A Generic Distributed Clustering Framework for Massive Data

Pingyi Luo
School of Computing
National University of Singapore
Singapore
pingyi@u.nus.edu

Qiang Huang
School of Computing
National University of Singapore
Singapore
huangq@comp.nus.edu.sg

Anthony K. H. Tung
School of Computing
National University of Singapore
Singapore
atung@comp.nus.edu.sg

ABSTRACT

In this paper, we introduce a novel Generic distributEd clustErng frameworK (GEEK) beyond k-means clustering to process massive amounts of data. To deal with different data types, GEEK first converts data in the original feature space into a unified format of buckets; then, we design a new Seeding method based on simILar buckEtS (SILK) to determine initial seeds. Compared with state-of-the-art seeding methods such as k-means++ and its variants, SILK can automatically identify the number of initial seeds based on the closeness of shared data objects in similar buckets instead of pre-specifying k. Thus, its time complexity is independent of k. With these well-selected initial seeds, GEEK only needs a one-pass data assignment to get the final clusters. We implement GEEK on a distributed CPU-GPU platform for large-scale clustering. We evaluate the performance of GEEK over five large-scale real-life datasets and show that GEEK can deal with massive data of different types and is comparable to (or even better than) many state-of-the-art customized GPU-based methods, especially in large k values.

KEYWORDS

K-Means Clustering, Initial Seeding, Locality-Sensitive Hashing, Heterogeneous Data, CPU-GPU Platforms

ACM Reference Format:

Pingyi Luo, Qiang Huang, and Anthony K. H. Tung. 2021. A Generic Distributed Clustering Framework for Massive Data. In Conf ’21: Proceedings of 2021 ACM XXX Conference, June, 2021, Online. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Over the last decade, data sets have emerged on an unprecedented scale across different scientific disciplines [7]. The data types in different scenarios may be quite different, i.e., dense or sparse. Moreover, the heterogeneous data with the co-existence of categorical and numeric attributes become very popular. This development has created a new demand for distributed data mining and machine learning methods to deal with massive amounts of data of different data types. This paper considers the generic distributed clustering beyond k-means, which is useful in two ways.

First, as a method to find clusters, many partitioning based methods like k-means are linear in complexity in terms of both cardinality (i.e., n) and dimensionality (i.e., d) of the data. Their results can also be easily interpreted by looking at the representative of each cluster that is found. Comparatively, other methods like hierarchical [23, 58], density-based [4, 20, 22] and grid-based clustering [1, 54] are typically more expensive especially in high-dimensional spaces [22, 53]. Interpretability wise, while hierarchical methods can form dendrograms to facilitate the understanding of these clusters, understanding the clusters that are discovered through density or grid based methods is much harder.

Second, using partitioning methods such as k-means to preprocess the data into a large number of microclusters (i.e., k sufficiently high but k << n) [32], other kinds of clustering methods can speed up substantially. This can be done in two ways. First, for hierarchical methods, one can directly use the microclusters as the base objects to form hierarchical clusters instead of using the actual data points which are much higher in numbers than the microclusters. Second, one can utilize the microclusters to build indexes to support the efficient execution of neighborhood queries in density or grid-based methods [33]. Both of them rely on the efficiency (i.e., can handle large values of n and k) and effectiveness (i.e., high-quality microclusters with low diameters) of the partitioning methods. In view of the usefulness of partitioning methods such as k-means, we will look at their speedup in this paper.

A popular version of k-means is the Lloyd’s algorithm [41], which is conducted repeatedly by assigning data objects to their closest central vectors (e.g., centroids) and then updating the k central vectors by their assigned members, such that the total distances between data objects and their closest central vectors are minimized. Various methods and optimizations are proposed based on this algorithm to deal with the problems of initial seeds selection [5–8, 14], the large values of d and k [15, 18, 19, 25, 47], and the acceleration for massive data (i.e., n) with modern hardware [12, 38, 40]. However, most of them are highly optimized for a specific distance or similarity measure, e.g., Euclidean distance or its variants. To the best of our knowledge, there is no generic k-means variant that can deal with different types of data.

On the other hand, Locality-Sensitive Hashing (LSH) [26, 31] and its variants [2, 3, 11, 13, 16, 29, 36, 37, 42, 51, 60] are often used for clustering. The most typical usage of LSH is to accelerate the Nearest Neighbor Search (NNS) during clustering [14, 34, 39]. For example, Cohen-Addad et al. leveraged LSH to shortlist the candidates to compare with the centroids, so that the expensive distance computations can be vastly relieved [14]. Another common scenario is to approximate the distance with LSH to avoid actual computation [17, 57]. In [57], they used MinHash [11] and SimHash [13] to approximate the distance for high-dimensional data (e.g.,
text) and applied a bisecton method to the LSH similarity matrix for clustering. For the large-scale, distributed setting, LSH is also applied for data partitioning to reduce communication cost, e.g., [9, 59] exploited LSH to distribute close data on the same machine in a distributed environment and find clusters locally; then, they aggregated local results to approximate the global clusters.

Most of these methods, however, only use LSH as a basic tool for speedup or data partitioning. They focus on the data objects in a single bucket but do not consider the connections between buckets. Intuitively, similar buckets often share many common data objects under many LSH functions. These shared data are close to each other, which are the ideal candidates to determine initial seeds for clustering. Moreover, there exist different kinds of LSH families for different types of data. With different LSH families, different types of data can be converted into a unified format of buckets.

Our Contributions. Based on the above motivations, we introduce a novel Generic distributed clustering framework (GEEK) to deal with large-scale data of different types. We first leverage different LSH families to convert different types of data into a unified format of buckets. Then, we design a new seeding algorithm based on similar bucket sets (SILK) to generate initial seeds. Specifically, we adapt MinHash [10, 11], which is a popular LSH scheme for NNS between sets, to retrieve similar buckets. Then, we use the shared data objects from similar buckets to determine the initial seeds. As distinct from state-of-the-art seeding methods such as k-means++ and its variants [5, 7, 8, 14], SILK can automatically identify initial seeds without pre-specifying k, and its time complexity is independent of k. With these well-selected initial seeds, we get the final clusters through a one-pass data assignment. We exploit the parallelism with multiple computing nodes and GPUs and implement GEEK on a distributed CPU-GPU platform for large-scale clustering. Extensive experiments over five massive real-life datasets validate the efficiency and effectiveness of GEEK.

Organization. The roadmap of this paper is as follows. Section 2 reviews some preliminary knowledge. We introduce GEEK and its distributed implementation in Section 3. Experimental results are reported and analysed in Section 4. Section 5 surveys the related work. Finally, we conclude our work in Section 6.

2 PRELIMINARIES

2.1 Problem Settings

In this paper, we focus on both dense and sparse data, where dense data consists of data objects whose values in most feature dimensions (or attributes) are non-zero, whereas the sparse data mainly contains zeros. The dense data can be further divided into the homogeneous dense data (with single numeric or categorical attribute) and the heterogeneous dense data (with the co-existence of numeric and categorical attributes). The sparse data is often homogeneous; otherwise, we can encode the data into a single attribute (e.g., one-hot encoding) due to the ultra-high dimensions.

Suppose data objects are represented as vectors in a d-dimensional space $\mathbb{R}^d$. Let $\text{Dist}(x, y)$ be the distance between any $x, y \in \mathbb{R}^d$.

**Definition 1** (k-means clustering): Given a data set $\mathcal{D}$ of data objects, i.e., $\mathcal{D} = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$, k-means clustering aims to find $k$ central vectors $C = \{c_1, \ldots, c_k\} \subset \mathbb{R}^d$ such that the total distance $\text{Dist}(\mathcal{D}, C)$ between all of the data objects and their closest central vectors is minimized, where $\text{Dist}(\mathcal{D}, C) = \sum_{i=1}^{n} \text{Dist}(x_i, c_i)$, and $ID_i$ is the index of the closest central vector of $x_i$, i.e.,

$$
ID_i = \arg \min_{j \in \{1, \ldots, k\}} \text{Dist}(x_i, c_j).
$$

$\text{Dist}(x, y)$ can be widespread distance functions, such as Euclidean distance, Manhattan distance, Hamming distance, etc. Given a similarity function $\text{Sim}(x, y) \in [0, 1]$, e.g., Jaccard similarity, we consider its corresponding distance $\text{Dist}(x, y) = 1 - \text{Sim}(x, y)$ in Equation 1. Notice that we do not claim that GEEK can support every distance and similarity measure – just the distances or similarity measures if and only if there exists the corresponding LSH family.

2.2 Locality-Sensitive Hashing

**Definition 2** (LSH family [31]): Given a search radius $R > 0$ and an approximation ratio $c$, a family of hash functions $\mathcal{H}$ is $(R, cR, p_1, p_2)$-sensitive for $\text{Dist}(\cdot, \cdot)$ if, for any $x, y \in \mathbb{R}^d$, the hash function $h$ drawn uniformly from $\mathcal{H}$ satisfies the following conditions:

- If $\text{Dist}(x, y) \leq R$, then $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \geq p_1$;
- If $\text{Dist}(x, y) > cR$, then $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq p_2$;
- $p_1 > p_2$ and $c > 1$.

Next, we review two famous LSH families, MinHash [10, 11] and Query-Aware LSH (QALSH) [27, 28], which will be used later.

**MinHash.** MinHash is designed for Jaccard similarity $J(A, B) = |A \cap B|/|A \cup B|$ between any two sets (or objects) $A, B \in \mathcal{U}$. The MinHash function $h_\pi(\cdot)$ is defined as follows:

$$
h_\pi(A) = \min_{a \in A} \pi(a),
$$

where $\pi$ is a random permutation from the ground universe $\mathcal{U}$ and $h_\pi(A)$ is the minimal member of $A$ with respect to $\pi$.

MinHash uses the static $(K, L)$-bucketing framework [31] to deal with the NNS query. It concatenates $K$ i.i.d. LSH functions to form a signature, i.e., $G(\cdot) = (h_{\pi_1}(\cdot), \ldots, h_{\pi_K}(\cdot))$. Since $G(\cdot)$ will reduce $p(\cdot)$ between close objects, MinHash builds $L$ hash tables to achieve a theoretical guarantee. MinHash looks up the $L$ buckets based on the signatures of the query to find the candidates.

**QALSH.** QALSH is developed for Euclidean distance $\|x - y\| = \sqrt{\sum_{j=1}^{d} (x_j - y_j)^2}$, where $x = (x_1, \ldots, x_d)$ and $y = (y_1, \ldots, y_d)$. The QALSH function $h_\alpha(\cdot)$ is defined as follows:

$$
h_\alpha(x) = a \cdot x,
$$

where $a$ is a random vector with each entry $a_i$ chosen i.i.d. from the standard normal distribution, i.e., $a_i \sim \mathcal{N}(0, 1)$.

QALSH adopts the dynamic collision counting framework [21] for the NNS. It draws $m$ i.i.d. LSH functions to build $m$ hash tables, where each stores the sorted LSH values only. Let $w$ be the bucket width. Given a query $q$, the interval $[h_\alpha(q) - \frac{w}{2}, h_\alpha(q) + \frac{w}{2}]$ is regarded as the anchor bucket of $q$. The bucket partitioning waits until $q$ arrives. We select an object as a candidate of $q$ if they collide more than $l$ times, where $l$ is a pre-specified collision threshold.

3 GEEK

We now present GEEK for generic distributed clustering. GEEK consists of three phases: data transformation, initial seeding, and data assignment. An overview of GEEK is shown in Figure 1.
3.1 Data Transformation
In this phase, we convert different types of data into a unified format of buckets. We illustrate one specific distance for each data type for demonstration purposes.

Homogeneous Dense Data. We consider high dimensional points (or vectors) for Euclidean distance. We first draw i.i.d. QALSH functions and build m hash tables. Let \( t \) be a threshold for bucket partitioning. Then, we sort the hash tables in ascending order of the LSH values and evenly partition the data IDs in each hash table into \( t \) buckets. Let \( B \) be a collection of buckets generated from the \( m \) hash tables. The pseudo-code is depicted in Algorithm 1.

### Example 1
We now use Figure 1 as an illustration. Suppose \( D = \{x_1, x_2, \cdots, x_6\} \) and there exist 3 clusters, i.e., \( \{x_1, x_2, x_4\} \), \( \{x_3, x_5\} \), and \( \{x_6\} \). We use \( m = 4 \) QALSH functions \( \{h_{a_1}, \cdots, h_{a_4}\} \) to transform all \( x_j \in D \) into buckets. Let \( t = 2 \). For the hash table \( T_1 \), we compute \( h_{a_1}(x_j) \) for each \( x_j \) and partition all of them into two buckets, i.e., \( B_1 = \{x_1, x_2, x_4\} \) and \( B_2 = \{x_3, x_5, x_6\} \). We repeat this procedure for \( T_2 \), \( T_3 \), and \( T_4 \) and get \( B = \{B_1, B_2, \cdots, B_8\} \).

### Remarks
Notice that we do not use the popular LSH function \( h_{a,b}(x) = \frac{a \cdot x + b}{d+1} \) [16] for bucket partitioning, because it contains a parameter named bucket width \( w \) that is hard to tune for different datasets if we want to partition data into buckets in a suitable granularity. Yet, we do not simply follow QALSH and use \( \tilde{h}_a(x) = \frac{w}{2}, \tilde{h}_a(x) + \frac{w}{2} \) as the bucket of each data object \( x \), because this way is customized for the query, which is not suitable for data objects. Instead of using \( w \), we introduce a new parameter \( t \) to evenly partition each hash table into \( t \) parts as buckets. This new way is simple and easy to control the granularity of buckets, and it can keep the proximity of most data objects in the same buckets.

Heterogeneous Dense Data. The categorical attributes are discrete values, which is a natural division for buckets. However, the numeric attributes are continuous values, which cannot be divided into buckets directly. Thus, we first follow the homogeneous dense data transformation to discretize the numeric attributes into categorical attributes. We then follow MinHash and leverage the static (\( K, L \))-bucketing framework to transform the categorical attributes into buckets. The pseudo-code is shown in Algorithm 2.

### Sparse Data
We consider data objects represented as sets for Jaccard similarity. Since sparse datasets are often ultra-high dimensional, e.g., million-scale dimensions, we first follow [55] and use DOPH [52] to reduce the dimensionality to a moderate level, e.g., hundreds of dimensions.\(^1\) We choose DOPH for dimensionality reduction because with DOPH, the distance among each pair of data after dimension reduction can still be approximately preserved with high probability [55]. Then, we follow MinHash and use the static

\(^1\)In our experiments, for the sparse dataset URL, we first use DOPH to reduce the data dimension to 400.
We now introduce SILK to generate initial seeds based on similar intuition, if initial seeds the larger the size of $C$. Identify the frequent shared data objects from similar buckets. In order to determine the initial seeds, we first adopt the static $(K, L)$-bucketing framework to transform the collection of buckets $B$ into bins. For each bin $B_i$, if it contains at most one bucket, i.e., $|B_i| \leq 1$, we ignore it because this bucket is probably isolated from others; otherwise, we identify a group of shared data objects (i.e., $C_{shared}$) from $B_i$ by majority voting, i.e., we select the data objects that appear in more than half number of the buckets in $B_i$. With majority voting, we can identify the frequent shared data objects from similar buckets. Intuitively, the higher the frequency, the closer the distance to other shared data objects. After identifying a collection of $C_{shared}$’s, we need to decide which $C_{shared}$ can be used to determine initial seeds. Intuitively, the larger the size of $C_{shared}$, the higher the probability that there exists a cluster among the data objects in $C_{shared}$. Let $\delta$ be a seeding threshold and $C$ be a collection of initial seeds. Based on this intuition, if $|C_{shared}| \geq \delta$, we will add it into $C$.

However, when we repeat the above procedure to identify the initial seeds $C$ for the $L$ hash tables, some very similar or even identical $C_{shared}$’s may be added into $C$ multiple times. Fortunately, $C$ can also be regarded as a set of buckets. We can use $C$ as input and repeat the above procedure again to remove the near-duplicates. The pseudo-code is depicted in Algorithm 4.

**Algorithm 4: SILK**

**Input:** $B$, a seeding threshold $\delta$, and $K, L$ for partitioning;

**Output:** A collection of initial seeds $C$;

1. $C \leftarrow \emptyset$;
2. for $i$ from 1 to $L$ do
3.   Draw $K$ MinHash functions $\{h_{\pi_1}, \cdots, h_{\pi_K}\}$;
4.   $T \leftarrow \emptyset$;
5.   for each bucket $B_j \in B$ do
6.     $G(B_j) = (h_{\pi_1}(B_j), \cdots, h_{\pi_K}(B_j))$;
7.     $T[G(B_j)], push(B_j)$;
8.   for each $Bin_j \in T$ do
9.     if $|Bin_j| \leq 1$ then continue;
10.    Find $C_{shared}$ from the buckets in $Bin_j$;
11.   if $|C_{shared}| \geq \delta$ then $C \leftarrow C \cup C_{shared}$;
12. Remove the near duplications of $C$;
13. return $C$;

only appear once, we ignore them. Since $|C_{shared}| = 3 \geq \delta = 1$, we add $C_{shared} = \{x_1, x_2, x_4\}$ into $C$. We ignore $B_{in_1}$ since $|B_{in_1}| \leq 1$. After identifying $C_{shared}$’s, some very similar or even identical $C_{shared}$’s are added into $C$, e.g., we also add $C_{shared} = \{x_1, x_2, x_4\}$ from $B_{in_2}$ into $C$. Thus, we repeat this procedure again with $L = 1$ to deduplicate $C_{shared}$’s. Finally, we get four $C_{shared}$’s in $C$, i.e., $\{x_1, x_2, x_4\}, \{x_3, x_5, x_6\}, \{x_8\}$, and $\{x_3, x_5\}$.

**Remarks.** The state-of-the-art seeding methods, such as $k$-means++ and its variant [5, 7, 14], require setting up $k$ in advance. Moreover, their time complexities are highly sensitive to $k$, which limit their scalability for large $k$ values. In contrast, as will be discussed in Section 3.5, even though SILK needs to tune a few parameters, i.e., $K, L$, and $\delta$, its time complexity is independent of $k$, and it can efficiently generate a large number of initial seeds.

**3.3 Data Assignment**

In this phase, we first compute the central vectors from the initial seeds $C$; then, we assign each data object to its closest central vector to achieve the final clusters. With the efficiency of SILK, we often generate more initial seeds to get more options to detect small clusters. Thus, one-pass data assignment is sufficient.

**Example 3:** From Figure 2, if we just build a single hash table $T_1$, GEEK generates two initial seeds from $C_{shared} = \{x_1, x_2, x_4\}$ and $C_{shared} = \{x_3, x_5, x_6\}$ only. The small cluster $\{x_9\}$ will be combined together with the cluster $\{x_3, x_5\}$ and cannot be distinguished. Thus, we often build more hash tables to get more initial seeds. As shown in Figure 1, with one more $T_2$ to get more seeds from $C_{shared} = \{x_4\}$ and $C_{shared} = \{x_8\}$, we can easily identify $\{x_8\}$ and avoid multiple assign-update iterations.

There are several ways to get more initial seeds. In Example 2, we only use hash tables in SILK to generate more seeds. Meanwhile, we can also generate more buckets from data transformation or decrease the seeding threshold $\delta$ to get more seeds.

**Central Vector Representation.** For the homogeneous dense data such as the high-dimensional points, we use centroid as the central
vector. For the heterogeneous dense data, it is hard to define a central vector for clustering with its original representation. Thus, we use the unified categorical attributes for data assignment and use mode to represent the central vector. For the sparse data such as sets, we also use mode as the central vector. Moreover, since the original dimension is often very large, we use the dimension after DOFP to represent data objects to reduce the computational cost.

3.4 Implementation

Next, we introduce how we design and implement GEEK on a distributed CPU-GPU platform.

**System Architecture.** Suppose we have $g$ GPUs in the system across $N$ computing nodes. Each node can have different number of GPUs, i.e., $g_i$ for node $i$ and $\sum_{i=1}^{N} g_i = g$. We adopt the Message Passing Interface (MPI) for the communication of different computing nodes. We launch $g$ processes with MPI. Each process $P_j$ corresponds to a single GPU for parallel computation, where $1 \leq j \leq g$. The system architecture is depicted in Figure 2.

**Data Transformation.** We first instruct GPU threads to compute the LSH values for the data objects that each process $P_j$ contains. Since each $P_j$ only has a part of the data, the buckets in different processes are incomplete. We need to conduct a bucket synchronization among different processes to rearrange the buckets so that (1) data objects with the identical LSH values are put into the same buckets; (2) each $P_j$ contains different hash tables. This procedure aims to deal with a load balance problem, which will be discussed later.

**Initial Seeding.** After the bucket synchronization, we instruct the GPU threads to compute bins for the buckets that each process $P_j$ contains. Similar to the case of data transformation, the bins in different processes are incomplete. However, to reduce the communication cost, we do not rearrange the bins. Instead, we determine the shared data objects $C_{shared}$ for each $P_j$ from its local bins and synchronize the $C_{shared}$’s among different processes. We will introduce the details of this procedure after the load balance issue.

**Data Assignment.** Since different data are located in different nodes, we cannot directly compute the central vectors (e.g., centroids). To reduce the communication cost, we first instruct the GPU threads to compute the local centroids and count their local size for each process $P_j$. Then, we broadcast the local centroids and their sizes to get the global centroids. At last, each $P_j$ performs a one-pass data assignment and gets the final clusters.

**Load Balance.** Load balance is an essential issue for distributed clustering. It consists of two aspects: original data load balance and intermediate data load balance. For the original data, we evenly split the dataset into $g$ parts so that each process has an equal portion of the dataset. Thus, the workload regarding the original data such as data transformation and data assignment are balanced.

For the intermediate data, the primary bottleneck is how to synchronize the buckets to keep the workload balance for SILK during the phase of initial seeding. A direct way is to assign the number of buckets to $g$ processes as even as possible. However, different buckets might have a different number of data IDs. Notice that we only evenly partition data objects into buckets for the homogeneous dense data. Some processes with small buckets will be idle, while other processes with large buckets may be hectic. In this way, the workload for SILK is imbalanced.

As will be discussed in Section 3.5, the time complexity of SILK is determined by the total number of data IDs among the buckets, so we evenly assign the number of hash tables to $g$ processes. The insight is that each hash table may have a different number of buckets, but it contains the same number of data IDs. Furthermore, we can control the number of hash tables (e.g., $L$) divisible by $g$. Thus, the intermediate workload for SILK can be balanced by this way of bucket synchronization.

**Multi-Loading Strategy.** The GPU memory might not be sufficient for the billion-scale or even larger-scale data to keep a single hash table. To fix this problem, we implement a multi-loading strategy as follows. Each process runs SILK multiple times; each time, we load as many buckets as possible to fit the GPU memory. Since the GPU memory of each process can be identical, the workload of each process will not be much affected by this multi-loading strategy.

**Communication Cost.** In SILK, the majority voting needs the global frequency of data IDs in different buckets to determine the $C_{shared}$’s. However, different processes only have a part of the bins. We need to conduct a synchronization to broadcast bins (including the local bucket IDs and their data IDs) among different processes, but it will lead to a very high communication cost.

In our implementation, we adapt majority voting in SILK to identify $C_{shared}$’s for each process $P_j$ from its local bins.
We now discuss the time and space complexities of GEEK. Let us be concise, we only analyse the time and space complexities for Data Transformation.

Example 4: We now use Figure 3 for an illustration. Suppose the data objects and buckets in Figure 3 are as same as those in Figure 1. Considering the process P1. For Bin1, since the data objects x1, x2, x4 appear more than half time of the buckets B1 and B3, we get C-shared = {x1, x2, x4} from Bin1. Similarly, we get C-shared = {x1, x2, x4} and C-shared = {x6} from Bin3 and Bin5, respectively. We ignore Bin2 and Bin4 because |Binj| ≤ 1. The left part of Figure 3 shows the C-shared’s from the local bins of P1 and P2. After that, we conduct SILK again to remove the duplicates {x1, x2, x4} and {x3, x5} from Bin3 and Bin5, respectively. The right part of Figure 3 shows the state after deduplication. \( \triangle \)

Since the majority voting is conducted on local bins, the C-shared’s in Figure 3 might be slightly different from those in Figure 1. For example, we get \{x3, x5\} instead of \{x3, x5, x6\} from Bin2. However, if the shared data objects are close enough, most of them could be identified from local bins. Thus, we argue that it is reasonable to use this operation to reduce the communication cost by allowing a minor loss inaccuracy.

### 3.5 Complexity Analysis

We now discuss the time and space complexities of GEEK. Let \( k \) be the number of central vectors for data assignment, i.e., \( k = |C| \). To be concise, we only analyse the time and space complexities for the homogeneous dense data.

**Data Transformation.** We first take \( O(md) \) time and \( O(md) \) space to use \( m \) QALSH functions to compute the LSH values. Then, we spend \( O(mn \log n) \) time to scan and sort the \( m \) hash tables for bucket partitioning. We also use \( O(mn) \) space to store buckets (with data IDs only). According to QALSH [28], \( m = O(\log n) \). Thus, we need \( O(dn \log n + n \log^2 n) \) time and \( O(n \log n) \) space.

**Initial Seeding.** We then analyse the time and space complexities of SILK. Suppose \( N_B \) is the number of buckets and \( D_B \) is the average size of data IDs in each bucket. For the homogeneous dense data, \( N_B = mt \) and \( D_B = \frac{1}{n} \), so \( N_B D_B = mn = O(\log n) \). Moreover, we can control the granularity of buckets so that \( N_B = O(n) \). SILK first uses MinHash [11] and takes \( O(N_B D_B K) \) time and \( O(N_B L) \) space to compute bins. The majority voting needs \( O(N_B D_B L) \) time to scan all buckets and their data IDs in bins to determine \( C_{\text{shared}} \)’s. Let \( N_C \) be the total number of data IDs in \( C \), i.e., \( N_C = \sum_{C_{\text{shared}} \in C} |C_{\text{shared}}| \). We need \( O(N_C) \) space to store \( C \). According to MinHash [11], \( k = O(\log n) \) and \( L = O(n^\rho) \), where \( \rho = \ln(1/p_1)/\ln(1/p_2) \). Some data IDs may be appeared in multiple \( C_{\text{shared}} \)’s, but \( k = |C| \ll n \). Thus, SILK needs \( O(N_B D_B K + N_B D_B L) = O(n^{1+p}\log^2 n) \) time and \( O(N_B L + N_C) = O(n^{1+p}) \) space. As can be seen, the time and space complexities of SILK are independent of \( k \), and its time complexity is highly dependent on the total number of data IDs among the buckets, i.e., \( N_B D_B \).

**Data Assignment.** In this phase, we first take \( O(N_C d) \) time to compute the \( k \) centroids; then, we spend \( O(n dk) \) time and \( O(n) \) space to achieve the final clusters. As \( N_C = O(n) \), we need \( O(n d k) \) time and \( O(n) \) space for data assignment. Table 1 summarizes the time and space complexities of GEEK among the three types of data.\( ^3 \)

### 3.6 Discussion

Similar to k-means and its variants, GEEK is good at discovering compact clusters with a spherical shape. However, it may fail to find complex-shaped clusters like DBSCAN [20, 49], or form dendrogram to facilitate the understanding of these clusters like hierarchical methods such as BIRCH [58] and CURE [23].

Nevertheless, as discussed in Section 3.5, the time complexity of SILK is independent of \( k \). With this characteristic, we can first use GEEK to form a large number of microclusters and then use density-based methods to build on top of it to identify complex-shaped clusters. Moreover, we can build an index for the micro-clusters to speed up the NNS (e.g., FAISS [33]) and improve the scalability of other kinds of clustering methods such as DBSCAN and BIRCH. Thus, besides discovering the clusters, GEEK can be a fundamental tool to support and accelerate other clustering methods.

\( ^3 \)DType is short for data type. Homo, Hetero, and Sparse are short for the homogeneous dense data, heterogeneous dense data, and sparse data, respectively.
4 EXPERIMENTS

We study the performance of GEEK over five large-scale datasets. All experiments are conducted on a distributed CPU-GPU platform which consists of 2 computing nodes and 6 GPUs. Node 1 has an Intel(R) Xeon(R) Platinum 8170 CPU @ 2.10 GHz with 52 cores, 1 TB RAM, and 2 Nvidia GeForce GTX 1080 Ti’s with 11 GB memory. Node 2 has an Intel(R) Xeon(R) Gold 6130 CPU @ 2.10 GHz with 32 cores, 256 GB RAM, and 4 Nvidia GeForce GTX 1080 Ti’s with 11 GB memory. Both machines are running on CentOS 7.4.

4.1 Experiment Setup

Competitors. To the best of our knowledge, GEEK is the first generic framework that supports clustering over different types of data, so there is no direct competitor. Alternatively, we select different state-of-the-art GPU-based methods as its competitors for different types of data. Specifically, for the homogeneous dense data, we choose Lloyd [41], Yinyang [18], ¹ and FAISS [33] ² as baselines; for the heterogeneous dense dataset and the sparse data, we implement a GPU-based k-modes [30] as its competitor.

Datasets. We evaluate the performance of GEEK over five real-life datasets, i.e., Gist, Sift10M, Sift1B, GeoNames, and URL.³ For the homogeneous dense datasets Gist, Sift10M, and Sift1B, we use Euclidean distance, while for the heterogeneous dense dataset GeoNames and the sparse dataset URL, we use the distance (1 − Jaccard). The statistics of datasets are summarized in Table 2. Their details are described as follows.

- **Gist.** This homogeneous dense dataset has 10⁶ Gist image features, where each feature is a 960-dimensional vector.
- **Sift10M, Sift1B.** These two homogeneous dense datasets consist of 10⁷ and 10⁸ Sift features from images, respectively, where each feature is a 128-dimensional vector.
- **GeoNames.** It contains over 11 million geographical names. After data cleaning, we get a heterogeneous dense dataset with 9 attributes, i.e., latitude, longitude, feature class, feature code, country code, population, elevation, dem, timezone.
- **URL.** This sparse dataset collects 2.3 million URLs, where it has around 3.2 million features but with only around 116 non-zeroes for each data object [44]. We adopt DOPH [52] to reduce the dimensionality to 400.

**Evaluation Metrics.** As done in FAISS [33], one of the most frequent applications for clustering is to use the clusters for large-scale NNS. For this case, we do not know the ground truth of clusters. The compactness of clusters becomes a vital indicator to evaluate the effectiveness of a clustering method. We use the following evaluation metrics for performance evaluation.

- **Time.** We use the running time (or simply time) to evaluate the efficiency of a method. It is defined as the wall-clock time (in seconds) of a method for clustering.

- **Radius.** We use radius to evaluate the effectiveness of a method. Given a cluster of m data objects \( \{x_1, \ldots, x_m\} \) and the central vector \( c \), the radius is defined as \( \max_{i \in \{1, \ldots, m\}} \text{Dist}(x_i, c) \). A small radius indicates a compact cluster.

We report the mean results of these metrics over five times of repetition. The number of clusters in FAISS [33] is fixed because it splits the large clusters into two when \( k \) reduces. However, Lloyd [41], Yinyang [18], k-modes [30], and GEEK do not implement this operation. Their final number of clusters (i.e., \( k' \)) may be smaller than the pre-specified \( k \) value. Since the radius is highly sensitive to \( k' \), we focus on \( k' \) during the performance evaluation.

4.2 Parameters Studies

We first study the impacts of parameters of GEEK in a single GPU fashion. To be concise, we only show the results of GEEK over Sift10M, where \( B \) is determined by \( t \) and \( m \). Thus, we study the impacts of \( t, m, L, \) and \( \delta \). The results are depicted in Figure 4.

**Impact of \( t \).** We fix \( m = 40, K = 3, \delta = 10 \) and vary \( t \in \{5000, 10000, 20000, 30000\} \). We use different \( L \in \{10, 20, 30, 40\} \) to get different \( k' \)'s. As done in FAISS [33], one of the most frequent applications for clustering is to use the clusters for large-scale NNS. For this case, we do not know the ground truth of clusters. The compactness of clusters becomes a vital indicator to evaluate the effectiveness of a clustering method. We use the following evaluation metrics for performance evaluation.

**Impact of \( m \) and \( L \).** We first fix \( L = 20, K = 3, \delta = 10 \) and vary \( m \in \{20, 40, 60\} \). For different ranges of \( k' \), GEEK achieves its best performance under different \( m \)'s. For example, when \( k' > 11,000 \), GEEK obtains the best results under \( m = 60 \). When \( K = 3, \delta = 10 \) and vary \( L \in \{10, 20, 30, 40\} \). Similar to the impacts of \( t \) and \( m \), GEEK achieves its best performance under different \( L \)'s. As the increase of \( m \) or \( L \), more buckets will be generated for seeding and results in a larger \( k' \). Based on the above observations, we tune a combination of \( t, m, \) and \( L \) to get the best result of GEEK for different ranges of \( k' \).

**Impact of \( K \) and \( \delta \).** We first fix \( m = 40, L = 20, \delta = 10 \) and vary \( K \in \{2, 3, 4, 5\} \). Except \( K = 2 \), the results of GEEK under other settings of \( K \) are comparable to each other. Thus, the performance of GEEK is less sensitive to \( K \). Then, we fix \( m = 40, L = 20, K = 3 \) and vary \( \delta \in \{1, 10, 100\} \). Similar to the impacts of \( K \), the results of GEEK under different \( \delta \)'s are almost identical to each other. For simplicity, we set \( K = 3 \) and \( \delta = 10 \) by default.

**Summary.** In the subsequent experiments, we set \( K = 3 \) and \( \delta = 10 \) for GEEK; to remove the impact of parameters, we run GEEK on a grid search of \( t \in \{5000, 10000, 20000, 30000\} \), \( m \in \{20, 40, 60\} \),

| Table 2: Statistics of five real-life datasets |
|-----------------------------------------------|
| **Datasets** | **n** | **d** | **DType** | **Distance** |
|-----------------|--------|------|---------|-------------|
| Gist            | 1.0 × 10⁶ | 960  | Homo    | Euclidean   |
| Sift10M         | 1.0 × 10⁵  | 128  | Homo    | Euclidean   |
| Sift1B          | 1.0 × 10⁷  | 128  | Homo    | Euclidean   |
| GeoNames        | 1.1 × 10⁷  | 9    | Hetero  | Jaccard     |
| URL             | 2.3 × 10⁶  | 3.2 × 10⁶| Sparse  | Jaccard     |

¹ Lloyd and Yinyang are available at https://github.com/src-d/kmeans.
² FAISS is a very popular GPU-based library for efficient similarity search and clustering, its implementation is available at https://github.com/facebookresearch/faiss
³ Gist, Sift10M, and Sift1B can be downloaded from http://corpus-texmex.irisa.fr/.
⁴https://www.kaggle.com/geonames/geonames-database.
⁵https://archive.ics.uci.edu/ml/datasets/URL+Reputation.
and \( L \in \{10, 20, 30, 40\} \) and report its best results for different \( k^* \) values. For Yinyang and FAISS, we use the settings suggested by the authors \([18, 33]\) and set \( k \) from 100 to 20,000 to get different \( k^* \)'s if possible. Lloyd and \( k \)-modes do not require extra parameters.

4.3 Results and Analysis

We evaluate GEEK in terms of five aspects: the clustering performance, the scalability to massive datasets, the initial seeding performance and the performance on multi-GPUs and multi-nodes.

**Clustering Performance.** We first study the performance of GEEK under different \( k^* \) values. For the competitors, we report their least running time as soon as they converge. Since Yinyang only supports a single GPU, we run all methods on 1 GPU to make a fair comparison. The results are shown in Figure 5.

GEEK is slightly faster than FAISS over Gist and Sift10M, and compared with Lloyd and Yinyang, GEEK achieves at least one order of magnitude acceleration when \( k^* \gg 2,000 \). The reason is that the phases of data transformation and initial seeding of GEEK are independent of \( k \), and it conducts a one-pass data assignment only. Thus, GEEK becomes increasingly efficient as \( k \) increases. Moreover, GEEK enjoys a smaller radius than the competitors so that it can generate more compact clusters than the competitors. Therefore, GEEK outperforms Lloyd, Yinyang, and FAISS over these homogeneous dense datasets.

As can be seen from Figure 5, GEEK also enjoys at least one order of magnitude less running time than \( k \)-modes over GeoNames and URL when \( k^* \gg 5,000 \). The radius values of GEEK are close to (or slightly larger than) those of \( k \)-modes, which indicates that GEEK can generate clusters as compact as \( k \)-modes. Notice that GEEK is much more efficient than \( k \)-modes, and as will be shown later, GEEK can support multi-GPUs and multi-nodes. If the compactness is much more important than the efficiency, we can conduct several passes of data assignments for GEEK with multi-GPUs to get more compact clusters yet without losing efficiency. Thus, we can claim that GEEK outperforms \( k \)-modes over the heterogeneous dense dataset GeoNames and the sparse dataset URL.

Additionally, as \( k^* \) increases, the slope of GEEK is smooth like FAISS, which is much smaller than those of Lloyd, Yinyang, and \( k \)-modes. This observation further demonstrates that the running time of GEEK is not very sensitive to \( k^* \). Due to the large GPU memory usage, Yinyang cannot support clustering with large \( k^* \) values. Thus, the curves of Yinyang are relatively short.

**Scalability to Massive Datasets.** We then study the scalability of GEEK on a billion-scale dataset Sift1B. Since the performance of Lloyd and Yinyang are much worse than GEEK and FAISS, we only run GEEK and FAISS on a distributed CPU-GPU platform with 2 computing nodes, where each computing node uses 2 GPUs (2+2 GPUs). The results are displayed in Figure 5.
The radius values of GEEK are much smaller than those of GEEK, especially when $k^* > 5,000$. However, FAISS is 2~4 times faster than GEEK because FAISS uses a sampling method in its implementation to reduce the computational cost and space overhead. Given a specific $k$ value, FAISS uniformly draws a sample set of $(256 \cdot k)$ data to determine $k$ centroids. The sample size is much smaller than $10^9$, e.g., only 0.5% of Sift1B for a considerable $k$ value such as $k = 20,000$. Thus, FAISS enjoys better scalability than GEEK. However, using a sample set to deal with billion-scale data might be problematic. The sample set is merely a sketch representation of the full data. The centroids may concentrate on the high-density areas, leading to a large radius for clusters with data objects from low-density areas. In fact, the scalability of GEEK is still good enough for most applications. According to Figure 5, GEEK can provide compact clusters for $k^* \approx 20,000$ with less than 20 minutes.

**Initial Seeding Performance.** Next, we study the performance of SILK. We choose $k$-means++ [5] and the random selection (Random) as competitors. To make a fair comparison, we only record their running time of initial seeding; then, we use their seeds for a one-pass data assignment to evaluate the seeding performance. We show the results on 1 GPU over Gist and Sift10M in Figure 6. Similar trends can be observed from other datasets.

The radius values of SILK are much smaller than those of $k$-means++ and Random, which indicates that the initial seeds generated by SILK are well-selected and can be used to achieve more compact clusters than others. Even though Random enjoys the least running time, SILK requires less running time than $k$-means++, and its advantage is more apparent when $k^*$ increases. Moreover, the running time of $k$-means++ is linear to $k^*$, while that of SILK and Random is less sensitive to $k^*$. This phenomenon could be because the time complexity of both SILK and Random is independent of $k$, while $k$-means++ requires $O(k)$ iterations to generate the same number of seeds. Thus, SILK outperforms $k$-means++ and Random over Gist and Sift10M, especially in large $k^*$ values.

**Performance on Multi-GPUs.** To verify the effectiveness of the distributed implementation, we first study the performance of GEEK on multi-GPUs. To reduce the impact of communication overhead from different computing nodes, we run GEEK on a single node (i.e., Node 2) and vary the number of GPUs in a range of $\{1, 2, 4\}$. The results of GEEK over Sift10M are displayed in Figure 7. Similar trends can be observed from other datasets.

The acceleration of GEEK is asymptotically linear to the number of GPUs, which indicates that each GPU’s workload for the original data and intermediate data is balanced. Besides, the radius values of GEEK under different number of GPUs are close to each other. Thus, GEEK can provide stable, compact results on multi-GPUs.

**Performance on Multi-Nodes.** To this end, we study the performance of GEEK on multi-nodes. We consider three cases: (1) Node 1 uses one GPU only (1 GPU); (2) both Node 1 and Node 2 use one GPU (1+1 GPUs); (3) both Node 1 and Node 2 use two GPUs (2+2 GPUs). The results over Sift10M are depicted in Figure 7. Similar trends can be observed from other datasets.

Similar to the results on multi-GPUs, the acceleration of GEEK is asymptotically linear to the total number of GPUs, demonstrating the effectiveness of the distributed implementation in reducing the communication cost across different nodes. Since the radius values under different cases are close to each other, GEEK can provide stable, compact clusters on multi-nodes. Thus, no matter on a single node with multi-GPUs or multi-nodes with multi-GPUs, the distributed implementation of GEEK is efficient and effective.

## 5 RELATED WORK

Clustering is a fundamental problem with extensive studies. Well-known methods include partitioning [30, 41, 43, 45, 48], hierarchical [23, 58], density-based [4, 20, 22, 49] and grid-based methods [1, 50, 54]. A comprehensive survey can be referred to [53, 56]. Since GEEK is most related to partitioning methods such as $k$-means and its variants, we briefly discuss those methods.

$k$-means was first introduced by MaxQueen [45]. The most famous version is Lloyd’s algorithm [41], which randomly selects $k$ data objects as initial seeds and conducts the assign-update iteration with $O(ndk)$ time until the clusters converge. As data have

---

**Figure 6:** The results of different initial seeding methods over Gist and Sift10M

The radius values of GEEK are much smaller than those of GEEK, especially when $k^* > 5,000$. However, FAISS is 2~4 times faster than GEEK because FAISS uses a sampling method in its implementation to reduce the computational cost and space overhead. Given a specific $k$ value, FAISS uniformly draws a sample set of $(256 \cdot k)$ data to determine $k$ centroids. The sample size is much smaller than $10^9$, e.g., only 0.5% of Sift1B for a considerable $k$ value such as $k = 20,000$. Thus, FAISS enjoys better scalability than GEEK. However, using a sample set to deal with billion-scale data might be problematic. The sample set is merely a sketch representation of the full data. The centroids may concentrate on the high-density areas, leading to a large radius for clusters with data objects from low-density areas. In fact, the scalability of GEEK is still good enough for most applications. According to Figure 5, GEEK can provide compact clusters for $k^* \approx 20,000$ with less than 20 minutes.

**Figure 7:** The results of GEEK over Sift10M on multi-GPUs and multi-nodes
been rapidly increased, many variants have been proposed to improve Lloyd’s algorithm in initial seed selection, the support of large $d$ and $k$, and the acceleration of massive data.

As is well-known, a good selection of initial seeds is crucial for $k$-means to quickly converge. The classic remedy is to use $k$-means++ [5], but it requires $O(ndk^2)$ time and $O(k)$ iterations, which is hard to parallel. Bahmani et al. [8] proposed a parallel variant $k$-means++ and reduced the iterations to $O(\log k)$. To get a fast and provably good seeding algorithm, Bachem et al. [6] designed a Markov chain Monte Carlo sampling method to generate $k$ seeds from similar buckets. As distinct from $\Omega\left(\frac{k^2}{\epsilon^2}\right)$ to scale up large datasets with categorical values. In Proceedings of the Twenty-third International Conference on Knowledge Discovery and Data Mining, 2017.

Another direction to optimize $k$-means is to support large values of $d$ and $k$. Elkan [19] introduced a representative work which uses triangle inequality to avoid unnecessary distance computations. Algorithms of the same kind include Hamerly [25], Ding et al. [18], Newling et al. [46, 47], and Curtin [15]. Since we aim to support different types of data, we cannot leverage the geometric properties such as triangle inequality for acceleration. Instead, we apply GPUs and implement GEEK on a distributed CPU-GPU platform to speed up the distance computation for data assignment.

Recently, many researchers also consider using modern hardwares, such as GPUs [12, 35, 40] and many-core supercomputers [24, 38], to scale up $k$-means for massive datasets. However, they are highly optimized for a specific distance, e.g., Euclidean distance. In contrast, GEEK is a generic distributed clustering framework, which can handle large-scale data with many distance functions.

6 CONCLUSION

In this paper, we introduce a novel generic distributed clustering framework GEEK to process massive data of three data types. We also propose a new seeding method SILK to efficiently generate seeds from similar buckets. As distinct from $k$-means++ and its variants, SILK does not need to pre-specify $k$, and its time complexity is independent of $k$. With this advantage, GEEK can also be a fundamental tool to support and accelerate other clustering methods. We design and implement GEEK on a distributed CPU-GPU platform. Extensive experiments over five large-scale datasets demonstrate the superior performance of GEEK, especially when $k$ is large.

REFERENCES

[1] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhanjan Raghavan. 1998. Automatic subspace clustering of high dimensional data for data mining applications. In SIGMOD. 94–105.

[2] Alexandr Andoni and Piotr Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In FOCS. 459–468.

[3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. In NeurIPS. 1225–1233.

[4] Michael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: ordering points to identify the clustering structure. SIGMOD 28, 2 (1999), 49–60.

[5] David Arthur and Sergei Vassilvitskii. 2007. k-means++. The advantages of careful seeding. In SODA. 1027–1035.

[6] Olivier Bachem, Mario Lucic, Hamed Hassani, and Andreas Krause. 2016. Fast and provably good seedings for $k$-means. In NeurIPS. 55–63.

[7] Olivier Bachem, Mario Lucic, and Andreas Krause. 2017. Distributed and provably good seedings for $k$-means in constant rounds. In ICML. 292–300.

[8] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Scalable k-means++. PVLDB 5, 7 (2012), 622–633.

[9] Aditya Bhaskara and Maheshkalya Wijewardena. 2018. Distributed clustering via ish based data partitioning. In ICML. 570–579.

[10] Andrei Z Broder. 1997. On the resemblance and containment of documents. In Proceedings of Compression and Complexity of Sequences. 21–29.

[11] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. 1998. Min-wise independent permutations. In STOC. 327–336.

[12] Feng Cao, Anthony KH Tung, and Aoying Zhou. 2006. Scalable clustering using graphics processors. In WAVIS. 372–384.

[13] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In STOC. 380–388.

[14] Vincent Cohen-Addad, Silvio Lattanzi, Ashkan Norouzi-Fard, Christian Sohler, and Ola Svensson. 2020. Fast and Accurate k-means++ via Rejection Sampling. In NeurIPS 33 (2020).

[15] Ryan R Curtin. 2017. A dual-tree algorithm for fast k-means clustering with large k. In SDM. 309–318.

[16] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In SoCG. 253–262.

[17] Fabrizio Olivetti de Franca. 2016. A hash-based co-clustering algorithm for categorical data. Expert Systems with Applications 64 (2016), 24–35.

[18] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Yin yang k-means: A drop-in replacement of the classic k-means with consistent speedup. In ICML. 579–587.

[19] Charles Elkan. 2003. Using the triangle inequality to accelerate k-means. In ICM. 147–153.

[20] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In KDD. Vol. 96. 226–231.

[21] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfried Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In SIGMOD. 541–552.

[22] Junhao Gan and Yufei Tao. 2015. DISCAN revisited: Mis-clou, un-fixability, and approximation. In SIGMOD. 519–530.

[23] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. 1998. CUBE: an efficient clustering algorithm for large databases. In SIGMOD. Vol. 27. 73–84.

[24] Ali Hadadian and Sareed Shahrivar. 2014. High performance parallel k-means clustering for disk-resident datasets on multi-core CPUs. The Journal of Supercomputing 69, 2 (2014), 845–863.

[25] Greg Hamerly. 2010. Making k-means even faster. In SDM. 130–140.

[26] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. 2012. Approximate nearest neighbor in fixed dimension. In STOC. 570–579.

[27] Qiang Huang, Jianlin Feng, Qiong Fang, Wilfred Ng, and Wei Wang. 2017. Query-aware locality-sensitive hashing scheme based on p-stable distributions. In SDM. 300–308.

[28] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-aware locality-sensitive hashing for approximate nearest neighbor search. PVLDB 9, 1 (2015), 1–12.

[29] Qiang Huang, Guihong Ma, Jianlin Feng, Qiong Fang, and Anthony KH Tung. 2018. Accurate and Fast Asymmetric Locality-Sensitive Hashing Scheme for Maximum Inner Product Search. In KDD. 1561–1570.

[30] Zhexue Huang. 1998. Extensions to the k-means algorithm for clustering large data sets with categorical values. DMKD 2, 3 (1998), 283–304.

[31] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In STOC. 604–613.

[32] Wen Jin, Anthony KH Tung, and Jiaswei Han. 2001. Mining top-n local outliers in large databases. In KDD. 293–298.

[33] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. IEEE Transactions on Big Data (2019).

[34] Hisashi Koga, Tetsuo Ishibashi, and Toshinori Watanabe. 2007. Fast agglomerative hierarchical clustering algorithm using Locality-Sensitive Hashing. KAIS 12, 1 (2007), 25–53.

[35] Kui J Kolhoff, Vijay S Pande, and Russ B Altman. 2013. K-Means for Parallel Architecture . Using All-Prefix-Sum Sorting and Updating Steps. IEEE Transactions on Parallel and Distributed Systems 24, 8 (2013), 1602–1612.

[36] Yifan Lei, Qiang Huang, Mohan Kankanhalli, and Anthony Tung. 2019. Sublinear Time Nearest Neighbor Search over Generalized Weighted Space. In ICML. 3773–3781.

[37] Yifan Lei, Qiang Huang, Mohan Kankanhalli, and Anthony KH Tung. 2020. Locality-Sensitive Hashing Scheme based on Longest Circular Co-Substring. In SIGMOD 2509–2539.
A Generic Distributed Clustering Framework for Massive Data

[38] Liandeng Li, Teng Yu, Wenlai Zhao, Haohuan Fu, Chenyu Wang, Li Tan, Guangwen Yang, and John Thomson. 2018. Large-scale hierarchical k-means for heterogeneous many-core supercomputers. In International Conference for High Performance Computing, Networking, Storage and Analysis. 160–170.

[39] Quanhong Li, Peng Wang, Wei Wang, Hao Hu, Zhongsheng Li, and Junxian Li. 2014. An efficient K-means clustering algorithm on MapReduce. In DASFAA. 357–371.

[40] You Li, Kaiyong Zhao, Xiaowen Chu, and Liming Liu. 2013. Speeding up k-means algorithm by gpus. J. Comput. System Sci. 79, 2 (2013), 129–137.

[41] Stuart Lloyd. 1982. Least squares quantization in PCM. IEEE Transactions on Information Theory 28, 2 (1982), 129–137.

[42] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. 2020. VHP: Approximate Nearest Neighbor Search via Virtual Hypersphere Partitioning. PVLDB 13, 9 (2020), 1443–1455.

[43] Carlos B Lucasius, Adrie D Duin, and Gerrit Kateman. 1993. On k-medoid clustering of large data sets with the aid of a genetic algorithm: background, feasibility and comparison. Analytica Chimica Acta 282, 3 (1993), 647–669.

[44] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. 2009. Identifying suspicious URLs: an application of large-scale online learning. In ICML. 681–688.

[45] James MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Vol. 1. 281–297.

[46] James Newling and François Fleuret. 2016. Fast k-means with accurate bounds. In ICML. 936–944.

[47] James Newling and François Fleuret. 2016. Nested mini-batch k-means. In NeurIPS. 1352–1360.

[48] Raymond T. Ng and Jiawei Han. 2002. CLARANS: A method for clustering objects for spatial data mining. TKDE 14, 5 (2002), 1003–1016.

[49] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. TODS 42, 3 (2017), 19.

[50] Gholamhossein Sheikholeslami, Surojit Chatterjee, and Amlong Zhang. 1998. Wavecluster: A multi-resolution clustering approach for very large spatial databases. In VLDB. Vol. 98. 428–439.

[51] Anshumali Shrivastava and Ping Li. 2014. Asymmetric LSH (ALSH) for sublinear time Maximum Inner Product Search (MIPS). In NeurIPS. 2321–2329.

[52] Anshumali Shrivastava and Ping Li. 2014. Densifying one permutation hashing via rotation for fast near neighbor search. In ICML. 557–565.

[53] Anthony K. H. Tung, Jiawei Han, and Micheline Kamber. 2001. Spatial clustering methods in data mining: A survey. Geographic data mining and knowledge discovery (2001), 188–217.

[54] Wei Wang, Jiong Yang, and Richard Muntz. 1997. STING: A statistical information grid approach to spatial data mining. In VLDB. Vol. 97. 186–195.

[55] Yiqiu Wang, Anshumali Shrivastava, Jonathan Wang, and Junhong Ryu. 2018. Randomized algorithms accelerated over cpu-gpu for ultra-high dimensional similarity search. In SIGMOD. 889–903.

[56] Dongkuan Xu and Yingjie Tian. 2015. A comprehensive survey of clustering algorithms. Annals of Data Science 2, 2 (2015), 165–193.

[57] Juan Zamora, Marcelo Mendoza, and Héctor Allende. 2016. Hashing-based clustering in high dimensional data. Expert Systems with Applications 62 (2016), 202–211.

[58] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: an efficient data clustering method for very large databases. In SIGMOD. Vol. 25. 103–114.

[59] Yanfeng Zhang, Shimin Chen, and Ge Yu. 2016. Efficient distributed density peaks for clustering large data sets in mapreduce. TKDE 28, 12 (2016), 3218–3230.

[60] Bolong Zheng, Xi Zhao, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S Jensen. 2020. PM-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search. PVLDB 13, 5 (2020), 643–655.