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

Amir Keramatian[1], Vincenzo Gulisano[1], Marina Papatriantafilou[1], and Philippas Tsigas[1]

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. In particular, we show how approximate DBSCAN clustering can be *fused* into the process of creating an LSH index structure, and, through data parallelization and fine-grained synchronization, also utilize efficiently available computing capacity as needed for massive data-sets. The resulting method, IP.LSH.DBSCAN, can effectively support a wide range of applications with diverse distance functions, as well as data distributions and dimensionality. Furthermore, IP.LSH.DBSCAN facilitates adjustable accuracy through LSH parameters. We analyse its properties and also 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.

1 Introduction

Digitalized applications' datasets are getting larger in size and number of features (i.e., dimensions), posing new challenges to established data mining methods such as data clustering, an unsupervised 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, and using user-defined similarity metrics (i.e., not limited to the Euclidean one). DBSCAN is used in many applications, including LiDAR [23], object detection [19], and GPS route analysis [31]. DBSCAN and some of its variants have been also used to cluster high dimensional data, e.g., medical images [4], text [30], and audio [10].

The computational complexity of traditional DBSCAN is in the worst-case quadratic in the input size [24], 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 [14], 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 [19], (ii) the *dimensionality curse* results in exact spatial data structures based on deterministic space partitioning being slower than linear scan [32], 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 [2,3,13,22,31] and approximation [12], studied alone or jointly [19,31]. 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, 16], 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 [16]) 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 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: § 2 reviews the preliminaries. § 3 and § 4 describe and analyse the proposed `IP.LSH.DBSCAN`. § 5 covers the empirical evaluation. Related work and conclusions are presented in § 6 and § 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 $\mathcal{D}$, and having a unique `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* [29]: 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 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 [16, 21], i.e., the property that, with high probability, similar points hash to the same value, and dis-similar ones hash to different ones.

Definition 1. *A family of functions* $\mathcal{H} = \{\mathtt{h} : \mathtt{S} \rightarrow \mathtt{U}\}$ *is* $(\mathtt{d}_1, \mathtt{d}_2, \mathtt{p}_1, \mathtt{p}_2)$*-sensitive for distance function* Dist *if for any* p *and* q *in* S *the following conditions hold: (i) if* $\mathtt{Dist}(\mathtt{p}, \mathtt{q}) \leq d_1$*, then* $\Pr_{\mathcal{H}}[\mathtt{h(p)} = \mathtt{h(q)}] \geq \mathtt{p}_1$ *(ii) if* $\mathtt{Dist}(\mathtt{p}, \mathtt{q}) \geq d_2$*, then* $\Pr_{\mathcal{H}}[\mathtt{h(p)} = \mathtt{h(q)}] \leq \mathtt{p}_2$*. 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 [21]:

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* M*, we can create a new LSH family* $\mathcal{G} = \{\mathtt{g} : \mathtt{S} \rightarrow \mathtt{U}^{\mathtt{M}}\}$ *by aggregating/concatenating* M *LSH functions from* $\mathcal{H}$*, where* g(p) *and* g(q) *are equal iff* $\mathtt{h_j}$(p) *and* $\mathtt{h_j}$(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* L*, we can create a new LSH family* $\mathcal{F}$ *where each* $\mathtt{f} \in \mathcal{F}$ *consists of* L $\mathtt{g_i}$*s chosen independently and uniformly at random from* $\mathcal{G}$*, where* f(p) *and* f(q) *are equal iff* $\mathtt{g_j}$(p) *and* $\mathtt{g_j}$(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 L hash tables; the i-th table is constructed by hashing each point in D using $\mathtt{g_i}$ [9, 16]. 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 [21], e.g.:
LSH for Euclidean distance: Let u be a randomly chosen unit vector in $\mathcal{D}$. A hash function $\mathtt{h_u(x)}$ in such a family is defined as $\lfloor \frac{\mathtt{x} \cdot \mathtt{u}}{\epsilon} \rfloor$, being $\cdot$ the inner product and ϵ 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 $\mathtt{h_u(x)}$ 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 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 [24]. DBSCAN can utilize any distance function e.g., Euclidean, Jaccard, Hamming, angular [21]. Its worst-case time complexity is $\mathcal{O}(\mathtt{N}^2)$, but in certain cases (e.g. for 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 [24].

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 [22]: 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 [17] 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 [31]: 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 ([25,34]) to consider it as a plain means for neighbourhood queries. We refer to them as `VLSHDBSCAN`.

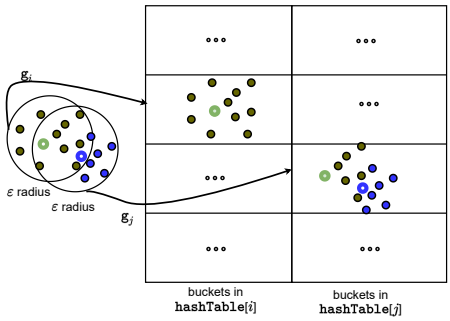
3 The Proposed `IP.LSH.DBSCAN` Method

`IP.LSH.DBSCAN` utilizes the LSH properties, for parallel density-clustering, through efficient fusion of the indexing and clustering formation.

At a 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, core-points within each others' ϵ-neighbourhood get merged. E.g., the core-bucket in the rightmost hash table in Fig. 1a contains two core-points, indicating the possibility that they 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. § 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 § 2, § 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*



(a)

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 LSHDBSCAN

(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.

`minPts` *points within its* ϵ*-radius in* `ccb`*, is called a* core-bucket. Core-forest *is a concurrent union-find structure containing core-points representing core buckets.*

Lemma 1. *A candidate core-point, having in its bucket at least* `minPts` *points within its* ϵ*-radius, is a* core-point *according to the DBSCAN definition (§ 2).*

The above follows from Definition 3. Next we present `IP.LSH.DBSCAN`'s basic phases, followed by detailed description of 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 in `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* 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` (pseudocode Alg. 1), targeting speed-up by distributing the workload 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*): The first step is 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 $\mathtt{S} \times \mathtt{L}$ *hash tasks*, corresponding

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],··· ,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 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
6: let G = {g : S → U^M} be an LSH family suitable for metric Dist, and let g_1, ···, g_L be hash
   functions chosen independently and uniformly at random from 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:        hashTable[i].insert(key = g_i(p), value = ptr(p))
13:        bucket=hashTable[i].getBucket(key = g_i(p))
14:        if bucket.size() ≥ 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 |{q ∈ ccb such that Dist(c, q)}| ≥ minPts then
19:        c→ 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 c ∈ cb such that Dist(core, c) ≤ ϵ 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 p→ corePoint then p.idx = findRoot(p).ID
30:        else p.idx = findRoot(core).ID
```

to the Cartesian product of the hash tables and the data batches. We utilize a mechanism through which the threads can *book* a hash task and thus share the workload. To that end, `hashTasks` is a boolean $\mathtt{S} \times \mathtt{L}$ array containing a *status* for each task, initially `false`. A thread in phase I scans the elements of `hashTasks`, and if it finds an non-booked task, then it tries to *atomically book* the task (e.g. via a `CAS` operation to change the status from `false` to `true`). The thread that successfully books a hash task $\mathtt{ht}_{\mathtt{b},\mathtt{t}}$ hashes each data point `p` in batch `b` into `hashTable`[`t`] using hash function $\mathtt{g}_\mathtt{t}$. 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 stored in the set `candidateCoreBuckets`. Since threads concurrently operate on the same tables, we use hash tables supporting concurrent insertions and traversals [33]. Alg. 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`. Afterwards, it 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` are identified as 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.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 `merge`s 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 appear instantaneously and are consistent with the sequential specification, while the threads can make progress independently of each other [15]) `find` and `merge`, proposed in [15]. The phase is finished when `coreBucekts` becomes empty. Its steps are shown in Alg. 1 l.21-l.24.
Phase IV (*data labeling*): Each non-labeled core-point in a core-bucket gets assigned its root `ID` in the core-forest as the clustering label. The clustering label assigned to all other non-labeled points in a core-bucket is the root `ID` of the associated core-point. The aforementioned process, shown in Alg. 1 l.25-l.30, can be performed concurrently for all the core-buckets.

4 Analysis

We study the consistency, time and memory properties of `IP.LSH.DBSCAN`. Due to space limitations, we omit some proofs, which can be found in [18]. Fig. 1b summarizes the notations.

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 § 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 2. *Any pair of concurrent executions of* `IP.LSH.DBSCAN` *that use the same hash functions, result to 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 § 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. Let `C` be the size of the core-forest, i.e., number of identified core-points by `IP.LSH.DBSCAN`.

Lemma 3. *[adapted from Theorem 2 in [17]] The probability that each* `findRoot` *and each* `merge` *perform* $\mathcal{O}(\log \mathtt{C})$ *steps is at least* $1 - \frac{1}{\mathtt{C}}$.

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

Lemma 4. *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 4. Theorem 1 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 of IP.LSH.DBSCAN 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 § 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 [21]. 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 § 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 it with the established state-of-the-art algorithms. Our implementation is publicly available [18]. 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 [33] 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 [31] in total.

Tested Methods: In addition to IP.LSH.DBSCAN, we perform experiments with PDSDBSCAN [22], HPDBSCAN [13], and the exact algorithm in [31], for which we use the label TEDBSCAN (Theoretically-Efficient and Practical Parallel DBSCAN). As the approximate algorithms in [31] are generally not faster than

their exact counterpart (see Fig. 9 and discussion on p. 13 in [31]), we consider their efficiency represented by the exact TEDBSCAN. We also implemented and tested VLSHDBSCAN, our version of a single-thread DBSCAN that uses LSH indexing, as we did not find open implementations for [25, 34]. Benchmarking VLSHDBSCAN allows a comparison regarding the approximation degree, as well the efficiency induced by the "fused" approach IP.LSH.DBSCAN that leads to the efficient combination of searching and combining through the same hash table. The aforementioned algorithms are reviewed in § 2.

Evaluation Data & Parameters Following common practices [13, 26, 31], we use datasets with different characteristics. We use varying ϵ but fixed minPts, as the sensitivity on the latter is significantly smaller [26]. We also follow earlier works' common practice to abort any execution that exceeds a certain bound, here 9X10^5 seconds (more than 24 hours). 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 § 4 and also the literature guidelines (e.g., [21] and the reference therein). The *default* ϵ values are shown in *italics*.

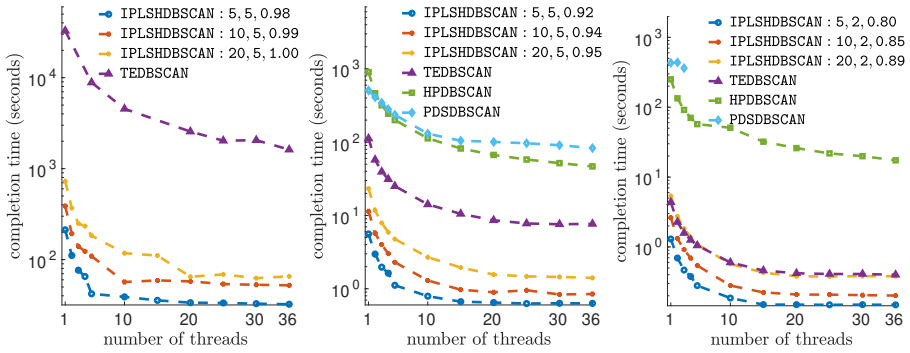
TeraClickLog [31]: Each point in this dataset corresponds a display ad served by Criteo and contains 13 integer and 26 categorical features. We choose a subset containing over 67 million points, free from missing features. Similar to [31], we consider the 13 integer features, and we choose ϵ from $\{$*1500*, 3000, 6000, 12000$\}$ and minPts 100. We choose {L=5,M=5}, {L=10,M=5}, and {L=20,M=5} giving 0.98, 0.99, and 1 rand index accuracy, respectively, with the default ϵ value.

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, 31], we scale each feature to [0,10000] interval and choose ϵ from {1500, *2000*, 2500, 3000} and minPts 100. We choose {L=5,M=5}, {L=10,M=5}, and {L=20,M=5} giving 0.92, 0.94, and 0.95 rand index accuracy, respectively, with the default ϵ value.

GeoLife [35]: This is a GPS trajectory dataset. We used ca 1.45 million points as selected in [19], containing latitude and longitude with a *highly skewed* distribution. Similar to [19], we choose ϵ from {*0.001*,0.002,0.004,0.008} and minPts 500. We choose {L=5,M=2}, {L=10,M=2}, and {L=20,M=2} giving 0.8, 0.85, and 0.89 rand index accuracy, respectively, with the default ϵ value.

MNIST: The set contains 70000 records, each of them a 28X28-pixel hand-written digit 0-9, where the actual labels are available [20]. We treat each record as a 784-dimensional data point, and normalized each data point to have a unit length (similar to [28]). We utilize the angular distance. Following [27], we choose ϵ from $\{0.18\pi, 0.19\pi, \textit{0.20}\pi, 0.21\pi\}$ and minPts 100. We choose {L=58,M=9}, {L=116,M=9}, and {L=230,M=9} giving 0.77, 0.85, and 0.89 rand index, respectively, with the default ϵ, computed with respect to the actual labels.

Experiments for the Euclidean Distance Completion time with varying K: Fig. 2a, Fig. 2b, and Fig. 2c show the completion time of IP.LSH.DBSCAN and other tested methods with varying K on TeraClickLog, Household, and Geolife datasets, respectively. PDSDBSCAN crashes by running out of memory on TeraClickLog for all K and on GeoLife for K$\geq$ 4, and none of HPDBSCAN's executions terminate within the 9X10^5 sec threshold. For the reference, in Fig. 2, the

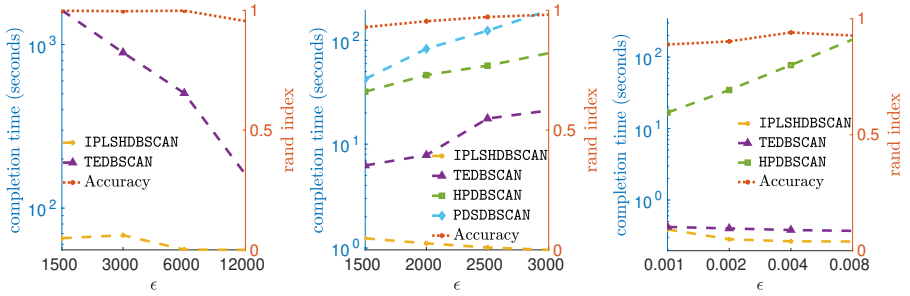


(a) TeraClickLog
ϵ:1500, `minPts`:100
‖`VLSHDBSCAN`: 5, 5, -‖ > $9X10^5$

(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. 2: *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 2a for all* `K` *and for* `K`≥ 4 *in 2c. In 2a no HPDBSCAN executions terminate within the* $9X10^5$*-sec threshold.*

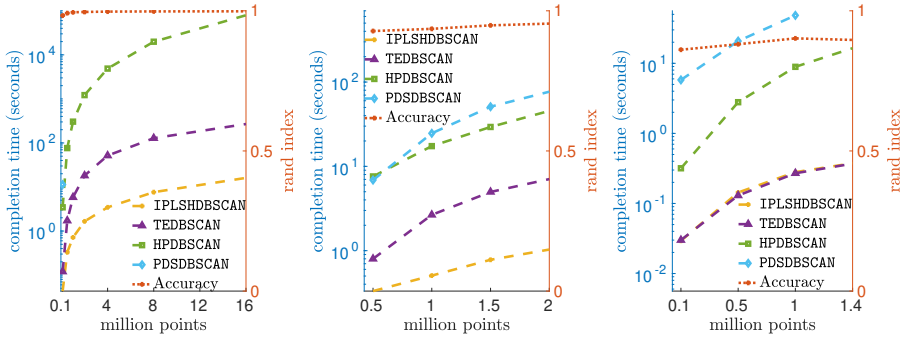


(a) TeraClickLog
`minPts`:100, `L`:20, `M`:5

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

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

Fig. 3: *Completion time using varying* ϵ *with 36 cores. PDSDBSCAN crashes by running out of memory in 3a and 3c for all* ϵ*. None of HPDBSCAN's executions terminate within the* $9X10^5$ *sec threshold in 3a. Right Y-axes show* `IP.LSH.DBSCAN`*'s rand index.*

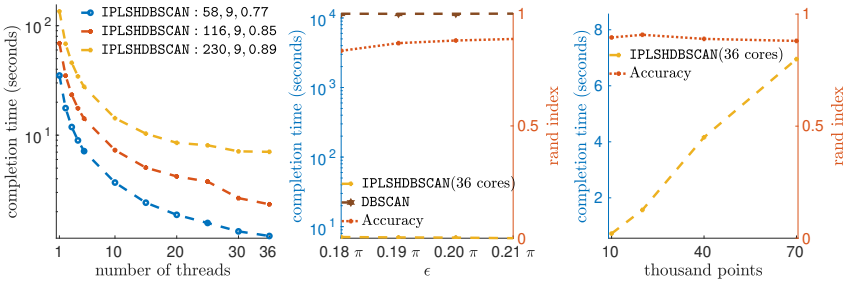


(a) TeraClickLog
ϵ:1500, `minPts`:100, `L`:20, `M`:5

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

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

Fig. 4: *Completion time with varying* `N` *using 36 cores. PDSDBSCAN runs out of memory in 4a with* `N`*>0.1 million points and with* `N`*>1 million points in 4c.* `IP.LSH.DBSCAN` *and TEDBSCAN coincide in 4c. Right Y-axes show* `IP.LSH.DBSCAN`*'s rand index.*



(a) $\epsilon : 0.2\pi$, `minPts`:100 ‖`VLSHDBSCAN`: 58,9,0.55‖=6316

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

(c) $\epsilon : 0.2\pi$, `minPts`:100, `L`:230, `M`:9

Fig. 5: *MNIST results with the angular distance (only* `IP.LSH.DBSCAN`*,* `VLSHDBSCAN`*, DBSCAN support the angular distance). 5a shows* `IP.LSH.DBSCAN`*'s completion time with varying* `K`*. The left Y-axes in 5b and 5c respectively show* `IP.LSH.DBSCAN`*'s completion time with varying* ϵ *and* `N`*, using 36 cores. The right Y-axes in 5b and 5c show the associated accuracy, computed with respect to the actual labels.*

completion time of single-thread `VLSHDBSCAN` is provided as a caption for each dataset, except for TeraClickLog, for the above reason. The results indicate the benefits of parallelization for work-load distribution in `IP.LSH.DBSCAN`, also validating that `IP.LSH.DBSCAN`'s completion time exhibits a linear behaviour with respect to `L`, as shown in Theorem 1. In cases where dimensionality is higher, challenging the state-of-the-art algorithms, `IP.LSH.DBSCAN`'s completion time is several orders of magnitude faster.

Completion time with varying ϵ: The left Y-axes in Fig. 3a, Fig. 3b, and Fig. 3c 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 $9\texttt{X}10^5$ sec threshold. The right Y-axes in Fig. 3a, Fig. 3b, and Fig. 3c show 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. 4a, Fig. 4b, and Fig. 4c 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 (complementing figures in provided in [18]). The right Y-axes in Fig. 4a, Fig. 4b, and Fig. 4c show the corresponding rand index accuracy of `IP.LSH.DBSCAN`.

Experiments for the Angular Distance For data with significantly high number of dimensions, as a side-effect of dimensionality curse, the Euclidean distance among all pairs of points are almost equal [21]. To overcome this issue, we use the angular distance. We only study the behaviour of `IP.LSH.DBSCAN`, `VLSHDBSCAN`, and `DBSCAN` as the other bench-marked methods do not support the angular distance. Here accuracy is calculated against the actual labels. Fig. 5a shows `IP.LSH.DBSCAN`'s completion time with varying `K`. The left Y-axes in Fig. 5b and Fig. 5c respectively show `IP.LSH.DBSCAN`'s completion time with varying ϵ and `N`, using 36 cores. The right Y-axes in Fig. 5b and Fig. 5c show the associated accuracies. The results show `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 clustering, in a memory-efficient way, and it supports various distance measures. `IP.LSH.DBSCAN`'s completion time for high-dimensional datasets is several orders of magnitude faster than state-of-the-art counterparts, while ensuring approximation with tunable accuracy and showing efficiency also with lower dimension data as well. 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

Density clustering is discussed in many related works. In § 5, we compared `IP.LSH.DBSCAN` with representative state-of-the-art related algorithms. We focus here on the related work considering approximation.

Gan et al. and Wang et al. in [12,31] proposed approximate DBSCAN grid-based algorithms, targeting only low-dimensional Euclidean distance, but its expected complexity is $\mathcal{O}(\mathtt{N}^2)$ if $2^{\mathtt{d}} > \mathtt{N}$ [7]. PARMA-CC [19] is another approximate concurrent clustering algorithm suitable only for low-dimensional data.

`VLSHDBSCAN` [25,34] uses LSH for neighbourhood queries. On the other hand, in `IP.LSH.DBSCAN`, creating the LSH index is embedded into the dynamics of the formation of the clusters. The `IP.LSH.DBSCAN` iterates over buckets, and it apply merges on core points that represent bigger entities, drastically reducing the search complexity. Furthermore, `IP.LSH.DBSCAN` is a concurrent algorithm as opposed to `VLSHDBSCAN` which is 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 each hyper-cube. 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

In the landscape of algorithmic implementations of DBSCAN, `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 that the method will help a variety of applications in the evolving landscape of cyber-physical system problems, that require to extract information from very large, high-dimensional, highly-skewed data sets. 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`.

References

1. A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.
2. G. Andrade, G. S. Ramos, D. Madeira, R. S. Oliveira, R. Ferreira, and L. Rocha. G-DBSCAN: A GPU accelerated algorithm for density-based clustering. In *Int. Conf. on Computational Science, ICCS 2013*, volume 18 of *Procedia Computer Science*, pages 369–378. Elsevier, 2013.
3. D. Arlia and M. Coppola. Experiments in parallel clustering with DBSCAN. In *7th Int. Euro-Par Conf.*, volume 2150 of *LNCS*, pages 326–331. Springer, 2001.
4. F. Baselice, L. Coppolino, S. D'Antonio, G. Ferraioli, and 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*, pages 2993–2996. IEEE, 2015.
5. J. L. Bentley. K-d trees for semidynamic point sets. In *6th Symp. on Comp. Geometry*, pages 187–197. ACM, 1990.
6. A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *23rd Conf. on Machine Learning*, ICML '06, page 97–104. ACM, 2006.
7. Y. Chen, S. Tang, N. Bouguila, C. Wang, J. Du, and H. Li. A fast clustering algorithm based on pruning unnecessary distance computations in DBSCAN for high-dimensional data. *Pattern Recognit.*, 83:375–387, 2018.
8. P. Ciaccia, M. Patella, and 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*, pages 426–435. M. Kaufmann, 1997.
9. M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *20th Symp. on Comp. Geometry*, SCG '04, pages 253–262. ACM, 2004.
10. H. Esfandiari, V. S. Mirrokni, and P. Zhong. Almost linear time density level set estimation via DBSCAN. In *Thirty-Fifth AAAI Conf. on Artificial Intelligence, AAAI 2021*, pages 7349–7357. AAAI Press, 2021.
11. M. Ester, H. Kriegel, J. Sander, and 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)*, pages 226–231. AAAI Press, 1996.
12. J. Gan and Y. Tao. On the hardness and approximation of euclidean DBSCAN. *ACM Trans. Database Syst.*, 42(3):14:1–14:45, 2017.
13. M. Götz, C. Bodenstein, and M. Riedel. HPDBSCAN: highly parallel DBSCAN. In *Workshop on Machine Learning in High-Perf. Comp. Environments, MLHPC 2015*, pages 2:1–2:10. ACM, 2015.
14. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *1984 SIGMOD Int. Conf. on Management of Data*, pages 47–57. ACM Press, 1984.

15. M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.
16. P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *30th ACM Symp. on the Theory of Comp.*, pages 604–613. ACM, 1998.
17. S. V. Jayanti and R. E. Tarjan. A randomized concurrent algorithm for disjoint set union. In *2016 ACM Symp. on Principles of Distributed Comp.* ACM, 2016.
18. A. Keramatian. . `https://github.com/amir-keramatian/IP.LSH.DBSCAN`, 2022.
19. A. Keramatian, V. Gulisano, M. Papatriantafilou, and P. Tsigas. PARMA-CC: parallel multiphase approximate cluster combining. In *ICDCN 2020: 21st Int. Conf. on Distributed Comp. and Networking*, pages 20:1–20:10. ACM, 2020.
20. Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11):2278–2324, 1998.
21. J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.
22. M. M. A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, and A. N. Choudhary. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *SC Conf. on High Perf. Comp. Networking, Storage and Analysis, SC '12*, page 62. IEEE/ACM, 2012.
23. R. B. Rusu and S. Cousins. 3d is here: Point cloud library (PCL). In *IEEE Int. Conf. on Robotics and Automation, ICRA*. IEEE, 2011.
24. E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu. Dbscan revisited, revisited: Why and how you should (still) use dbscan. *ACM Trans. Database Syst.*, 42(3):19:1–19:21, July 2017.
25. Y. Shiqiu and Z. Qingsheng. Dbscan clustering algorithm based on locality sensitive hashing. *Journal of Physics: Conf. Series*, 1314:012177, 10 2019.
26. H. Song and J. Lee. RP-DBSCAN: A superfast parallel DBSCAN algorithm based on random partitioning. In *2018 SIGMOD Int. Conf. on Management of Data*, pages 1173–1187. ACM, 2018.
27. A. Starczewski, P. Goetzen, and M. J. Er. A new method for automatic determining DBSCAN parameters. *J. Artif. Intell. Soft Comput. Res.*, 10(3):209–221, 2020.
28. N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *VLDB Endow.*, 6(14):1930–1941, 2013.
29. S. Wagner and D. Wagner. Comparing clusterings- an overview, 2007.
30. X. Wang, L. Zhang, X. Zhang, and K. Xie. Application of improved DBSCAN clustering algorithm on industrial fault text data. In *18th IEEE Int. Conf. on Industrial Inf., INDIN*, pages 461–468. IEEE, 2020.
31. Y. Wang, Y. Gu, and J. Shun. Theoretically-efficient and practical parallel DBSCAN. In *2020 SIGMOD Int. Conf. on Management of Data*, pages 2555–2571. ACM, 2020.
32. R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB'98, 24rd Int. Conf. on Very Large Data Bases*, pages 194–205. M. Kaufmann, 1998.
33. T. Willhalm and N. Popovici. Putting intel threading building blocks to work. In *1st Int. Workshop on Multicore Software Eng.*, IWMSE '08, page 3–4. ACM, 2008.
34. Y.-P. Wu, J.-J. Guo, and X.-J. Zhang. A linear dbscan algorithm based on lsh. In *Int. Conf. on ML and Cybernetics*, volume 5, pages 2608–2614, 2007.
35. Y. Zheng, X. Xie, and W. Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.*, 33(2):32–39, 2010.

A Supplementary Material for Analysis

A.1 Proof of Lemma 4

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.

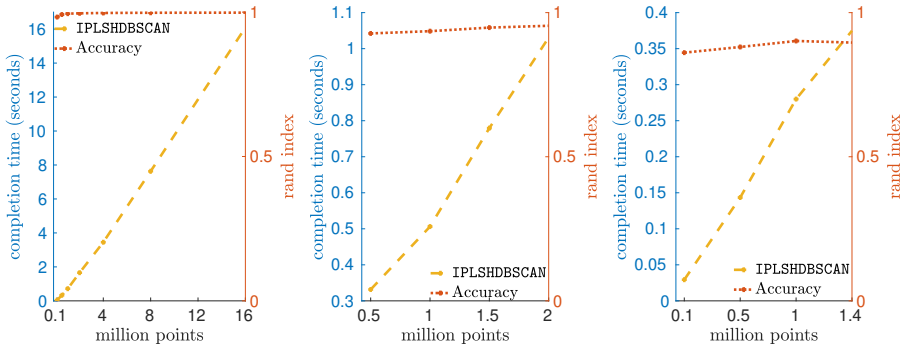
(ii) To identify the candidate core points, all the `N` points are iterated in each table in the worst-case. Similar is the worst case for identifying the merge tasks. For most instances of the problem, the expected completion time of phase II and III can be significantly smaller than the worst-case 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 $\mathcal{O}(\log \texttt{C})$ (Corollary 1), are performed.

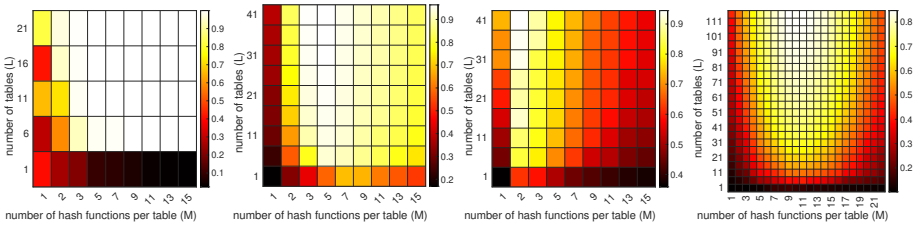
The partitioning into `S` batches and the fine-grained work-sharing schemes as described in the previous section make it possible for the work-load to get evenly distributed among the `K` threads in each of the above cases.

B Supplementary Material for Evaluation



(a) TeraClickLog L:20, M:5 **(b)** Household L:20, M:5 **(c)** Geolife L:20, M:2

Fig. 6: Completion time measurements using 36 cores. Right Y-axes show the rand index accuracy of `IP.LSH.DBSCAN`.



(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. 7: Heat-maps visualizing the rand index accuracy of `IP.LSH.DBSCAN` as a function of `L` and `M`