclidean distance is one of the most commonly used distance measures, but its usability is limited in use-cases concerning high dimensions because of the concentration effect of L_p norms. Furthermore, the computational complexity of classical neighbourhood query methods, commonly used in data processing, increases exponentially with the number of dimensions [37]. Quoting from [37] about the classical indexing structures: "*Similar to R*-trees and most index structures, grid-based approaches tend to not scale well with increasing number of dimensions, due to the curse of dimensionality [...]: the number of possible grid cells grows exponentially with the dimensionality*". It has been shown that angular distance is a more suitable alternative for high dimensional data as it better reflects the contrast between near and far points [25,27]. However, most challenges of efficient parallel clustering using the angular distance (and other alternative distance measures) remain unaddressed as most scientific efforts on parallel data clustering have only focused on utilization of the Euclidean distance, e.g., [13,14,16,32,33,44].

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 [2,8,21], 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 [21]) 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 `IP.LSH.DBSCAN` analytically and empirically, showing how it complements 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.[1]

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

[1] The article extends the homonymous work presented at the 28th International Conference on Parallel and Distributed Computing, in Glasgow, UK, August 2022. The present manuscript extends the description of the contributions providing also a more detailed context, includes the proofs of the analysis, and an extended discussion of the evaluation and insights into potential impact and applications.

2. Preliminaries

2.1. 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 neighbours, 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* [43]: 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 K threads, supporting the following atomic operations: read, write and read-modify-write, e.g., CAS (Compare-And-Swap), available in all contemporary general-purpose processors.

2.2. Locality Sensitive Hashing (LSH)

The following defines the *sensitivity* of a family of LSH functions [21,27], 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 $\mathcal{H} = \{h : \mathcal{D} \to U\}$ is (d_1, d_2, p_1, p_2)-sensitive for distance function Dist if for any p and q in $\mathcal{D}$ the following conditions hold: (i) if $Dist(p, q) \leq d_1$, then $Pr_{\mathcal{H}}[h(p) = h(q)] \geq p_1$ (ii) if $Dist(p, q) \geq d_2$, then $Pr_{\mathcal{H}}[h(p) = h(q)] \leq p_2$. The probabilities are over the random choices in $\mathcal{H}$.

A family $\mathcal{H}$ is useful when $p_1 > p_2$ and $d_1 < d_2$.

2.2.1. LSH amplification

LSH functions can be *combined*, into more effective (in terms of sensitivity) ones, as follows [27].

AND-*construction.* This construction creates a new LSH family where each member is composed by independently choosing M arbitrary functions in $\mathcal{H}$, namely h_1, h_2, ..., h_M. With the new hash function, p and q get hashed to the same bucket if $h_i(p)$ is equal to $h_i(q)$ for all $i \in \{1, \ldots, M\}$. The resulting LSH family is $(d_1, d_2, {p_1}^M, {p_2}^M)$-sensitive as the probabilities multiply as h_is are chosen independently and randomly.

OR-*construction.* Similar to *AND*-construction, *OR*-construction creates a new LSH family hash function where each member is composed by independently choosing L arbitrary functions in $\mathcal{H}$, namely h_1, h_2, ..., h_L. Nevertheless, in this case, p and q get hashed to the same bucket if $h_i(p)$ is equal to $h_i(q)$ for at least one i. The resulting LSH family is $(d_1, d_2, 1-(1-p_1)^L, 1-(1-p_2)^L)$-sensitive.

Cascading *the two constructions.* The aforementioned constructions can be arbitrarily cascaded in order to create new LSH families with different sensitivities, suitable for different applications. Fig. 2 shows the probability of two given points p and q being hashed to the same bucket as a function of their angular distance using a variety of cascading, where the AND construction is applied first, and then cascaded with an OR construction.

2.2.2. LSH structure

An instance of family $\mathcal{F}$ is implemented as L hash tables; the ith table is constructed by hashing each point in D using h_i [8,21]. 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 [27].

LSH for Euclidean distance: Let u be a randomly chosen unit vector in $\mathcal{D}$. A hash function $h_u(x)$ in such a family is defined as $\lfloor \frac{x \cdot u}{\epsilon} \rfloor$, 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 u be a randomly chosen vector in $\mathcal{D}$. A hash function $h_u(x)$ in such a family is defined as $sgn(x \cdot u)$. 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 $\theta_1 < \theta_2$.

2.3. Related terms and algorithms

DBSCAN: partitions 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*,

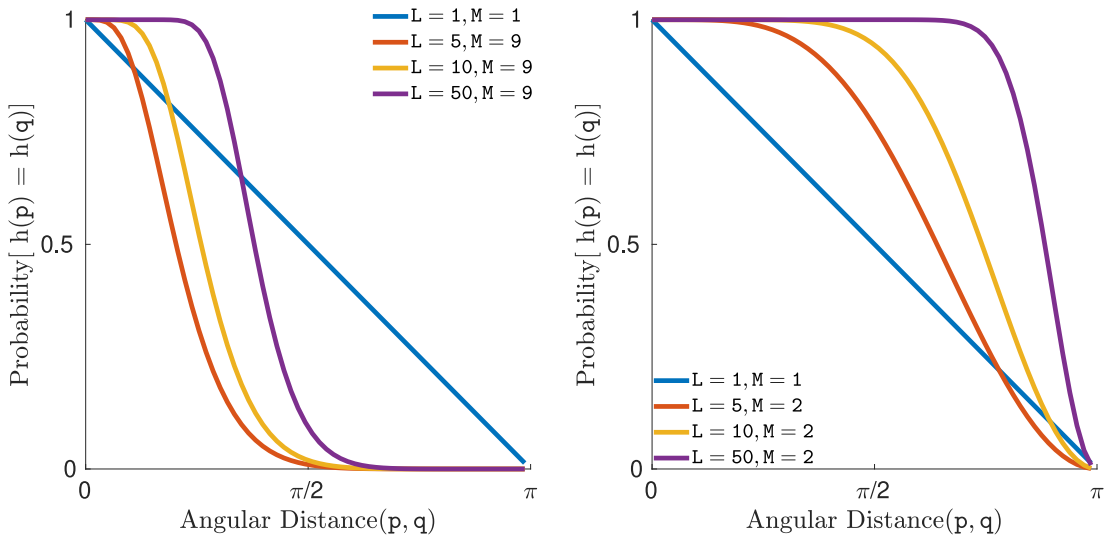


Fig. 2. Illustration of the probability that two given points, p and q, are hashed into the same bucket as a function of their angular distance. The figure depicts how this probability varies when M hash functions are first amplified using the AND construction and then cascaded with the OR construction across L hash tables.

while others are noise [37]. DBSCAN can utilize any distance function e.g., Euclidean, Jaccard, Hamming, angular [27]. Its worst-case time complexity is $\mathcal{O}(N^2)$, but in certain cases (e.g., Euclidean distance and low-dimensional datasets) its expected complexity lowers to $\mathcal{O}(N \log N)$, through indexing structures facilitating range queries to find ϵ neighbours [37].
HP-DBSCAN: [16] 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 relabelling rules get broadcasted and applied locally.
PDS-DBSCAN: [33] 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 [22] 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: [44] 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 [38,49] 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 dataset `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. As an example, the illustration in Fig. 3 shows that numerous nearby points in a subset of `D` are hashed into the same buckets across 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. 3, 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. In the example illustrated in Fig. 3, the core-bucket in the rightmost hash table contains two core-points. This suggests that they may be within each other's ϵ-neighbourhood—if confirmed, they will be merged. The merging is performed using a forest of union-find data structures composed of core points, which essentially represent core buckets. As we will see later, multiple threads can operate in parallel during these steps.

3.1. Key elements and phases

Similar to an LSH structure (cf. Section 2), we use `L` hash tables—`hashTable[1]`, . . ., `hashTable[L]`—each constructed using `M` hash functions, selected based on the distance metric `Dist` and threshold ϵ (see Sections 2 and 4).

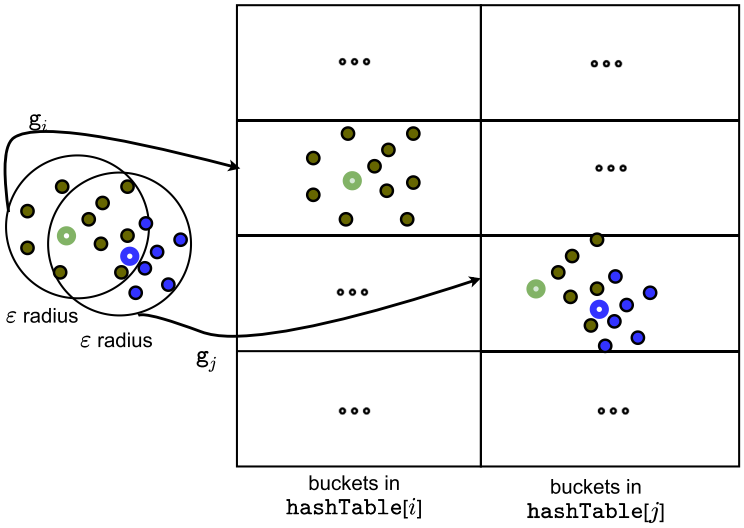


Fig. 3. The diagram illustrates nearby points getting hashed to the same bucket at least once across hash tables, whp. Core-points are the bold ones, with a dot inside.

Definition 2. 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 2. 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 `hashTable[i]`, every point `p` in dataset `D` is hashed using the LSH function h_i and inserted into `hashTable[i]`. The algorithm also 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 lie within each others' ϵ-neighbourhood. As a result, the elements in the core-forest begin to form sets according to the performed merge tasks. In phase IV (*data labeling*), the algorithm assigns clustering labels to the points. Each core-point receives the same label as other core-points within its set in the core-forest. Border points — non-core-points within the ϵ-neighbourhood of a core-points — inherit the corresponding core-point's label. All remaining points are considered noise.

3.2. Parallelism and algorithmic implementation

We here present the parallelization in `IP.LSH.DBSCAN` (Alg. 1), targeting speed-up by distributing 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. This results in a total of $S \times L$ *hash tasks*, corresponding to the Cartesian product of hash tables and data batches. Threads share the workload by *booking* hash tasks through `hashTasks`, which is a boolean $S \times L$ array that maintains the *status* of each hash task, all initially set to `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 succeeds to book a hash task $ht_{b,t}$ hashes each data point `p` in batch `b` into `hashTable[t]` using the hash function h_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. Alg. 1 l. 9-l. 15 describesPhase 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

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
2: Output: a clustering label for each point in D
3: let D be logically partitioned into S mutually disjoint batches
4: let hashTable[1], ···, hashTable[L] be hash tables supporting concurrent insertions and traversals
5: let candidateCoreBuckets and coreBuckets be initially empty sets that support concurrent operations
6: let hashTasks be a S × L boolean array initialized to false, indicating the status of hash tasks corresponding to the Cartesian product of S batches
   and L hash tables
7: let H = {h : S → U^M} be an LSH family suitable for metric Dist, and let h_1, ···, h_L be hash functions chosen independently and uniformly at random
   from H (according to the definitions in Section 2.2.1)
8: for all threads in parallel do
9:   phase I: hashing and bucketing
10:    while the running thread can book a task from hashTasks do
11:      for each point p in task.batch do
12:        let i be index of the hashTable associated with task
13:        hashTable[i].insert(key = h_i(p), value = ptr(p))
14:        bucket=hashTable[i].getBucket(key = h_i(p))
15:        if bucket.size() ≥ minPts then candidateCoreBuckets.insert(ptr(bucket))
16:  phase II: core-point identification (starts when all threads reach here)
17:    for each ccb in candidateCoreBuckets do
18:      let c be the closest point in ccb to ccb points' centroid
19:      if len({q ∈ ccb such that Dist(c, q)}) ≥ minPts then
20:        c→ corePoint := TRUE and insert c into the core-forest
21:        coreBuckets.insert(ccb)
22:  phase III: merge-task identification and processing (starts when all threads reach here)
23:    while cb := coreBuckets.pop() do
24:      let core be the core-point associated with cb
25:      for core-point c ∈ cb such that Dist(core, c) ≤ ϵ do merge(core, c)
26:  phase IV: data labeling (starts when a thread reaches here)
27:    for each core bucket cb do
28:      let core be the core-point associated with cb
29:      for each non-labeled point p in cb do
30:        if p→ corePoint then p.idx = findRoot(p).ID
31:        else p.idx = findRoot(core).ID
```

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 Alg. 1 l. 16-l. 21, 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 [22]. This phase, shown in Alg. 1 l. 22-l. 25, completes when coreBucekts is empty.
Phase IV (*data labeling*): Each non-labelled core-point in a core-bucket gets its clustering label after its root ID in the core-forest. All other non-labelled points in a core-bucket are labelled with the root ID of the associated core-point. The process, shown in Alg. 1 l. 26-l. 31, 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. Fig. 4 summarizes the notations.

4.1. Accuracy analysis

Let p_ϵ be the probability that two given points with maximum distance ϵ have the same hash value using $\mathcal{H}$ (see Definition 1). Lemma 1 in the previous section showed 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* $1 - \left(1 - \frac{1}{\chi} p_\epsilon^{M(\text{minPts}-1)}\right)^L$, *where* χ *denotes the maximum number of points in any bucket.*

Proof. We first calculate the probability that c gets identified as a core-point in a fixed hash table. Afterwards, we find the probability that c gets identified as a core-point in at least one of the L hash tables.

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

Fig. 4. Table of notation.

Consider the ith hash table, and let $\mathtt{b}$ be the bucket to which $\mathtt{c}$ gets hashed. Let $\mathtt{X}_i$ be an indicator random variable showing whether $\mathtt{c}$ get identified as a core-point in the ith hash table. For $\mathtt{X}_i$ to be one, two conditions must hold. ($\mathtt{A}$): at least $\mathtt{minPts}$-1 ϵ-neighbours of $\mathtt{c}$ are hashed to $\mathtt{b}$, and ($\mathtt{B}$): $\mathtt{c}$ is the closest point to the centroid of the points in $\mathtt{b}$. Regarding $\mathtt{A}$, we take into consideration that $\mathtt{c}$ has at least $\mathtt{minPts}$-1 ϵ-neighbours, each of which can independently get hashed into $\mathtt{b}$ with probability $\mathtt{p}_\epsilon^{\mathtt{M}}$. Regarding $\mathtt{B}$, $\mathtt{c}$ can be the closest point to the centroid equally at random among all the points in $\mathtt{b}$. We compute the probability of $\mathtt{A} \cap \mathtt{B}$ in the following:

$$\Pr[\mathtt{X}_i = 1] = \Pr[\mathtt{A} \cap \mathtt{B}] = \Pr[\mathtt{A}] \Pr[\mathtt{B} \mid \mathtt{A}] \geq \frac{1}{\chi} \mathtt{p}_\epsilon^{\mathtt{M}(\mathtt{minPts}-1)}$$

We now calculate the probability that $\mathtt{c}$ gets identified as a core-point in any of the hash tables by subtracting one from its complement:

$$\begin{aligned}\Pr[\mathtt{X}_1 = 1 \vee \mathtt{X}_2 = 1 \vee \cdots \vee \mathtt{X}_\mathtt{L} = 1] &= 1 - \Pr[\mathtt{X}_1 = 0 \wedge \mathtt{X}_1 = 0 \wedge \cdots \wedge \mathtt{X}_\mathtt{L} = 0] \\ &= 1 - \prod_{i=1}^{i=\mathtt{L}} (1 - \Pr[\mathtt{X}_i = 1]) \\ &\geq 1 - \left(1 - \frac{1}{\chi} \mathtt{p}_\epsilon^{\mathtt{M}(\mathtt{minPts}-1)}\right)^{\mathtt{L}}\end{aligned}$$

Lemma 2 shows that the probability of identifying any DBSCAN core-point can be made arbitrarily close to 1 by choosing sufficiently large $\mathtt{L}$.

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.

1. As explained in Section 3.2, IP.LSH.DBSCAN only merges the core-points that are within each other's ϵ neighbourhood, see Alg. 1 l. 25.
2. Let $\mathtt{b}_1$ and $\mathtt{b}_2$ be the core-buckets corresponding to $\mathtt{c}_1$ and $\mathtt{c}_2$, respectively. IP.LSH.DBSCAN merges $\mathtt{c}_1$ and $\mathtt{c}_2$ if either ($\mathtt{C}$): $\mathtt{c}_1$ gets hashed to $\mathtt{b}_2$, or ($\mathtt{D}$): $\mathtt{c}_2$ gets hashed to $\mathtt{b}_1$. We calculate the probability of $\mathtt{C} \cup \mathtt{D}$ in the following:

$$\begin{aligned}\Pr[\mathtt{C} \cup \mathtt{D}] &= \Pr[\mathtt{C}] + \Pr[\mathtt{D}] - \Pr[\mathtt{C} \cap \mathtt{D}] \\ &= \mathtt{p}_\epsilon^{\mathtt{M}} + \mathtt{p}_\epsilon^{\mathtt{M}} - \mathtt{p}_\epsilon^{2\mathtt{M}}\end{aligned}$$

Please note that $2\mathtt{p}_\epsilon^{\mathtt{M}} - \mathtt{p}_\epsilon^{2\mathtt{M}}$ is a lower bound because, in addition to $\mathtt{C} \cup \mathtt{D}$, there are other circumstances under which $\mathtt{c}_1$ and $\mathtt{c}_2$ can be merged. For example, $\mathtt{c}_1$ and/or $\mathtt{c}_2$ can be identified as core-points in multiple hash tables, thus increasing the chances that $\mathtt{c}_1$ and $\mathtt{c}_2$ get merged.

4.2. 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 Section 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 of 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 Section 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.

4.3. Completion time analysis

Lemma 5 (*Adapted from Theorem 2 in [22]*)**.** *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* $\mathcal{O}(\frac{\mathtt{N}\log \mathtt{C}}{\mathtt{K}})$.

Proof. In the following, we argue about each of the listed items. We take into consideration that each insertion into a hash table or a set, as associative containers, takes constant time with respect to L, M, and d.

(i) The dominant workload in this phase is to hash N d-dimensional data points into L hash tables using M functions.

(ii) To identify the candidate core-points, all the N points are iterated in each table in the worst-case. Similar is the worst case for identifying the merge tasks. For most instances of the problem, the expected completion time of phase II and III can be significantly smaller than the worst-case bound.

(iii) Similar to the previous case; a merge operation (whose expected time complexity is given in Corollary 1) is performed for each identified merge task, the latter being in the worst case linear in the number of buckets. This upper bound is loose for most data distributions as in the previous case.

(iv) In this phase, a maximum of N findRoot operations, each with expected time complexity of $\mathcal{O}(\log \mathtt{C})$ (Corollary 1), are performed.

The partitioning into S batches and the fine-grained work-sharing schemes as described in the previous section make it possible for the work-load to get evenly distributed among the K threads in each of the above cases.

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 Section 5, the effect of these properties is discussed.

4.4. Parameters 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 [27]. 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 Section 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 [23]. 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.

5.1. Experiment 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 [44] in total.

In addition to IP.LSH.DBSCAN, we benchmark PDSDBSCAN [33], HPDBSCAN [16], and the exact algorithm in [44], for which we use the label TEDBSCAN (Theoretically-Efficient and Practical Parallel DBSCAN). As the approximate algorithms in [44] are generally not faster than their exact counterpart (see Fig. 9 and discussion on p. 13 in [44]), 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 [38,49]. 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.

5.2. Evaluation data & parameters

Following common practices [16,39,44], we use datasets with different characteristics. We use varying ϵ but fixed minPts, as the sensitivity on the latter is significantly smaller [39]. We also follow earlier works' common practice to abort any execution that exceeds a certain bound, here 9×10^5 s (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 Section 4 and also the literature guidelines (e.g., [27] and the reference therein). The *default* ϵ values are shown in *italics*.

TeraClickLog: [44] 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. Following [44], we only consider the integer features, and we choose ϵ from {*1500*, 3000, 6000, 12000} and minPts 100. Household: [14] 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 [14]). Following the practice in [14,44], we scale each feature to [0,10 000] interval and choose ϵ from {1500, *2000*, 2500, 3000} and minPts 100.

GeoLife: [50] From this GPS trajectory dataset, we choose ca 1.5 million points as selected in [24], containing latitude and longitude with a *highly skewed* distribution. We choose ϵ from {*0.001*, 0.002, 0.004, 0.008} and minPts 500, following [24].

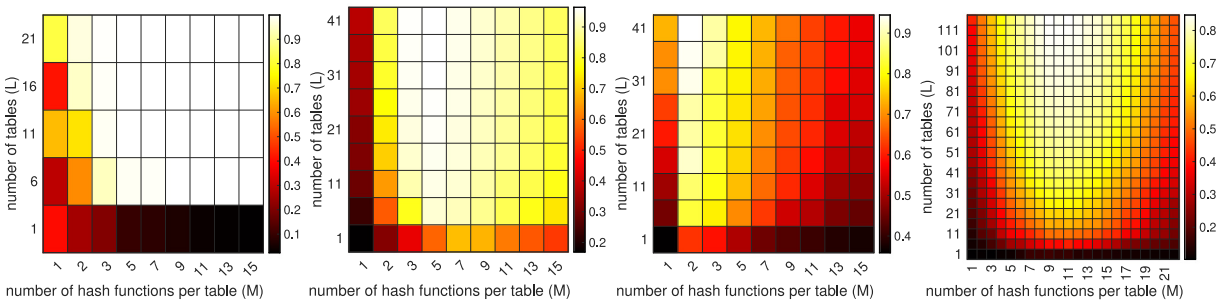
MNIST: This dataset contains 70 000 28×28-pixel hand-written and labelled 0–9 digits [26]. We treat each record as a 784-dimensional data point, normalizing each point to have a unit length (similar to [41]). We utilize the angular distance. Following [40], we choose ϵ from $\{0.18\pi, 0.19\pi, \mathit{0.20}\pi, 0.21\pi\}$ and minPts 100.

The heat-maps in Figs. 5(a)–5(d) visualize IP.LSH.DBSCAN's rand index accuracy as a function of L and M. For TeraClickLog (Fig. 5(a)), {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. 5(b)), {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. 5(c)), {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. 5(d)) 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.

5.3. Experiments for the Euclidean distance

Completion time with varying: K Figs. 6(a), 6(b), and 6(c) 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. 6, 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 s. The results indicate the benefits of parallelization for work-load distribution in IP.LSH.DBSCAN, also validating that IP.LSH.DBSCAN's completion time behaviour 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 Figs. 7(a), 7(b), and 7(c) 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 s threshold. The right Y-axes in Figs. 7(a), 7(b), and 7(c) show the corresponding



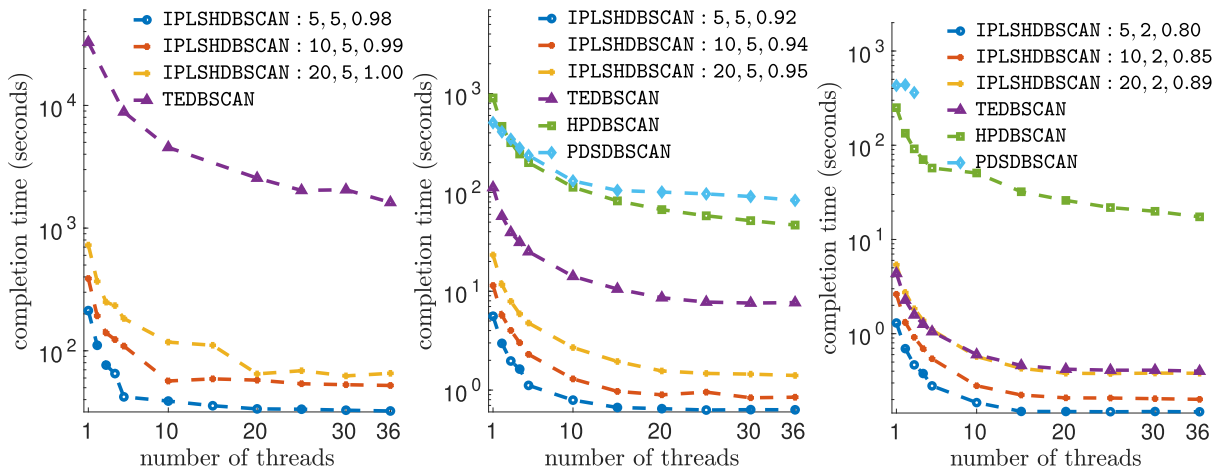
(a) TeraClickLog
ϵ=1500, minPts=100

(b) Household
ϵ=2000, minPts=100

(c) Geolife
ϵ=0.001, minPts=500

(d) MNIST
$\epsilon = 0.2\pi$, minPts=100

Fig. 5. Visualizing the rand index accuracy of IP.LSH.DBSCAN as a function of L and M.

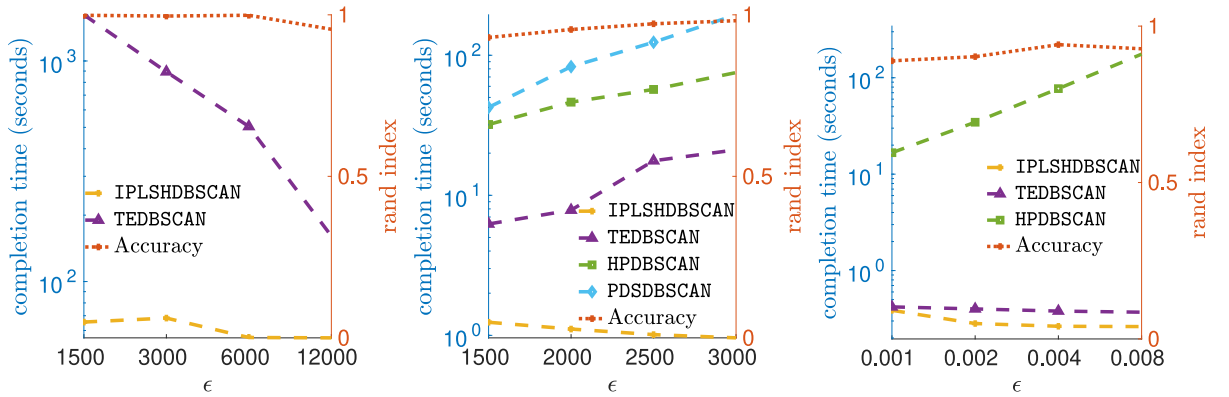


(a) TeraClickLog
ϵ:1500, minPts:100
||VLSHDBSCAN: 5, 5, -|| > 9X10^5

(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. 6. 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 6(a) for all K and for K $\geq$ 4 in 6(c). In 6(a) no HPDBSCAN executions terminate within the 9 $\times$ 10^5-sec threshold.

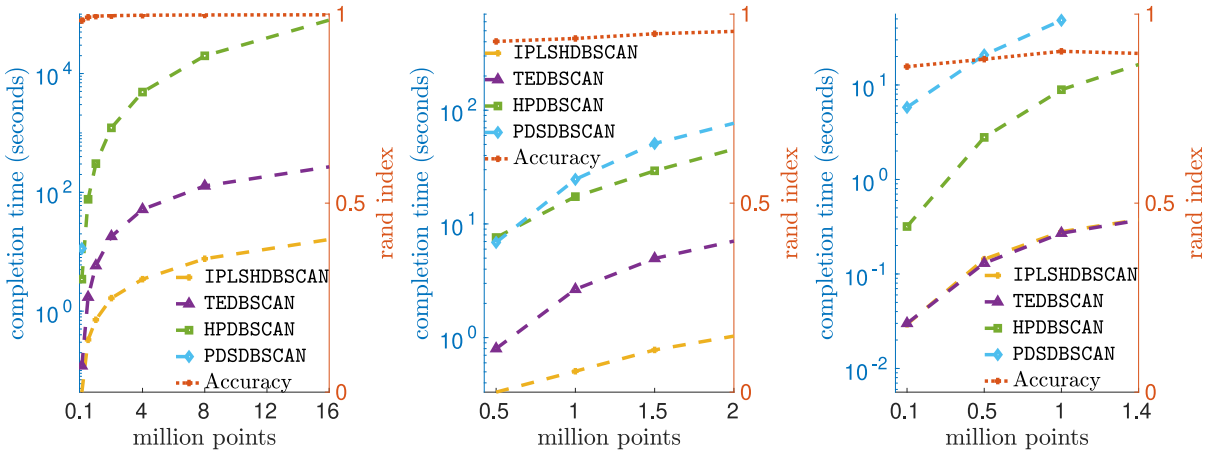


(a) TeraClickLog
minPts:100, L:20, M:5

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

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

Fig. 7. Completion time using varying ϵ with 36 cores. PDSDBSCAN crashes by running out of memory in 7(a) and 7(c) for all ϵ. None of HPDBSCAN's executions terminate within the 9 $\times$ 10^5 s threshold in 7(a). Right Y-axes show IP.LSH.DBSCAN's rand index.

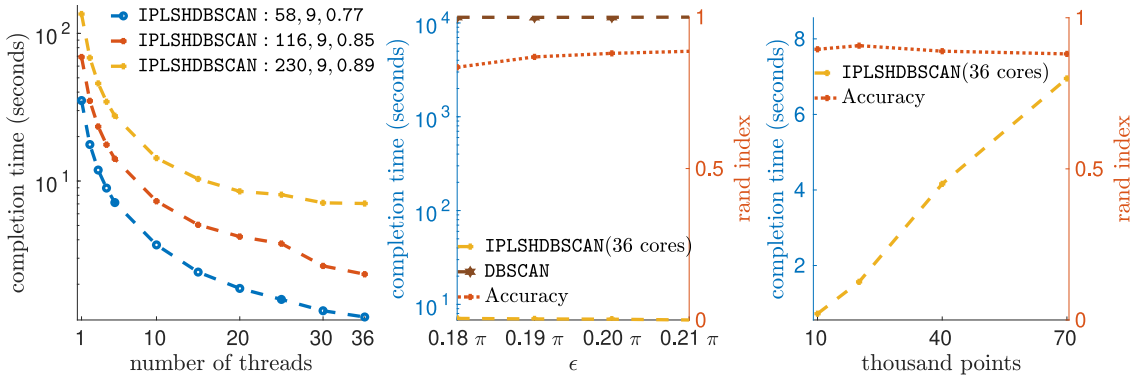


(a) TeraClickLog
ϵ:1500, minPts:100, L:20, M:5

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

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

Fig. 8. Completion time with varying N using 36 cores. PDSDBSCAN runs out of memory in 8(a) with N > 0.1 million points and with N > 1 million points in 8(c). IP.LSH.DBSCAN and TEDBSCAN coincide in 8(c). 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. 9. MNIST results with the angular distance (only IP.LSH.DBSCAN, VLSHDBSCAN, DBSCAN support the angular distance). 9(a) shows IP.LSH.DBSCAN's completion time with varying K. The left Y-axes in 9(b) and 9(c) respectively show IP.LSH.DBSCAN's completion time with varying ϵ and N, using 36 cores. The right Y-axes in 9(b) and 9(c) show the associated accuracy, computed with respect to the actual labels.

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 Figs. 8(a), 8(b), and 8(c) show the completion time of the bench-marked methods using 36 cores on varying size subsets of TeraClickLog, Household, and Geolife datasets, respectively. 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 Figs. 8(a), 8(b), and 8(c) show the corresponding rand index accuracy of IP.LSH.DBSCAN.

5.4. 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 [27]. 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. Fig. 9(a) shows completion time with varying K. The left Y-axes in Figs. 9(b) and 9(c) respectively show completion time with varying ϵ and N, using 36 cores. The right Y-axes in Figs. 9(b) and 9(c) 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.

5.5. Discussion of results

IP.LSH.DBSCAN targets high dimensional, memory-efficient clustering for various distance measures. Its 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, as it 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 Section 5, we focus on related work considering approximation. Gan et al. and Wang et al. in [14,44] proposed approximate DBSCAN clustering for low-dimensional Euclidean distance, with $\mathcal{O}(\mathtt{N}^2)$ complexity if $2^{\mathtt{d}} > \mathtt{N}$ [6]; as when the number of dimensions is considered a constant, the corresponding part of the induced overhead does not show in the asymptotic complexity. The PARMA-CC [24] approach is also suitable only for low-dimensional data. VLSHDBSCAN [38,49] 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.

According to [47], hashing algorithms to facilitate similarity-related data analysis, can be broadly distinguished into two main categories. The first one, locality-sensitive hashing (LSH), is a data-independent approach, which is the type employed in this work. The second category, known as *learning to hash*, comprises data-dependent methods that train hash functions on a given dataset so that nearest-neighbour relationships in the binary code space closely reflect those in the original feature space, while minimizing computational and storage costs. This family of methods has been used for nearest-neighbour retrieval in computer vision tasks, see for example [29], while an end-to-end variant called DeepLSH [34] has also been applied to detect near-duplicate crash reports in software. IP.LSH.DBSCAN can be adopted for custom distance measures, as DeepLSH method can learn custom similarity measures. This could be particularly useful for distance measures that do not have commonly known LSH functions.

Besides LSH-based methods, there exist other approximate-based methods that sacrifice accuracy to gain performance. For example, ρ-approximate DBSCAN [37] produces a clustering outcome which is *sandwiched* between those of DBSCAN with density parameters ϵ and $\epsilon(1+\rho)$. Nevertheless, it has been shown that exact DBSCAN is faster than ρ-approximate DBSCAN for appropriately chosen parameters [37,44]. STING [45] also approximates the clustering outcome of DBSCAN. Other approximation approaches employ sampling techniques. For example, Rough-DBSCAN [42] is a modification of the DBSCAN algorithm that derives representative prototypes from a given dataset, along with density information, which are then used to identify density-based clusters in the dataset. Rough*-DBSCAN [30] is an enhanced version of Rough-DBSCAN that employs an improved prototype selection strategy. These approaches do not overcome the challenges posed by the curse of dimensionality. However, for low-dimensional data clustering, they can be integrated into PARMA-CC's approximation-based synopsis [24].

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 [17,19] and to enable efficient exploration of data (cf. e.g. [31]). 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

We acknowledge the support by: VR grants "Relaxed Concurrent Data Structure Semantics for Scalable Data Processing" (2021–05443), "EPITOME" (2021–05424); Marie Skłodowska-Curie Doctoral Network RELAX-DN funded by EU under Horizon Europe 2021–2027 Framework Programme Grant Agreement nr. 101072456 –www.relax-dn.eu/; Chalmers AoA frameworks Energy and Production, WPs INDEED, and Scalability, Big Data and AI"; TANDEM project within the framework of the Swedish Energy Agency "Electricity Storage and Balancing Centre" (SESBC).

We would like to thank Ralf Klasing for valuable sessions as Licentiate discussion leader and grading committee member of Amir Keramatian's PhD defence. The insightful comments have been inspiring and have significantly influenced the quality of this work. We also wish to acknowledge Ralf's presence and work in the academic community. With sharpness, kindness and insightful contributions, he has shaped his field and inspired many colleagues. Last but not least, we would like to congratulate Ralf on his 60th birthday, and wish him to celebrate numerous birthdays, with many reasons, both professional and personal, to smile with joy!

Data availability

Source code generated for the current study is available in the Figshare repository https://doi.org/10.6084/m9.figshare.19991786; the link is shared in the manuscript.

References

[1] C.C. Aggarwal, A. Hinneburg, D.A. Keim, On the surprising behavior of distance metrics in high dimensional spaces, in: Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings, in: Lecture Notes in Computer Science, vol. 1973, Springer, 2001, pp. 420–434, http://dx.doi.org/10.1007/3-540-44503-X_27.

[2] A. Andoni, P. Indyk, Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, Commun. ACM 51 (1) (2008) 117–122, URL https://doi.org/10.1145/1327452.1327494.

[3] F. Baselice, L. Coppolino, S. D'Antonio, G. Ferraioli, L. Sgaglione, A DBSCAN based approach for jointly segment and classify brain MR images, in: 37th Int. Conf. of the IEEE Eng. in Medicine and Biology Society, EMBC 2015, IEEE, 2015, pp. 2993–2996, URL https://doi.org/10.1109/EMBC.2015.7319021.

[4] J.L. Bentley, K-d trees for semidynamic point sets, in: 6th Symp. on Comp. Geometry, ACM, 1990, pp. 187–197, URL https://doi.org/10.1145/98524.98564.

[5] A. Beygelzimer, S. Kakade, J. Langford, Cover trees for nearest neighbor, in: 23rd Conf. on Machine Learning, ICML '06, ACM, 2006, pp. 97–104, URL https://doi.org/10.1145/1143844.1143857.

[6] Y. Chen, S. Tang, N. Bouguila, C. Wang, J. Du, H. Li, A fast clustering algorithm based on pruning unnecessary distance computations in DBSCAN for high-dimensional data, Pattern Recognit. 83 (2018) 375–387, URL https://doi.org/10.1016/j.patcog.2018.05.030.

[7] P. Ciaccia, M. Patella, P. Zezula, M-tree: An efficient access method for similarity search in metric spaces, in: VLDB'97, 23rd Int. Conf. on Very Large Data Bases, M. Kaufmann, 1997, pp. 426–435, URL http://www.vldb.org/conf/1997/P426.PDF.

[8] M. Datar, N. Immorlica, P. Indyk, V.S. Mirrokni, Locality-sensitive hashing scheme based on P-stable distributions, in: 20th Symp. on Comp. Geometry, SCG '04, ACM, 2004, pp. 253–262, URL http://doi.acm.org/10.1145/997817.997857.

[9] J. Elseberg, D. Borrmann, A. Nuchter, One billion points in the cloud - an octree for efficient processing of 3D laser scans, Int. J. Photogramm. Remote. Sens. 76 (2013) 76–88, http://dx.doi.org/10.1016/j.isprsjprs.2012.10.004.

[10] H. Esfandiari, V.S. Mirrokni, P. Zhong, Almost linear time density level set estimation via DBSCAN, in: 35th AAAI Conf. on Artificial Intelligence, AAAI 2021, AAAI Press, 2021, pp. 7349–7357, URL https://ojs.aaai.org/index.php/AAAI/article/view/16902.

[11] M. Ester, H. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in: 2nd Conf. on Knowledge Discovery and Data Mining, KDD-96, AAAI Press, 1996, pp. 226–231, URL http://www.aaai.org/Library/KDD/1996/kdd96-037.php.

[12] A. Flexer, D. Schnitzer, Choosing L_p norms in high-dimensional spaces based on hub analysis, Neurocomputing 169 (2015) 281–287, http://dx.doi.org/10.1016/j.neucom.2014.11.084.

[13] J. Gan, Y. Tao, DBSCAN revisited: Mis-claim, un-fixability, and approximation, in: 2015 ACM SIGMOD Int. Conf. on Management of Data, ACM, 2015, pp. 519–530, http://dx.doi.org/10.1145/2723372.2737792.

[14] J. Gan, Y. Tao, On the hardness and approximation of euclidean DBSCAN, ACM Trans. Database Syst. 42 (3) (2017) 14:1–14:45, URL https://doi.org/10.1145/3083897.

[15] C.H. Goh, A. Lim, B.C. Ooi, K. Tan, Efficient indexing of high-dimensional data through dimensionality reduction, Data Knowl. Eng. 32 (2) (2000) 115–130, http://dx.doi.org/10.1016/S0169-023X(99)00031-2.

[16] M. Götz, C. Bodenstein, M. Riedel, HPDBSCAN: highly parallel DBSCAN, in: Workshop on Machine Learning in High-Perf. Comp. Environments, MLHPC 2015, ACM, 2015, pp. 2:1–2:10, URL https://doi.org/10.1145/2834892.2834894.

[17] V. Gulisano, Y. Nikolakopoulos, D. Cederman, M. Papatriantafilou, P. Tsigas, Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types, ACM Trans. Parallel Comput. (TOPC) 4 (2) (2017) 1–28.

[18] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: 1984 SIGMOD Int. Conf. on Management of Data, ACM Press, 1984, pp. 47–57, URL https://doi.org/10.1145/602259.602266.

[19] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, A.C. Koppisetty, M. Papatriantafilou, Driven: a framework for efficient data retrieval and clustering in vehicular networks, in: 35th Int'L Conference on Data Engineering, ICDE, IEEE, 2019, pp. 1850–1861.

[20] Y. He, H. Tan, W. Luo, S. Feng, J. Fan, MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data, Front. Comput. Sci. 8 (1) (2014) 83–99, http://dx.doi.org/10.1007/s11704-013-3158-3.

[21] P. Indyk, R. Motwani, Approximate nearest neighbors: Towards removing the curse of dimensionality, in: 30th ACM Symp. on the Theory of Comp., ACM, 1998, pp. 604–613, URL https://doi.org/10.1145/276698.276876.

[22] S.V. Jayanti, R.E. Tarjan, A randomized concurrent algorithm for disjoint set union, in: 2016 ACM Symp. on Principles of Distributed Comp., ACM, 2016, URL https://doi.org/10.1145/2933057.2933108.

[23] A. Keramatian, V. Gulisano, M. Papatriantafilou, P. Tsigas, 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", http://dx.doi.org/10.6084/m9.figshare.19991786.

[24] A. Keramatian, V. Gulisano, M. Papatriantafilou, P. Tsigas, PARMA-CC: parallel multiphase approximate cluster combining, in: 21st Int. Conf. on Distributed Comp. and Networking, ACM, 2020, pp. 20:1–20:10, URL https://doi.org/10.1145/3369740.3369785.
[25] H. Kriegel, M. Schubert, A. Zimek, Angle-based outlier detection in high-dimensional data, in: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008, ACM, 2008, pp. 444–452, http://dx.doi.org/10.1145/1401890.1401946.
[26] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. the IEEE 86 (11) (1998) 2278–2324, http://dx.doi.org/10.1109/5.726791.
[27] J. Leskovec, A. Rajaraman, J.D. Ullman, Mining of Massive Datasets, 2nd ed., Cambridge University Press, 2014, URL http://www.mmds.org/.
[28] H. Lin, High index compression without the dependencies of data orders and data skewness for spatial databases, J. Inf. Sci. Eng. 27 (2) (2011) 561–576, URL http://www.iis.sinica.edu.tw/page/jise/2011/201103_11.html.
[29] H. Liu, R. Wang, S. Shan, X. Chen, Deep supervised hashing for fast image retrieval, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016, IEEE Computer Society, 2016, pp. 2064–2072, http://dx.doi.org/10.1109/CVPR.2016.227.
[30] D. Luchi, A.L. Rodrigues, F.M. Varejão, Sampling approaches for applying DBSCAN to large datasets, Pattern Recognit. 117 (2019) 90–96, http://dx.doi.org/10.1016/J.PATREC.2018.12.010.
[31] L. Magnusson, R. Thorsson, CLUE – Cluster-based Load Understanding and Exploration: Summarizing High-Dimensional Electricity Grid Data for Scenario Analysis Master's thesis, Gothenburg University, Sweden, 2025, URL https://gupea.ub.gu.se/handle/2077/89783.
[32] H. Najdataei, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, Continuous and parallel LiDAR point-cloud clustering, in: 2018 IEEE 38th Int. Conf. on Distributed Comp. Systems, ICDCS, 2018, pp. 671–684, http://dx.doi.org/10.1109/ICDCS.2018.00071.
[33] M.M.A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, A.N. Choudhary, A new scalable parallel DBSCAN algorithm using the disjoint-set data structure, in: SC Conf. on High Perf. Comp. Networking, Storage and Analysis, SC '12, IEEE/ACM, 2012, p. 62, URL https://doi.org/10.1109/SC.2012.9.
[34] Y. Remil, A. Bendimerad, R. Mathonat, C. Raïssi, M. Kaytoue, DeepLSH: Deep locality-sensitive hash learning for fast and efficient near-duplicate crash report detection, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024, ACM, 2024, pp. 198:1–198:12, http://dx.doi.org/10.1145/3597503.3639146.
[35] R.B. Rusu, Semantic 3D object maps for everyday manipulation in human living environments, KI - Künstliche Intell. 24 (4) (2010) 345–348, http://dx.doi.org/10.1007/s13218-010-0059-6.
[36] R.B. Rusu, S. Cousins, 3D is here: Point cloud library (PCL), in: IEEE Int. Conf. on Robotics and Automation, ICRA, IEEE, 2011, URL https://doi.org/10.1109/ICRA.2011.5980567.
[37] E. Schubert, J. Sander, M. Ester, H.P. Kriegel, X. Xu, DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN, ACM Trans. Database Syst. 42 (3) (2017) 19:1–19:21, URL http://doi.acm.org/10.1145/3068335.
[38] Y. Shiqiu, Z. Qingsheng, DBSCAN clustering algorithm based on locality sensitive hashing, J. Phys.: Conf. Ser. 1314 (2019) 012177, http://dx.doi.org/10.1088/1742-6596/1314/1/012177.
[39] H. Song, J. Lee, RP-DBSCAN: a superfast parallel DBSCAN algorithm based on random partitioning, in: 2018 SIGMOD Int. Conf. on Management of Data, ACM, 2018, pp. 1173–1187, URL https://doi.org/10.1145/3183713.3196887.
[40] A. Starczewski, P. Goetzen, M.J. Er, A new method for automatic determining DBSCAN parameters, J. Artif. Intell. Soft Comput. Res. 10 (3) (2020) 209–221, URL https://doi.org/10.2478/jaiscr-2020-0014.
[41] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, P. Dubey, Streaming similarity search over one billion tweets using parallel locality-sensitive hashing, VLDB Endow. 6 (14) (2013) 1930–1941, URL http://www.vldb.org/pvldb/vol6/p1930-sundaram.pdf.
[42] P. Viswanath, V.S. Babu, Rough-DBSCAN: A fast hybrid density based clustering method for large data sets, Pattern Recognit. 30 (16) (2009) 1477–1488, http://dx.doi.org/10.1016/J.PATREC.2009.08.008.
[43] S. Wagner, D. Wagner, Comparing clusterings- an overview, 2007.
[44] Y. Wang, Y. Gu, J. Shun, Theoretically-efficient and practical parallel DBSCAN, in: 2020 SIGMOD Int. Conf. on Management of Data, ACM, 2020, pp. 2555–2571, URL https://doi.org/10.1145/3318464.3380582.
[45] W. Wang, J. Yang, R.R. Muntz, STING: A statistical information grid approach to spatial data mining, in: 23rd Conf. on Very Large Data Bases, VLDB '97, M. Kaufmann, 1997, pp. 186–195, URL http://dl.acm.org/citation.cfm?id=645923.758369.
[46] W. Wang, J. Yang, R. Muntz, PK-tree: A spatial index structure for high dimensional point data, in: Information Organization and Databases: Foundations of Data Organization, Springer US, Boston, MA, 2000, pp. 281–293, http://dx.doi.org/10.1007/978-1-4615-1379-7_20.
[47] J. Wang, T. Zhang, J. Song, N. Sebe, H.T. Shen, A survey on learning to hash, IEEE Trans. Pattern Anal. Mach. Intell. 40 (4) (2018) 769–790, http://dx.doi.org/10.1109/TPAMI.2017.2699960.
[48] X. Wang, L. Zhang, X. Zhang, K. Xie, Application of improved DBSCAN clustering algorithm on industrial fault text data, in: 18th IEEE Int. Conf. on Industrial Inf., INDIN, IEEE, 2020, pp. 461–468, URL https://doi.org/10.1109/INDIN45582.2020.9442093.
[49] Y.-P. Wu, J.-J. Guo, X.-J. Zhang, A linear DBSCAN algorithm based on LSH, in: Int. Conf. on ML and Cybernetics, Vol. 5, 2007, pp. 2608–2614, http://dx.doi.org/10.1109/ICMLC.2007.4370588.
[50] Y. Zheng, X. Xie, W. Ma, GeoLife: A collaborative social networking service among user, location and trajectory, IEEE Data Eng. Bull. 33 (2) (2010) 32–39, URL http://sites.computer.org/debull/A10june/geolife.pdf.