A New Sparse Data Clustering Method Based On Frequent Items

QIANG HUANG*, National University of Singapore, Singapore
PINGYI LUO*, National University of Singapore, Singapore
ANTHONY K. H. TUNG*, National University of Singapore, Singapore

Large, sparse categorical data is a natural way to represent complex data like sequences, trees, and graphs. Such data is prevalent in many applications, e.g., Criteo released a terabyte size click log data of 4 billion records with millions of dimensions. While most existing clustering algorithms like k-Means work well on dense, numerical data, there exist relatively few algorithms that can cluster sets of sparse categorical features.

In this paper, we propose a new method called k-FreqItems that performs scalable clustering over high-dimensional, sparse data. To make clustering results easily interpretable, k-FreqItems is built upon a novel sparse center representation called FreqItem which will choose a set of high-frequency, non-zero dimensions to represent the cluster. Unlike most existing clustering algorithms, which adopt Euclidean distance as the similarity measure, k-FreqItems uses the popular Jaccard distance for comparing sets.

Since the efficiency and effectiveness of k-FreqItems are highly dependent on an initial set of representative seeds, we introduce a new randomized initialization method, SILK, to deal with the seeding problem of k-FreqItems. SILK uses locality-sensitive hash (LSH) functions for oversampling and identifies frequently co-occurred data in LSH buckets to determine a set of promising seeds, allowing k-FreqItems to converge swiftly in an iterative process. Experimental results over seven real-world sparse data sets show that the SILK seeding is around 1.1~3.2× faster yet more effective than the state-of-the-art seeding methods. Notably, SILK scales up well to a billion data objects on a commodity machine with 4 GPUs. The code is available at https://github.com/HuangQiang/k-FreqItems.

CCS Concepts: • **Information systems** → **Clustering**; • **Theory of computation** → *Unsupervised learning and clustering*.

Additional Key Words and Phrases: Sparse Data Clustering, Frequent Items, Jaccard distance, Seeding, MinHash, Locality-Sensitive Hashing

ACM Reference Format:
Qiang Huang, Pingyi Luo, and Anthony K. H. Tung. 2023. A New Sparse Data Clustering Method Based On Frequent Items. *Proc. ACM Manag. Data* 1, 1, Article 5 (May 2023), 28 pages. https://doi.org/10.1145/3588685

1 INTRODUCTION

Unstructured or semi-structured data in the form of sequences (e.g., text), trees, and graphs are often represented as a set of categorical features resulting in sparse data sets with enormous cardinality and ultra-high dimensionality. For example, Criteo released a terabyte size sparse data set in 2013,[1]

*All authors contributed equally to this research and were ordered alphabetically by their last names.
[1]https://labs.criteo.com/2013/12/download-terabyte-click-logs/.

Authors' addresses: Qiang Huang, huangq@comp.nus.edu.sg, National University of Singapore, Singapore; Pingyi Luo, pingyi@u.nus.edu, National University of Singapore, Singapore; Anthony K. H. Tung, atung@comp.nus.edu.sg, National University of Singapore, Singapore.

2836-6573/2023/5-ART5
https://doi.org/10.1145/3588685

which has around 4 billion records, and each record is represented as a sparse binary vector with millions of dimensions, but only about 39 non-zero values on average [77].

Data clustering is a fundamental problem in data management and data mining. While there exist many clustering algorithms for dense, numerical data [6, 31, 34, 35, 82, 86, 100], they cannot be easily adapted to handle sparse data sets since:

(1) They tend to use similarity functions for numerical data (with Euclidean distance being the most common one), which is not suitable for comparing sparse data sets.
(2) While there are dimension reduction and feature embedding methods [2, 64] that can convert sparse data into a dense format of lower dimensionality, they are typically done as a pre-processing step before the clusters are identified. Since, for ultra-high dimensional data sets, data in different clusters can be similar to each other in completely different subsets of dimensions, such approaches can result in a large amount of information loss [64, 97]. Furthermore, the cluster representatives that are found in such approaches are not easily interpretable due to the data transformation.
(3) Due to their choice of similarity functions, they often have to represent clusters using the full set of dimensions which is prohibitively expensive for ultra-high dimensional sparse data sets. For subspace clustering algorithms [16, 21, 51, 95, 96, 98] which try to simultaneously determine cluster memberships and important features for each cluster, using dense numerical cluster representatives with millions of dimensions also incurs many errors when determining initial cluster membership due to a large number of noisy dimensions.

To address the challenges we have highlighted, in this paper, we look at the problem of performing partitioning-based clustering algorithms [23, 29, 30, 38, 47, 66, 75, 80, 81] over sparse data with enormous cardinality and ultra-high dimensionality. Our choice is motivated by the fact that many of these algorithms, such as k-Means [66, 75] and k-Modes [47], have linear-time complexity with respect to both cardinality and dimensionality while producing cluster representatives that are simple and easy to interpret. Furthermore, their simple, iterative nature provides many opportunities for utilizing modern hardware, such as GPUs, to accelerate the execution of the clustering operation [19, 63].

Given that our focus is on sets of sparse categorical features, we adopt Jaccard distance as the metric for data clustering, instead of using the commonly used Euclidean distance in numeric data clustering algorithms. Jaccard distance is a widely used and versatile metric for set data, and has been successfully applied in sparse data clustering scenarios, such as in the work of Lu et al. [70] for scalable news recommendation.

In addition to choosing an appropriate distance function, the representation of cluster centers is another crucial aspect to consider when clustering sparse data. In this paper, we propose a novel sparse center representation called FreqItem, specifically designed for set data. Our approach draws inspiration from k-Modes, a commonly used method for categorical data, but we relax its condition by selecting a set of high-frequency, non-zero dimensions to effectively represent the cluster center in the context of sparse data clustering. Our experiments demonstrate that FreqItem outperforms other commonly used center representations for set data, such as Mode [47] and Medoid [52, 71], in terms of achieving a smaller sum of squared distances between data points and their closest centers. In addition, FreqItem exhibits a computation time that scales linearly with the cluster size, similar to other widely used center representations such as Centroid and Mode in existing partitioning-based algorithms.

With the FreqItem representation, we introduce a new method named k-FreqItems for sparse data clustering based on Jaccard distance. k-FreqItems adopts the iterative refinement paradigm of the k partitioning algorithms to cluster the set data in two aspects: (1) it uses Jaccard distance to

evaluate the closeness of set data; (2) it adopts the FreqItem to represent the cluster center. The time complexity of k-FreqItems in each iteration is $O(n\bar{d}k)$, where n is the data cardinality and $\bar{d}$ is the average number of non-zero dimensions.

As a second step, we tackle the seeding problem of k-FreqItems. Like most iterative refinement clustering algorithms, the seeding step of finding k initial cluster centers is crucial for the performance of k-FreqItems since having well-selected seeds can speed up the convergence and give better clustering results [7, 11]. With the FreqItem representation, some existing seeding methods (with theoretical guarantees) [7, 11, 84] can be adapted for k-FreqItems to provide favorable initial cluster centers. The primary downside is their inherently sequential nature. For example, k-Means++ [7, 84] requires k sequential iterations through the data set to produce k initial centers. k-Means‖ [11] reduces the sequential iterations from k to $O(\log n)$ by an overseeding strategy, but both methods rely on the information from past iterations to select seeds in the next iterations, restricting their scalability to massive data sets.

Locality-Sensitive Hashing (LSH) [39, 50] is a widely used approximation technique in many large-scale clustering methods, especially for clustering acceleration [54, 62, 76] and data partitioning [15, 102]. The locality-sensitive property of hash functions in LSH allows for the creation of buckets that often contain similar data points from the same cluster, making them suitable for providing effective seeds for quick convergence. Nevertheless, despite its potential, there has been limited research on leveraging LSH for initial seeding in (sparse data) clustering algorithms.

With this insight, we propose a new Seeding method based on simILar bucKets (SILK). Motivated by k-Means‖, we introduce an overseeding step that utilizes the locality-sensitive property of MinHash functions [18] for oversampling and selects the frequently co-occurred data from similar buckets to obtain good initial centers. With LSH for oversampling, we remedy the sequential issue in k-Means++ and k-Means‖ seeding and empirically justify that SILK can converge rapidly with the well-selected seeds. At the implementation level, we exploit the parallelism with GPUs and implement SILK on a distributed platform for large-scale clustering. In the experiments, we adopt two state-of-the-art methods (i.e., k-Means++ and k-Means‖) and four hybrid approaches (embedding with conventional clustering) as baselines to make a systematic comparison with SILK. Extensive results over five large-scale sparse data sets and two small ones with ground truth labels show that the SILK seeding is around 1.1~3.2× faster yet more effective than k-Means++ and k-Means‖ seeding, especially for large k values.

The rest of this paper is organized as follows. Section 2 reviews the preliminary knowledge. Section 3 introduces k-FreqItems for sparse data clustering. We present a new seeding technique SILK for set data in Section 4. Experimental results are reported and analyzed in Section 5. Section 6 surveys related work. Finally, we conclude our work in Section 7.

2 PRELIMINARIES

Before introducing k-FreqItems and SILK, we review the k-Means problem and the *Lloyd's* algorithm; then, we present two famous k-Means variants k-Means++ and k-Means‖ for initial seeding.

Suppose $\mathcal{X}$ is a set of dense, numerical data represented as vectors in a d-dimensional Euclidean space $\mathbb{R}^d$. Let $\|x - y\|$ be the Euclidean distance for any $x, y \in \mathbb{R}^d$. For any set $C \subset \mathbb{R}^d$ and $x \in \mathcal{X}$,

$$D(x, C)^2 = \min_{c \in C} \|x - c\|^2. \tag{1}$$

Definition 1 (k-Means Problem)**:** *Given an integer k and a data set $\mathcal{X} \subset \mathbb{R}^d$, the k-Means problem aims to find a set of k cluster centers $C \subset \mathbb{R}^d$ minimizing the sum of the squared distances $\phi_{\mathcal{X}}(C)$, i.e.,*

$$\phi_{\mathcal{X}}(C) = \sum_{x \in \mathcal{X}} D(x, C)^2 = \sum_{x \in \mathcal{X}} \min_{c \in C} \|x - c\|^2. \tag{2}$$

Algorithm 1: k-Means++ Seeding

Input: data set $\mathcal{X}$, weight vector w, # clusters k;

$C \leftarrow$ Select a data point $x \in \mathcal{X}$ with probability $\frac{w_x}{\sum_{x' \in \mathcal{X}} w_{x'}}$;

for $i = 2$ **to** k **do**

  Select $x \in \mathcal{X}$ with probability $\frac{w_x D(x,C)^2}{\sum_{x' \in \mathcal{X}} w_{x'} D(x',C)^2}$;

  $C \leftarrow C \cup \{x\}$;

return C;

Algorithm 2: k-Means‖ Overseeding

Input: data set $\mathcal{X}$, # rounds r, oversampling factor l;

$C \leftarrow$ Select a data point $x \in \mathcal{X}$ uniformly at random from $\mathcal{X}$;

for $i = 1$ **to** r **do**

  $\mathcal{S} \leftarrow$ Add $x \in \mathcal{X}$ with probability $\min(1, \frac{l \cdot D(x,C)^2}{\phi_{\mathcal{X}}(C)})$;

  $C \leftarrow C \cup \mathcal{S}$;

return C;

Given a subset $X \subset \mathcal{X}$ as a cluster, we usually use Centroid (i.e., the arithmetic mean position of all data points in X) to represent the cluster center c for Euclidean distance, i.e., $c = \frac{1}{|X|} \sum_{x \in X} x$.

***Lloyd's* Algorithm.** The vanilla k-Means is the *Lloyd's* algorithm [66], which starts with a random set of k distinct data as the cluster centers. In each iteration, it first assigns data to their closest cluster center; then, it updates the centers (i.e., Centroids) based on the data in their clusters. The iteration is repeated until the centers are no longer changed. This iteration is also known as the *Lloyd's* iteration, which can be done in $O(ndk)$ time.

The *Lloyd's* algorithm has been widely used because it is simple, efficient, and easy to parallel. Unfortunately, it is guaranteed only to achieve a local optimum. Finding the optimal solution $\phi_{OPT}(\mathcal{X})$ to the k-Means problem (even with $k = 2$) is NP-hard [24].

k-Means++. To obtain a near-optimal solution, k-Means++ has been proposed with a carefully designed seeding step based on the D^2-sampling strategy [7, 84]. Given a data set $\mathcal{X}$ and any set of cluster centers $C \subset \mathcal{X}$, the D^2-sampling strategy selects a data point $x \in \mathcal{X}$ as a new center with probability as follows

$$p(x) = \frac{D(x,C)^2}{\phi_{\mathcal{X}}(C)} = \frac{D(x,C)^2}{\sum_{x' \in \mathcal{X}} D(x',C)^2}. \tag{3}$$

We follow [10] and describe the seeding step of k-Means++ for the possibly weighted data in Algorithm 1. Let C be the cluster centers found so far. It starts by randomly selecting an $x \in \mathcal{X}$ as the first center; then, it sequentially chooses the remaining $(k-1)$ centers by D^2-sampling and incrementally updates C and $D(x,C)$.

Arthur and Vassilvitskii [7] demonstrate that the sum of the squared distances $\phi_{\mathcal{X}}(C)$ of k-Means++ seeding is bounded in expectation by $E[\phi_{\mathcal{X}}(C)] \leq 8(\log_2 k + 2) \cdot \phi_{OPT}(\mathcal{X})$. Note that k-Means++ seeding takes $\Theta(ndk)$ time only, which is linear to n and d. However, it requires exact k sequential iterations through the data, limiting its data scalability.

k-Means‖. To tackle large-scale data sets, Bahmani et al. [11] design an overseeding step to reduce the number of sequential iterations, which is depicted in Algorithm 2. Let C be a set of initial centers. It draws a data point $x \in \mathcal{X}$ uniformly at random as the first center. Then, it proceeds in r

Algorithm 3: k-Means|| Seeding

Input: data set $\mathcal{X}$, k, # rounds r, oversampling factor l;

$C \leftarrow k$-Means|| Overseeding$(\mathcal{X}, r, l)$;

foreach $c \in C$ **do**

  Assign $x \in \mathcal{X}$ to X_c such that $c = \arg\min_{c' \in C} \|x - c'\|$;

  $w_c \leftarrow |X_c|$;

$C' \leftarrow k$-Means++ Seeding(C, w, k);

return C';

sequential rounds, where in each round, it adds $x \in \mathcal{X}$ into C with probability $\min(1, \frac{l \cdot D(x,C)^2}{\phi_{\mathcal{X}}(C)})$, where l is called the oversampling factor. As such, one can select $|C| = r \cdot l + 1$ centers in expectation, which is usually larger than k.

To reduce $|C|$ to k, k-Means|| further assigns weights to the centers $c \in C$ based on the number of data in their clusters; then, it calls k-Means++ seeding to re-cluster the weighted centers to determine the final k centers. As $|C| \ll |\mathcal{X}|$, the re-clustering can be done efficiently. The complete k-Means|| seeding is displayed in Algorithm 3, where the time complexity is $O(ndrl)$. In practice, r is often set to $O(\log n)$, preferably even constant [10, 11]. As the number of rounds (i.e., synchronizations) is largely reduced, Algorithm 2 is suitable for a distributed setting [78].

3 K-FREQITEMS

We now present k-FreqItems for sparse data clustering. We first illustrate some notations used throughout the rest of this paper in Subsection 3.1. We then define a new center representation called FreqItem and validate its effectiveness in Subsection 3.2. Finally, we present the details of k-FreqItems in Subsection 3.3.

3.1 Notations

In the following sections, we focus on the set data for Jaccard distance. Suppose the set data are one-hot encoded as binary vectors[2] in a d-dimensional Hamming space $\{0, 1\}^d$. To be concise, we continue to use $\mathcal{X}$ to denote a set of binary vectors. Let $\bar{d}$ be the number of average non-zero dimensions of $\mathcal{X}$, and we assume $\bar{d} \ll d$. For any two $x, y \in \{0, 1\}^d$, the Jaccard distance is defined as follows:

$$J(x, y) = 1 - \frac{\langle x, y \rangle}{|x| + |y| - \langle x, y \rangle} = \frac{|x| + |y| - 2\langle x, y \rangle}{|x| + |y| - \langle x, y \rangle}, \tag{4}$$

where $|x|$ is the number of non-zero values in x; $\langle x, y \rangle$ is the inner product of x and y. For ease of reference, from now on we use the set data and binary vectors interchangeably.

3.2 FreqItem: A New Center Representation

First of all, the formulation of the k-Means problem in Section 2 is valid for the set data. However, the dense center representation of k-Means (e.g., Centroid) might not be suitable for sparse data.

To leverage the simplicity and efficiency of the k-Means paradigm, it is vital to define a suitable sparse center representation for the set data. Given a subset $X \subset \mathcal{X}$ as a cluster, according to Definition 1, the ideal cluster center c^* for Jaccard distance should minimize the sum of the squared

[2]We do not store the set data in a binary vector format. In practice, we only store their non-zero dimensions as we focus on the ultra-high dimensional, sparse data.

distances for all $x \in X$, i.e.,

$$c^* = \arg\min_{c \in \{0,1\}^d} \sum_{x \in X} J(x, c)^2. \tag{5}$$

Challenges. There are up to 2^d solutions for the ideal center, which is hard to determine. Thus, we consider two alternative options to represent the cluster center for the set data. One is to use Medoid. Given a set of n binary vectors, Medoid is the vector that has the minimum sum of the squared distances to the remaining vectors. The advantage of Medoid is that the center comes from the data. Hence, we do not need to design a different method to compute the center. Nonetheless, the time complexity to determine Medoid is $O(n^2\bar{d})$, limiting its scalability to tackle massive data sets.

As binary vectors can be regarded as categorical data, another option is to develop different variants of Mode as the cluster center. Depending on what the dimensions represent, we have

(1) Mode1: if we consider each dimension as a binary attribute (i.e., 0 and 1), a center can be defined by the dimensions whose frequency is at least 1;
(2) Mode2: if we treat each dimension as a value in a multivalued attribute, a center can be defined by the dimension(s) with the maximum frequency.

Whether using Mode1 or Mode2, one can determine the center in $O(n\bar{d})$ time, which is much more efficient than using Medoid. Nevertheless, its effectiveness might be far from the ideal center c^*, which will be validated later in this subsection.

FreqItem Representation. Intuitively, one can design different center representations for different kinds of data to approach the ideal center. The Mode might be suitable with the concern of its $O(n\bar{d})$ efficiency. Nonetheless, it is too strict to consider all non-zero dimensions (Mode1) or the dimension(s) with the maximum frequency (Mode2). We now define a new center representation named *FreqItem*, which is inspired by Mode, but we relax the condition of the non-zero dimensions with an extra parameter α.

Definition 2 (FreqItem)**:** *Given a cluster $X \subset \mathcal{X}$ and a parameter α ($0 \le \alpha \le 1$), the cluster center c is represented by the frequent items (FreqItem), i.e., the non-zero dimensions whose frequency is at least* $\max(1, \lceil \alpha \cdot F \rceil)$*, where F is the maximum frequency among them.*

Example 1: We now use an example to illustrate how to compute FreqItem. Suppose $\alpha = 0.4$ and $X = \{x_1, x_2, x_3, x_4\}$ is a cluster with four data, where x_1 = 10101 00000, x_2 = 10100 00100, x_3 = 10000 10100, and x_4 = 10000 00101. We first retrieve and scan the non-zero dimensions from the data in X and count their frequency, i.e., $(d_1, 4), (d_3, 2), (d_5, 1), (d_6, 1), (d_8, 3), (d_{10}, 1)$, where d_i is the i-th dimensions. Thus, the maximum frequency is $F = 4$ and the filtering threshold is $\max(1, \lceil \alpha \cdot F \rceil) = 2$. Based on Definition 2, we determine the non-zero dimensions $\{d_1, d_3, d_8\}$ as the frequent items, and the FreqItem representation is c = 10100 00100. △

We use α to control the sparsity of FreqItem. According to Definition 2, the FreqItem is Mode1 if $\alpha = 0$ (the extremely dense case of FreqItem); and it is Mode2 if $\alpha = 1$ (the extremely sparse case of FreqItem). Like the time to compute Mode, the time to compute FreqItem is also $O(n\bar{d})$, which is more efficient than the time to determine Medoid. However, there remains a problem for FreqItem: how to set a proper value of α?

Choice of α. According to Equation 4, if we set $\alpha \to 1$, as $\langle x, c \rangle$ will be close to 0, i.e., $\langle x, c \rangle \to 0$, then $J(x, c) \to 1$, which is contradicted to Equation 5; if we set $\alpha \to 0$, since $|c| \gg |x|$, then $J(x, c) \to 1$, also conflict to Equation 5. Therefore, it might be tricky to find a suitable value of α.

To further analyse the impact of α, we rewrite Equation 5 as below:

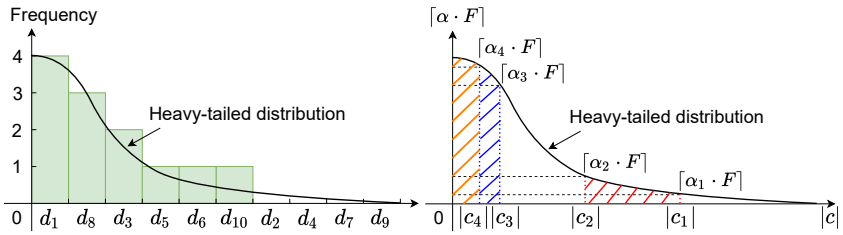
$$
\begin{aligned}
c^* &= \arg\min_{c\in\{0,1\}^d} \sum_{x\in X} \left(\frac{|x| + |c| - 2\langle x, c\rangle}{|x| + |c| - \langle x, c\rangle} \right)^2 \\
&= \arg\min_{c\in\{0,1\}^d} \sum_{x\in X} \left(2 - \frac{|x| + |c|}{|x| + |c| - \langle x, c\rangle} \right)^2 \\
&= \arg\max_{c\in\{0,1\}^d} \sum_{x\in X} \left(\frac{\langle x, c\rangle}{|x| + |c|} \right)^2 .
\end{aligned}
$$

Let $s = \sum_{x\in X} (\frac{\langle x,c\rangle}{|x|+|c|})^2$. According to Definition 2, the filtering threshold $\max(1, \lceil \alpha \cdot F \rceil)$ increases as α increases. As such, the number of non-zero dimensions of the FreqItem c is reduced, and hence the number of identical non-zero dimensions between c and x is also reduced. Thus, both $|c|$ and $\langle x, c\rangle$ decreases as α increases. Nevertheless, the relationship between s and α is still not clear.

Fortunately, given a cluster of sparse data, the frequency of different dimensions is usually quite different. For example, for the cluster $X = \{x_1, x_2, x_3, x_4\}$ shown in Example 1, only a few dimensions contain a high frequency (e.g., 3 or 4), while most dimensions only have a shallow frequency (e.g., 0 or 1). We sort the frequency of all dimensions in descending order and plot the histogram in Figure 1(a). As can be seen, the corresponding histogram asymptotically satisfies the heavy-tailed distribution.

Suppose the histogram of the frequency of the sorted dimensions (descending order) follows a heavy-tailed distribution. Based on Definition 2, we can also use this heavy-tailed distribution to represent the curve of $\lceil \alpha \cdot F \rceil$ vs. $|c|$. As depicted in Figure 1(b), given a parameter α ($0 \le \alpha \le 1$) and its corresponding FreqItem c, the y-axis of the heavy-tailed distribution is $\lceil \alpha \cdot F \rceil$, and according to Figure 1(a), the x-axis can be considered as the number of non-zero dimensions of c whose frequency is at least $\lceil \alpha \cdot F \rceil$, i.e., $|c|$.

Moreover, the cumulative distribution function (CDF) of this heavy-tailed distribution is correlated to $\langle x, c\rangle$. As shown in Figure 1(b), as α decreases (e.g., $\alpha_4 \rightarrow \alpha_3$), $|c|$ increases (e.g., $|c_4| \rightarrow |c_3|$), and hence the CDF of this heavy-tailed distribution increases (e.g., orange area $\rightarrow$ orange area + blue area); since the FreqItem c contains more non-zero dimensions, $\langle x, c\rangle$ also increases. Thus, this heavy-tailed distribution *simultaneously* represents the relationships of $|c|$ vs. α and $\langle x, c\rangle$ vs. α. We can use this heavy-tailed distribution to study the relationship between s and α.



(a) Heavy-tailed distribution (b) The curve of $\lceil \alpha \cdot F \rceil$ vs. $|c|$

Fig. 1. An illustration of the choice of α.

From Figure 1(b), with the same increment of α, i.e., $0 < \alpha_1 < \alpha_2 < \alpha_3 < \alpha_4 < 1$ and $\alpha_2 - \alpha_1 = \alpha_4 - \alpha_3$, we have two interesting observations: (1) $|c_1| - |c_2| > |c_3| - |c_4|$, which indicates that $|c|$ decreases significantly when α is small, and $|c|$ decreases very gently when α is large. (2) As the

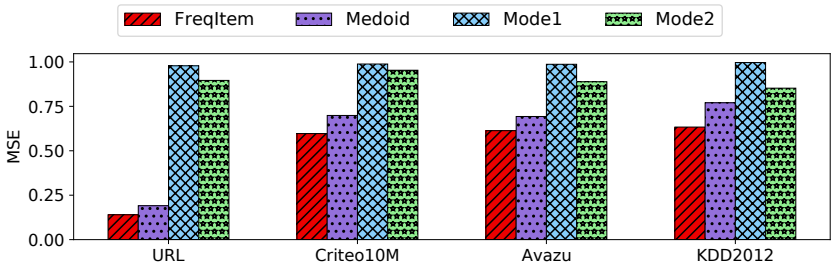


Fig. 2. The MSEs of four center representations.

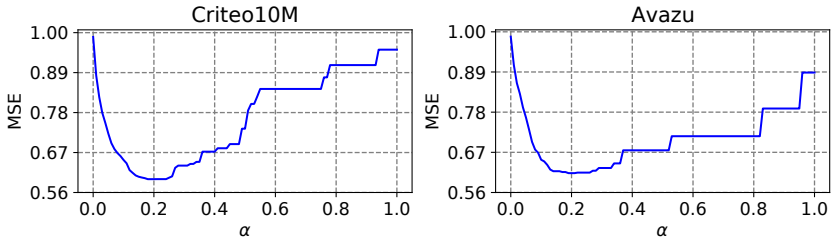


Fig. 3. The curves of MSE vs. α.

blue area is larger than the red area, we have $\langle x, c_3\rangle - \langle x, c_4\rangle > \langle x, c_1\rangle - \langle x, c_2\rangle$, which means that $\langle x, c\rangle$ decreases very gently when α is small, and $\langle x, c\rangle$ decreases significantly when α is large.

Based on these two observations, we can infer that: (1) When α is small, as $|c|$ decreases significantly but $\langle x, c\rangle$ decreases very gently, s increases as α increases. (2) When α is large, as $|c|$ decreases very gently but $\langle x, c\rangle$ decreases significantly, s decreases as α increases. Thus, the curve of s vs. α might have an *inverse U* shape, and there might exist an optimal α ($0 < \alpha < 1$) maximizing s. We can use grid search to find such a value to approach the ideal center c^*.

Validation. To validate the effectiveness of the FreqItem representation, we conduct experiments on four real-world sparse data sets, where the details of the data sets can be found in Subsection 5.1. We randomly sample 1,000 binary vectors from each data set as a cluster and compute the Mean Square Error (MSE) for performance evaluation. Intuitively, the lower the MSE, the more effective the center representation. Figure 2 shows the MSEs of FreqItem, Medoid, Mode1, and Mode2 for the four sparse data sets. For the FreqItem representation, we report the lowest MSE by a grid search of $\alpha \in [0, 1]$ with step 0.01. According to Figure 2, we find that the MSEs of FreqItem are the lowest among the four center representations, which empirically shows that compared with the other three baselines, the FreqItem is more effective and suitable to be the center representation for the set data on Jaccard distance.

To further investigate the impact of α, we plot the curve of MSE vs. α in Figure 3. To be concise, we only show results on Criteo10M and Avazu. However, similar trends can be observed from URL and KDD2012. From Figure 3, we have three interesting findings: (1) These curves are in a U shape. Thus, there exists an optimal α minimizing MSE, which justifies our analysis of the choice of α. (2) The cases $\alpha = 0$ and $\alpha = 1$ are always worse than others, which indicates that the FreqItem representation is uniformly better than Mode1 and Mode2. (3) There exists a wide range of α values corresponding to the (near) smallest MSE, i.e., $\alpha \in [0.17, 0.25]$ for Criteo10M and $\alpha \in [0.14, 0.26]$ for Avazu. This reason might be that the filtering threshold $\max(1, \lceil \alpha \cdot F \rceil)$ contains a ceiling function $\lceil \cdot \rceil$. Therefore, we can efficiently determine a near-optimal α with grid search (even with a coarse step size).

Algorithm 4: k-FreqItems

Input: data set $\mathcal{X} \subset \{0,1\}^d$, # clusters k, parameter α;
1 $C \leftarrow$ Select k binary vectors uniformly at random from $\mathcal{X}$;
2 **do**
3 Assign $x \in \mathcal{X}$ to X_c such that $c = \arg\min_{c' \in C} J(x, c')$;
4 Update $c \in C$ from (X_c, α);
5 **while** C *is not stable*;
6 **return** Clusters $\{X_c\}_{c \in C}$;

3.3 k-FreqItems

k-FreqItems is a k-Means variant designed for the set data on Jaccard distance. Suppose C is a set of cluster centers. The pseudo-code of k-FreqItems is depicted in Algorithm 4.

In the beginning, we choose k initial centers C uniformly at random from the data set $\mathcal{X}$ (Line 1). In the assignment step (Line 3), a tie can be broken arbitrarily as long as the method is consistent. In the update step (Line 4), with a pre-specified parameter α ($0 \leq \alpha \leq 1$), we re-compute each FreqItem $c \in C$ from its corresponding cluster X_c according to Definition 2. We call this assignment-update step as a *k-FreqItems iteration*. We repeat this iteration until the set of FreqItems C is stable (Line 5). Finally, we return all clusters $\{X_c\}_{c \in C}$ as the clustering results of $\mathcal{X}$ (Line 6).

The formal proof of convergence for k-FreqItems is not yet available. However, according to our experiments, we observe that the MSE decreases in each k-FreqItems iteration and approach a constant after several iterations. Those results empirically demonstrate that k-FreqItems is able to converge to a local optimum. More analyses will be discussed in Subsection 5.7.

Similar to the *Lloyd's* iteration, the k-FreqItems iteration can be done in $O(n\bar{d}k)$ time as the FreqItem can be computed in $O(n\bar{d})$ time. Thus, like k-Means, k-FreqItems is generally simple and efficient. Unfortunately, it only finds a local optimum, which might be poor in practice. Next, we will consider the seeding problem of k-FreqItems and introduce SILK to deal with this problem.

4 SILK

Once the center representation (i.e., FreqItem) is specified, state-of-the-art seeding methods, such as k-Means++, can be generalized to obtain an initial set of k centers that is $O(\log k)$-approximate to the optimal solution. Their time complexities, however, are highly sensitive to k, which might be unsuitable for massive data sets, especially when k is large. In this section, we design a new seeding step by leveraging LSH functions and propose SILK to remedy the scalability issue.

4.1 LSH Background

We first present the LSH background. LSH [4, 5, 12, 20, 25, 33, 39, 41–46, 50, 58, 59, 65, 67–69, 73, 87–91, 103–105] is a classic hashing scheme with an appealing property that the collision probability of the hash function for any two vectors x, y is inversely related to their distance. Given a distance function $Dist(\cdot, \cdot)$, a family of hash functions $\mathcal{H}$ is *locality-sensitive* to $Dist(\cdot, \cdot)$ if, for any x_1, x_2, y_1, y_2, the hash function $h \in \mathcal{H}$ satisfies the condition that $\Pr[h(x_1) = h(y_1)] \geq \Pr[h(x_2) = h(y_2)] \Leftrightarrow Dist(x_1, y_1) \leq Dist(x_2, y_2)$.

MinHash. It is a well-known LSH scheme [17, 18] for similarity search over Jaccard distance. Given any $x \in \{0,1\}^d$ and a random permutation $\pi : [d] \rightarrow [d]$, the MinHash function $h_\pi(\cdot)$ is defined as follows:

$$h_\pi(x) = \min_{i: x_i = 1} \pi(i). \tag{6}$$

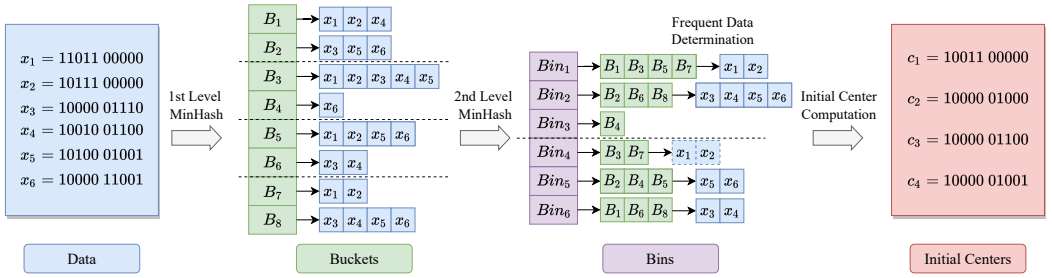


Fig. 4. An example of SILK overseeding.

Let $\delta = J(x, y)$ be the Jaccard distance for any two $x, y \in \{0,1\}^d$. The collision probability of x and y is computed as follows:

$$p(\delta) = \Pr[h_\pi(x) = h_\pi(y)] = 1 - \delta. \tag{7}$$

MinHash usually uses the (K, L)-bucketing framework [50] to find similar data. For any $x \in \{0,1\}^d$, it concatenates K i.i.d. MinHash functions to form a bucket, i.e., $H(x) = (h_{\pi_1}(x), \cdots, h_{\pi_K}(x))$. Due to the K-concatenation, the collision probability of any x and y decreases from p to p^K. Thus, the far data are hard to collide, but the collision probability of the close data is also reduced. As a remedy, MinHash builds L such hash tables to increase the accuracy.

4.2 SILK: A New Seeding Method

Overview. SILK is motivated by k-Means||, where both methods adopt the oversampling strategy to avoid finding a single seed in each iteration. Different from the k-Means|| seeding, we do not apply the D^2-sampling strategy in the overseeding step because this strategy inherently includes the comparison with the formerly selected seeds when we update Equation 3, leading to sequential iterations. With the insight that the close data in the same clusters often collide in the same buckets, we apply MinHash functions for oversampling and obtain the initial centers in one round.

SILK Overseeding. The SILK overseeding consists of three steps: two-level MinHash, frequent data determination, and initial center computation. Its pseudo-code is depicted in Algorithm 5.

Two-Level MinHash. We first apply MinHash to convert binary vectors $\mathcal{X}$ into buckets $\mathcal{B}$ (Line 1). Note that we do not directly obtain the initial centers from these buckets. The reasons are two folds: (1) There may be too many buckets with once MinHash, and many of them are very similar. (2) Though the data in the same bucket could be close to each other, they do not necessarily belong to the same cluster. Thus, the FreqItem constructed from a single bucket might not be an excellent initial center.

We apply MinHash again to convert the buckets $\mathcal{B}$ into bins $\mathcal{T}$ (Line 2).[3] With the second level MinHash, we can group similar buckets into the same bins and retrieve the *frequent data* from similar buckets. Since these frequent data are commonly found in multiple similar buckets, they are relatively rare and have a higher probability in the same cluster. Thus, using such data to determine initial centers is probably more efficient and promising. Next, we illustrate how to determine the frequent data from bins.

Frequent Data Determination. For ease of illustration, we first provide a concrete definition of the frequent data, which is inspired by the FreqItem representation.

[3]To distinguish the buckets from the first level MinHash, we call the buckets produced from the second level MinHash as bins. Each bin is a set of bucket identities (IDs).

Algorithm 5: SILK Overseeding

Input: data set $\mathcal{X} \subset \{0,1\}^d$, filtering parameters α, β, parameters K_1, L_1, K_2, L_2 for the two-level MinHash;

1 $\mathcal{B} \leftarrow$ Apply MinHash(K_1, L_1) for $\mathcal{X}$;
2 $\mathcal{T} \leftarrow$ Apply MinHash(K_2, L_2) for $\mathcal{B}$;
3 $\mathcal{S} \leftarrow \emptyset$;
4 **foreach** $Bin \in \mathcal{T}$ **do**
5 **if** $|Bin| \leq 1$ **then continue**;
6 Determine a set of frequent data X from (Bin, β);
7 $\mathcal{S} \leftarrow \mathcal{S} \cup \{X\}$;
8 Remove the near duplications of $\mathcal{S}$;
9 $C \leftarrow \emptyset$;
10 **foreach** $X \in \mathcal{S}$ **do**
11 Compute the FreqItem c from (X, α);
12 $C \leftarrow C \cup \{c\}$;
13 **return** C;

Definition 3 (Frequent Data)**:** *Given a set of buckets from the same bin and a parameter* β $(0 \leq \beta \leq 1)$*, the frequent data is a set of data that are appeared in those buckets with at least* $\max(1, \lceil \beta \cdot F' \rceil)$ *time, where* F' *is the maximum frequency among the data.*

The procedure of frequent data determination (Lines 3–7) is described as follows. Let $\mathcal{S}$ be a set to store the frequent data. For each $Bin \in \mathcal{T}$, if it has a single bucket only, i.e., $|Bin| \leq 1$, we ignore it because it cannot ensure most data from this bucket are in the same cluster. Otherwise, we retrieve and scan the data from the buckets in Bin and count their frequency, which is the number of occurrences of each data in those buckets. As such, we can get the maximum frequency (i.e., F') among the data. Based on Definition 3, we determine a set of frequent data X from Bin and add X into $\mathcal{S}$. We repeat this procedure to determine all X's from $\mathcal{T}$. Note that the frequent data determination is similar to the computation of FreqItem, and they have the same linear time complexity.

Due to the randomness of MinHash functions, some near duplicate X's might be added into $\mathcal{S}$ multiple times. Fortunately, since X consists of a set of data IDs, we can consider $\mathcal{S}$ as a set of buckets as input; then, we repeat the second level MinHash and the frequent data determination for deduplication (Line 8).

In practice, we suggest setting a large value of β to reduce the frequent data size $|X|$ and get the highly frequent data to produce initial centers. For example, we set $\beta = 0.9$ in the experiments. Moreover, since the motivation of deduplication is to remove the highly similar or identical X's, it is unnecessary to use large K and L values for the second level MinHash. Therefore, to improve efficiency, we set $K = 2$ and $L = 1$ for deduplication in the experiments.

Initial Center Computation. Let C be a set of initial centers. It starts as an empty set (Line 9). For each set of frequent data $X \in \mathcal{S}$, we follow Definition 2 to compute the FreqItem c and add c into C. We repeat this procedure to get all initial centers (Lines 10–12).

Example 2: We now use Figure 4 to illustrate the SILK overseeding. Suppose $\mathcal{X} = \{x_1, x_2, \cdots, x_6\}$ and there exist 3 clusters, i.e., $\{x_1, x_2\}$, $\{x_3, x_4\}$, and $\{x_5, x_6\}$. In this example, we set $L_1 = 4$, $L_2 = 2$, and $\alpha = \beta = 0.6$. We first apply MinHash to partition $\mathcal{X}$ into buckets $\mathcal{B}$. With $L_1 = 4$, we get 8

Algorithm 6: SILK

Input: data set $\mathcal{X} \subset \{0,1\}^d$, # clusters k, parameters α, β, parameters K_1, L_1, K_2, L_2 for the two-level MinHash;

1 $C \leftarrow$ SILK Overseeding($\mathcal{X}, \alpha, \beta, K_1, L_1, K_2, L_2$);
2 **foreach** $c \in C$ **do**
3 Assign $x \in \mathcal{X}$ to X_c such that $c = \arg\min_{c' \in C} J(x, c')$;
4 $w_c \leftarrow |X_c|$;
5 $C' \leftarrow$ k-Means++ Seeding(C, w, k);
6 Proceed with the standard k-FreqItems iterations;
7 **return** Clusters $\{X_c\}_{c \in C'}$;

buckets, e.g., we get 2 buckets $B_1 = \{x_1, x_2, x_4\}$ and $B_2 = \{x_3, x_5, x_6\}$ from the first hash table. Next, with $L_2 = 2$, we use $\mathcal{B}$ as input and apply MinHash again to get 6 bins.

We then determine the frequent data. For $Bin_1 \in \mathcal{T}$, we first scan the data from $\{B_1, B_3, B_5, B_7\}$ and count their frequency, i.e., $(x_1, 4)$, $(x_2, 4)$, $(x_3, 1)$, $(x_4, 2)$, $(x_5, 2)$, $(x_6, 1)$. Thus, we have $F' = 4$ and $\max(1, \lceil \beta \cdot F' \rceil) = 3$. Based on Definition 3, we get the frequent data $X_1 = \{x_1, x_2\}$ from Bin_1. Similarly, we get $X_2 = \{x_3, x_4, x_5, x_6\}$ from Bin_2. We ignore Bin_3 as $|Bin_3| \leq 1$. Some identical X_i's have been added into $\mathcal{S}$ in multiple times, e.g., we add the same $\{x_1, x_2\}$ twice from both Bin_1 and Bin_4. Thus, we apply MinHash again for deduplication and keep 4 sets of frequent data, i.e., $X_1 = \{x_1, x_2\}$, $X_2 = \{x_3, x_4, x_5, x_6\}$, $X_3 = \{x_5, x_6\}$, and $X_4 = \{x_3, x_4\}$.

Finally, we compute the initial centers from $\mathcal{S}$. For $X_1 \in \mathcal{S}$, we count the frequency of the non-zero dimensions from $\{x_1, x_2\}$, i.e., $(d_1, 2)$, $(d_2, 1)$, $(d_3, 1)$, $(d_4, 2)$, $(d_5, 2)$. Thus, $F = 2$ and $\max(1, \lceil \alpha \cdot F \rceil) = 2$. Based on Definition 2, we select $\{d_1, d_4, d_5\}$ as the frequent items and get $c_1 =$ 10011 00000. Similarly, we get $c_2 =$ 10000 01000, $c_3 =$ 10000 01100, and $c_4 =$ 10000 01001 from X_2, X_3, and X_4. △

SILK Seeding. The SILK overseeding is the first step of SILK seeding. After that, we get a set of initial centers C, where its size $|C|$ is usually larger than k. Thus, we follow the procedure of k-Means|| seeding to reduce $|C|$ to k, as it is efficient yet very effective.

We perform once data assignment to get the weights of initial centers. Specifically, we first assign each $x \in \mathcal{X}$ to its closest cluster X_c, where $c = \arg\min_{c' \in C} J(x, c')$; then, we get the weight w_c of each initial center $c \in C$ based on the number of data in X_c, i.e., $w_c = |X_c|$. After that, we call k-Means++ seeding (Algorithm 1 by replacing Euclidean distance to Jaccard distance) to re-cluster the weighted initial centers to determine the final k cluster centers. Since $|C| \ll |\mathcal{X}|$, the cost of re-clustering is not significant.

There are several ways to control $|C|$. For example, we can get more initial centers from more hash tables (i.e., set larger L_1 and L_2). Moreover, to reduce $|C|$, we can ignore more bins when we determine the frequent data (i.e., Line 5 in Algorithm 5).

The Full Algorithm. SILK is a randomized initialization method. After the SILK seeding, it is followed by the standard k-FreqItems iterations, until the set of cluster centers is stable. The pseudo-code of SILK is depicted in Algorithm 6.

Time Complexity Analysis. As SILK proceeds with the standard k-FreqItems iterations, we focus on the SILK seeding. According to MinHash [18], $K_1 = K_2 = O(\log n)$ and $L_1 = L_2 = O(n^\rho)$, where $\rho = \ln p_1 / \ln p_2$; p_1 and p_2 can be computed by Equation 7 with two distance thresholds r_1 and r_2 ($0 < r_1 < r_2 < 1$). For ease of analysis, we assume $|C|$ and k are of the same magnitude, i.e., $|C| = O(k)$.

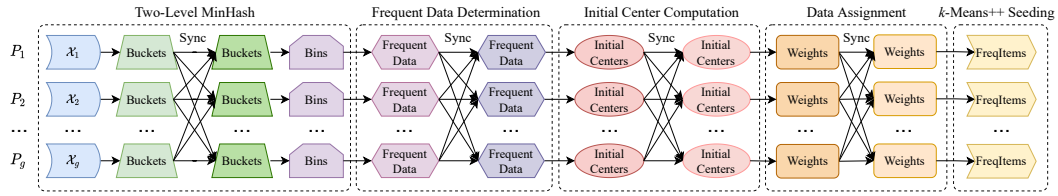


Fig. 5. The system architecture of SILK seeding.

We first analyze the time complexity of SILK overseeding, which consists of four parts. (1) The first level MinHash takes $O(n\bar{d}K_1L_1) = O(\bar{d}n^{1+\rho}\log n)$ time. (2) Let $n_{\mathcal{B}}$ be the number of buckets and $\bar{d}_{\mathcal{B}}$ be the average bucket size. As $n_{\mathcal{B}}\bar{d}_{\mathcal{B}} = nL_1 = O(n^{1+\rho})$, the second level MinHash spends $O(n_{\mathcal{B}}\bar{d}_{\mathcal{B}}K_2L_2) = O(n^{1+2\rho}\log n)$ time. (3) The frequent data determination requires $O(L_2 n_{\mathcal{B}}\bar{d}_{\mathcal{B}}) = O(n^{1+2\rho})$ time. (4) The initial center computation takes $O(\bar{d}n_{\mathcal{S}})$ time, where $n_{\mathcal{S}} = \sum_{X\in\mathcal{S}}|X|$. Thus, the time complexity of SILK overseeding is $O(\bar{d}n^{1+\rho}\log n + n^{1+2\rho}\log n + n^{1+2\rho} + \bar{d}n_{\mathcal{S}}) = O(n^{1+2\rho}\log n)$.

After that, the data assignment to get weights takes $O(n\bar{d}|C|) = O(n\bar{d}k)$ time. Finally, the k-Means++ seeding spends $O(|C|\bar{d}k) = O(\bar{d}k^2)$ time to get the final k centers. Thus, the time complexity of SILK seeding is $O(n^{1+2\rho}\log n + n\bar{d}k + \bar{d}k^2) = O(n^{1+2\rho}\log n)$.

Remarks. With the two-level MinHash for oversampling, the SILK overseeding can produce $|C|$ seeds at a time and hence avoid the sequential iterations required by k-Means++ and k-Means||. However, as k-Means|| overseeding only takes $O(n\bar{d}|C|)$ time, where $|C| = r \cdot l + 1$, it has a lower time complexity than the SILK overseeding. Note that the two-level MinHash is the most time-consuming part of the SILK overseeding. Fortunately, due to the *independence* nature of MinHash functions, the SILK overseeding can be implemented in a fully parallel fashion. In contrast, the parallelism exploited in k-Means|| overseeding only applies in each iteration. Thus, with sufficient resources, the SILK overseeding can be more efficient than k-Means|| overseeding in practice.

4.3 A Distributed Implementation

We now present the implementation of SILK on a distributed platform with multiple GPUs, with a specific focus on SILK seeding. The k-FreqItems iteration can be considered as comprising of the initial center computation (update step) and the subsequent data assignment to calculate weights (assignment step), which are the two key intermediate steps of SILK seeding. To facilitate communication, we utilize the Message Passing Interface (MPI). In a system with g GPUs, we evenly distribute the data sets and launch g processes, where each process corresponds to a single GPU for parallel computation. The system architecture is visually depicted in Figure 5.

Two-Level MinHash. We first instruct GPU threads to compute the MinHash functions to convert the data into buckets. Since each process P_j only has a part of the data, the buckets are incomplete. Thus, we perform a bucket synchronization to gather the buckets such that (1) the data IDs with the same MinHash values are put into the same buckets; (2) each P_j contains a *subset* of hash tables with (almost) equal-sized to keep the workload balanced. Afterward, we instruct the GPU threads to calculate the second-level MinHash functions to transform buckets into bins.

Frequent Data Determination. Note that the bins are incomplete as we do not broadcast the local buckets to all processes, but determining frequent data requires the complete bucket information in each bin to determine the maximum frequency. Nonetheless, to reduce the communication cost, we do not synchronize bins. Instead, we first determine the *local* frequent data for each P_j from its local buckets with a pre-specified local β; then, we broadcast the local frequent data and their frequency to all processes; finally, we obtain the *global* frequent data with the global β. By employing local β,

the global frequent data is rendered as an approximation of the ground truth. Encouragingly, any potential error stemming from this approximation can be effectively minimized through meticulous adjustment of the local β, which shall be validated in Subsection 5.8.

Initial Center Computation. So far, we have obtained the global frequent data IDs in each process. However, due to the dispersed data storage across processes, precise computation of the initial centers would incur substantial communication overhead. Hence, like determining frequent data, we design a strategy to approximately compute the initial centers. This strategy involves determining the *local* frequent items using a pre-defined local α, broadcasting them along with their frequencies to all processes, and subsequently computing the *global* FreqItems using the global α. By appropriately selecting the local α, the global FreqItems can closely approximate the exact ones. The validation of this assertion will be provided in Subsection 5.8.

Data Assignment and k-Means++ Seeding. After the SILK overseeding, each P_j has the global initial centers. We initially compute the local weights for the local data through data assignment. Subsequently, the local weights are broadcasted to all processes to obtain the global weights. Finally, leveraging the global initial centers and global weights as inputs, we utilize OpenMP to perform k-Means++ seeding, resulting in the generation of the final k cluster centers.

4.4 Discussions

Generalization. Although we only present SILK in the Hamming spaces for the objective function $\phi_{\mathcal{X}}(C) = \sum_{x \in \mathcal{X}} \min_{c \in C} J(x, c)^2$, this seeding technique can be extended to arbitrary metric spaces with more general objective functions. These generalizations only need to change (1) the center representation and (2) the LSH scheme for the input metric. For example, for the $\phi_{\mathcal{X}}(C)$ defined by Equation 2, we can extend the SILK seeding to support the most commonly used Euclidean distance by (1) using Centroid as the center and (2) utilizing the vanilla E2LSH [3, 25] to produce buckets.

Application Scenarios. As k-FreqItems and SILK aim to optimize $\phi_{\mathcal{X}}(C)$, they are good at finding compact clusters. Therefore, when dealing with sparse data sets that exhibit a clustering pattern in a spherical shape, they have the potential to yield high-quality results. However, it should be noted that, akin to many other partitioning-based techniques, they may encounter limitations when encountering data sets with complex shapes. On the other hand, as will be validated in Section 5, k-FreqItems and SILK can efficiently handle large k values. They are ideal as a pre-processing tool to generate many small, compact microclusters, which can then be used as summarized information for other data analytical operations, including classification [99] and clustering [100]. Thus, the output of k-FreqItems and SILK is not an end but can act as input to scale up many other operations.

5 EXPERIMENTS

5.1 Data Sets

We use five real-world data sets (i.e., URL,[4] Avazu,[5] KDD2012,[6] Criteo10M and Criteo1B[7]) with large cardinality and ultra-high dimensionality to evaluate the efficiency and effectiveness of k-FreqItems and SILK. We also select two small, sparse data sets (i.e., News20[8] and RCV1[9]) with ground truth cluster labels to asses the clustering quality. Their statistics are summarized in Table 1.

[4]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#url.
[5]https://www.kaggle.com/c/avazu-ctr-prediction/data.
[6]https://www.kaggle.com/c/kddcup2012-track2/data.
[7]http://labs.criteo.com/2013/12/download-terabyte-click-logs/.
[8]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#news20.
[9]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#rcv1.multiclass.

Table 1. The statistics of seven sparse data sets.

Data Sets	n	d	$\bar{d}$	Global α	Local α
News20	2.0×10^4	6.2×10^4	80	0.2	–
RCV1	5.3×10^5	4.7×10^4	65	0.2	–
URL	2.3×10^6	3.2×10^6	116	0.5	0.3
Criteo10M	1.0×10^7	1.0×10^6	39	0.3	0.3
Avazu	4.0×10^7	1.0×10^6	15	0.4	0.4
KDD2012	1.5×10^8	5.4×10^7	11	0.3	0.4
Criteo1B	1.0×10^9	1.0×10^6	39	0.3	0.3

- **News20** is a collection of nearly 20,000 newsgroup documents, partitioned almost evenly across 20 newsgroups [57].
- **RCV1** is an archive of 804,414 manually categorized newswire stories from Reuters, Ltd [60]. We follow [14] and obtain a multiclass subset with 53 categories.
- **URL** is a collection of 2.3 million URLs that are used for detecting malicious URLs [74]. It has about 3.2 million features but with only around 116 non-zeros.
- **Avazu** comes from a competition of click-through rate prediction jointly hosted by Avazu and Kaggle in 2014. We generate it by applying the winning solution.[10]
- **KDD2012** is extracted from the training file of the second track in KDD CUP 2012. We convert it into binary vectors according to the number of possible categories.
- **Criteo10M and Criteo1B** are the first 10 million and 1 billion records extracted from Criteo, which consists of 24 days' click logs with around 4 billion records.

5.2 Evaluation Metrics

We use the following 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.
- **MSE.** We use the Mean Square Error (MSE) to measure the effectiveness of a method. Given a set of n binary vectors $\mathcal{X} \in \{0,1\}^d$ and k centers C, $MSE = \frac{1}{n}\sum_{x\in\mathcal{X}} \min_{c\in C} J(x,c)^2$. A small MSE indicates an effective clustering result.
- **ARI.** We use Adjust Rand Index (ARI) [49, 93] to evaluate the quality of clusters generated by a method compared with ground truth labels. A large ARI indicates a high-quality result.
- **Purity.** We also adopt purity [28] to assess the clustering quality, i.e., purity $= \frac{1}{n}\sum_{i=1}^{k} \max_{1\le j\le k} n_i^j$, where n_i^j is the number of data in cluster i that belongs to the ground truth cluster j. A large purity value means a high-quality result.

5.3 Competitors

As demonstrated in Subsection 3.2, the FreqItem representation is much better than Medoid and Mode. Thus, we use FreqItem as the cluster center for Jaccard distance. To validate the clustering performance of k-FreqItems and SILK, we adapt two state-of-the-art methods (i.e., k-Means++ and k-Means||) for Jaccard distance as baselines. To make a systematic evaluation of k-FreqItems and SILK, we also implement four hybrid methods (i.e., embedding together with conventional clustering) for comparison. Their MSEs are computed on the original sparse data based on the output labels. The details of these methods are described below:

[10]https://github.com/ycjuan/kaggle-avazu.

- k-**FreqItems.** It is implemented as depicted in Section 3. We select k data uniformly at random as k initial centers (random seeding).
- **SILK.** We implement SILK as described in Section 4. We use the SILK seeding to obtain the k initial centers.
- k-**FreqItems++.** We adapt k-Means++ to tackle sparse data clustering based on Jaccard distance: (1) we use the FreqItem as the cluster center; (2) we conduct data assignment based on Jaccard distance. Referring to Algorithm 1 (the seeding step), we apply OpenMP to speed up the computation of $D(x, C)$ of the k sequential iterations. We call it k-FreqItems++ and name the seeding step k-FreqItems++ seeding.
- k-**FreqItems‖.** Similar to k-FreqItems++, we adapt k-Means‖ for sparse data clustering. Referring to Algorithm 2, as it adds $l = O(k)$ seeds into C in each sequential round, we leverage GPUs to compute $D(x, C)$ in parallel. We call it k-FreqItems‖ and name the seeding step k-FreqItems‖ seeding.
- **SVD+**k-**Means**, **SE+**k-**Means**, **SVD+DBSCAN**, **SE+DBSCAN.** We implement these four hybrid methods for performing sparse data clustering: (1) We use Singular Value Decomposition (SVD) [37] and Sparse Embedding (SE) [64] to embed the high-dimensional, sparse data into a series of low-dimensional spaces, i.e., $d' \in \{10, 20, 50, 100, 200\}$.[11] (2) For the embedded data, we run Mini-Batch k-Means [85] and DBSCAN [31] based on Euclidean distance to get the clustering results.

Experiments Environments. All experiments are conducted on a distributed platform with two computing nodes: (1) node 1 has an Intel® Xeon® Platinum 8170 CPU @ 2.10 GHz with 52 cores; (2) node 2 has an Intel® Xeon® Gold 6130 CPU @ 2.10 GHz with 32 cores. In addition, both nodes have 4 NVIDIA GeForce GTX 1080 Ti's with 11 GB memory and run on CentOS 7.4.

For the two small data sets News20 and RCV1, we conduct the experiments on node 1 with 1 GPU. For the five large data sets, we conduct the experiments on node 1 with 4 GPUs by default except for the scalability test to multiple GPUs (Subsection 5.8).

Parameter Settings. We consider k from 1,000 to 10,000. For k-FreqItems‖, we set $r = 5$ and $l = k$ [10, 11]. For SILK, we set $K_1, K_2 \in \{4, 6\}$, $L_1 \in \{8, 16, 24\}$, and $L_2 \in \{6, 8, 10\}$ to get different number of seeds. Also, we observe that the performance of SILK is comparable for a wide range of large global β and local β values. For simplicity, we fix global $\beta = 0.9$ and local $\beta = 0.8$. The settings of global α and local α will be discussed in the next subsection.

5.4 Impacts of Global α and Local α

We first study the impacts of global α and local α for k-FreqItems (with $k = 1,000$) to guide the users for parameter setting. We vary global $\alpha \in \{0.1, 0.2, \cdots, 0.9\}$ and local $\alpha \in \{0.1, 0.2, \cdots, 0.9\}$ and plot the 3D scatter points in Figure 6.

We have two interesting observations. First, the scatter points over the four data sets are all in a U shape, which indicates that there may exist a pair of global α and local α minimizing MSE and further justifies our analysis of the choice of α discussed in Subsection 3.2. Second, there exists a wide range of global α and local α such that their corresponding MSEs are very close to the smallest MSE, i.e., global $\alpha \in \{0.2, \cdots, 0.5\}$ and local $\alpha \in \{0.2, \cdots, 0.5\}$. This finding implies that we can efficiently find a near-optimal pair with grid search even with a coarse increment like 0.1. These observations also correspond to the findings from Figure 3.

In the following experiments, we set up global α and local α in two steps: (1) we keep the top-m pairs (e.g., $m = 5$) of global α and local α with the smallest MSEs as candidates such that all of

[11]We only report $d' = 200$ if not otherwise specified, as they reach the best results.

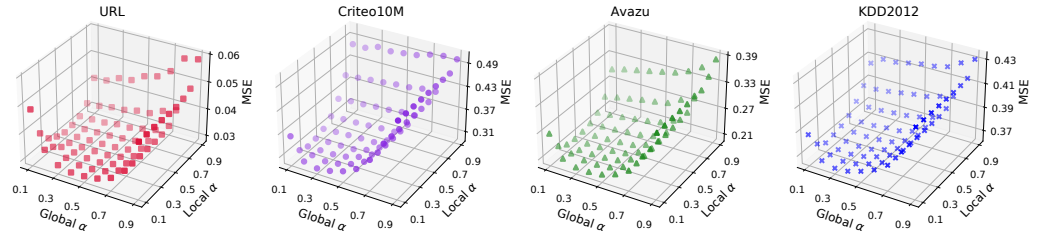


Fig. 6. Impacts of global α and local α for k-FreqItems.

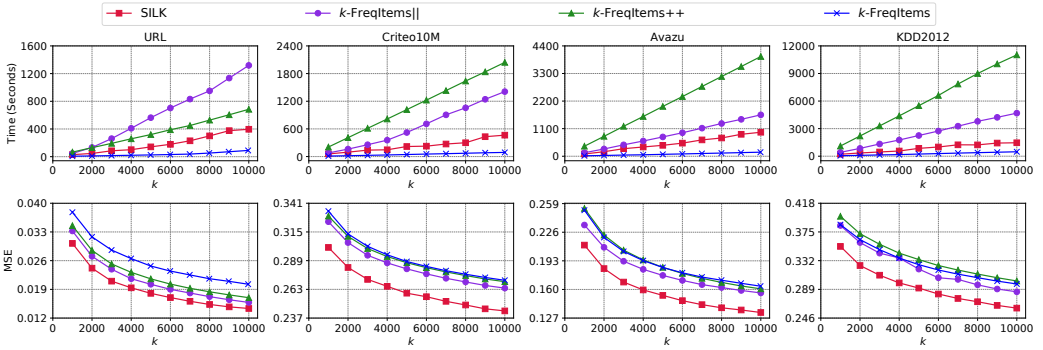


Fig. 7. Seeding results of different methods.

which allow FreqItem as an effective center; (2) as their differences in MSE are almost negligible, we select the pair with the largest global α and local α to determine FreqItem. Since this pair leads to the smallest $\bar{d}$, the FreqItem can be computed efficiently. Their settings are depicted in Table 1. Note that as k-FreqItems++, k-FreqItems||, and SILK also apply FreqItem as the cluster center, the settings of global α and local α are the same for them.

5.5 Seeding Performance

We now study the seeding performance of SILK. To make a fair comparison, we evaluate the seeding performance of the four k-FreqItems series methods by considering only the seeding step and the first iteration. Their seeding results are shown in Figure 7.

The SILK seeding is around 1.1∼3.2× faster than k-FreqItems++ seeding and k-FreqItems|| seeding, and its advantage is more significant as k increases. The primary reason is that the SILK overseeding is independent of k, and it produces seeds in one round. In contrast, the time complexity of k-FreqItems++ seeding and k-FreqItems|| seeding is linear to k, and they respectively require sequential k iterations and r rounds to generate seeds.

The running time of k-FreqItems (with random seeding) is the least among the competitors, but its MSEs are also the largest in most cases (except KDD2012). The MSEs of SILK are much smaller than those of the three competitors. For example, their differences to the second-best method (except URL) are even more significant than those between the second-best and the worst method. These results justify the effectiveness of our intuition in leveraging LSH functions to determine seeds from similar buckets.

In addition, the running time of k-FreqItems++ seeding on URL is less than that of k-FreqItems|| seeding. The reason might be that the data set URL has larger $\bar{d}$ than other data sets. Thus, with a

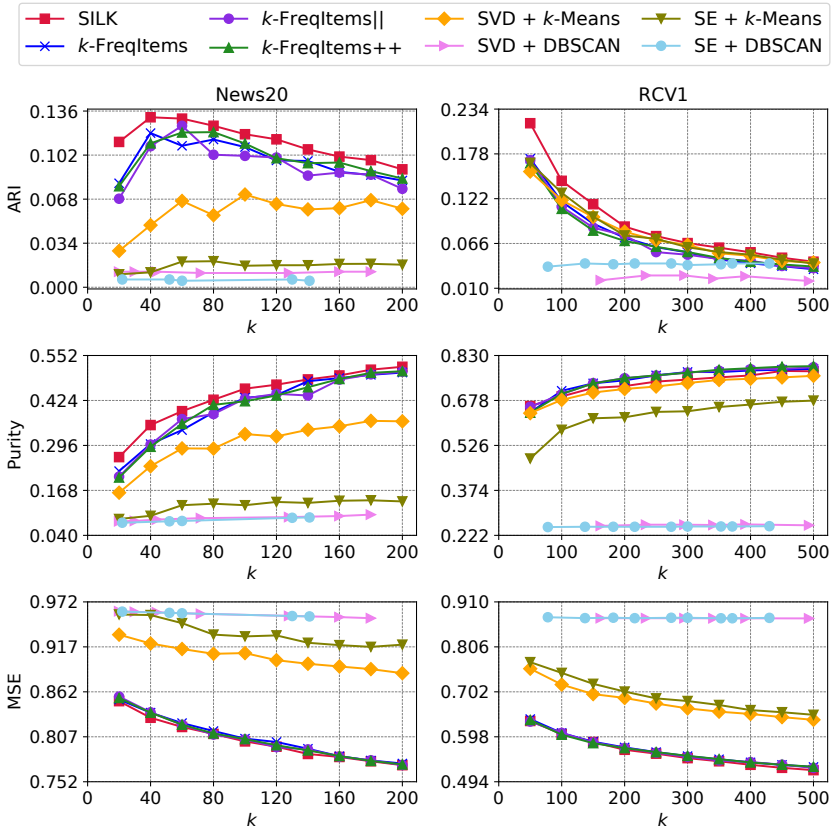


Fig. 8. Clustering results on News20 and RCV1.

fixed GPU memory budget, the number of data that k-FreqItems|| can process in a single batch becomes less, leading to more times of GPU memory loading, which is generally expensive. As $\bar{d}$ decreases, k-FreqItems|| seeding will be more efficient than k-FreqItems++ seeding, which is in accordance with the results in Figure 7.

5.6 Clustering Performance

Performance on Small Data Sets. We first evaluate the clustering quality of the four k-FreqItems series methods and four hybrid methods on two small, sparse data sets with ground truth labels. The results are depicted in Figure 8.

For News20, SILK achieves the highest ARI and Purity together with the lowest MSE, and the k-FreqItems series methods outperform the four hybrid methods in all metrics. For RCV1, the k-FreqItems series methods also perform better in terms of MSE and Purity, and SILK also has marginal improvement on ARI compared to embedding methods. Thus, the k-FreqItems series methods outperform the four hybrid methods, especially in Purity and MSE. Moreover, SILK has a uniformly larger ARI than the four hybrid methods. The reason might be that the hybrid methods require embedding the high-dimensional, sparse data into low-dimensional space. They prune off many dimensions and incur a large amount of *information loss*. Thus, their clustering quality is worse than that of the k-FreqItems series methods, which indicates that the hybrid methods might not be suitable for the high-dimensional, sparse data sets.

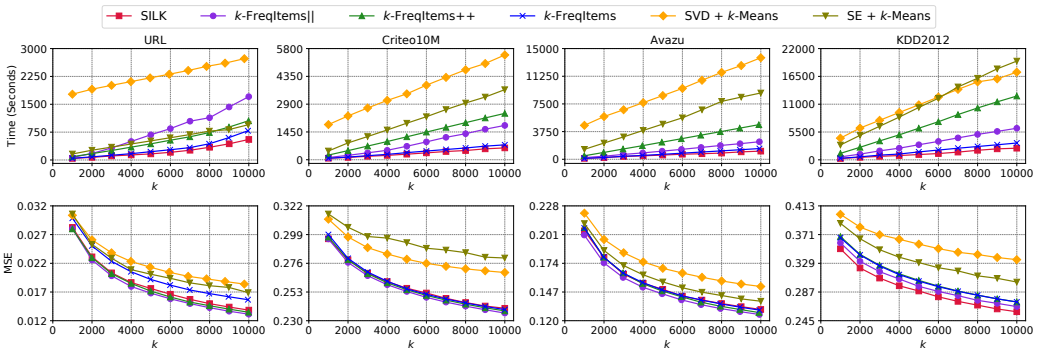


Fig. 9. Clustering results of different methods.

For DBSCAN, we show its results with $d' = 10$ as it achieves the best clustering quality. Moreover, as it cannot directly specify k, we perform a grid search of its parameters (i.e., $\varepsilon \in [0.05, 20]$ with step 0.05 and MinPts $\in [3, 10]$ with step 1) to obtain its best results for different k. However, the clustering quality of SVD+DBSCAN and SE+DBSCAN is still the worst among the competitors. These results indicate that DBSCAN might not be suitable for sparse data even after embedding as the similarity only preserves better with more dimensions, but DBSCAN degrades rapidly as d' increases. Due to the poor efficiency of DBSCAN (e.g., 10∼100× slower than Mini-Batch k-Means in our experiments) under the exhaustive parameter searching, we only test it on two small data sets.

Performance on Large-scale Data Sets. We continue to run the four k-FreqItems series methods as soon as they converge to their local optimums. The clustering results are shown in Figure 9, where the running time of SVD+k-Means and SE+k-Means consists of the embedding time and the clustering time.

The four k-FreqItems series methods are much more efficient than SVD+k-Means and SE+k-Means for most data sets except for the URL, whose cardinality and dimensionality are relatively smaller. Moreover, the running time of SILK is the least among all six competitors as the good seeds of SILK lead to fast convergence. As for the effectiveness, the k-FreqItems series methods also have an obvious advantage against SVD+k-Means and SE+k-Means, especially on Criteo10M and KDD2012. On the other hand, the worse performance of SVD+k-Means and SE+k-Means on both measures further validate that the hybrid approaches might not be suitable for the sparse data sets with ultra-high dimensionality.

Among the four k-FreqItems series methods, k-FreqItems (with random seeding) generally has the worst performance, while SILK with good seeds achieves the least or at least comparable MSE. These results justify that the seeding strategies indeed affect the clustering quality. Besides, compared to SVD+k-Means and SE+k-Means, the k-FreqItems series methods have very close MSEs. This might be because they all use the FreqItem as the cluster center (with the same global α and local α) and follow the identical k-FreqItems iteration. In summary, the k-FreqItems series methods (especially SILK) perform better against the hybrid methods regarding finding high-quality clusters on sparse data sets.

5.7 Validation of Convergence

Next, we validate the convergence of the k-FreqItems series methods. We show their curves of MSE vs. # iterations in Figure 10. To be concise, we only display the convergence results with $k = 10,000$. Similar patterns can be observed from other k values.

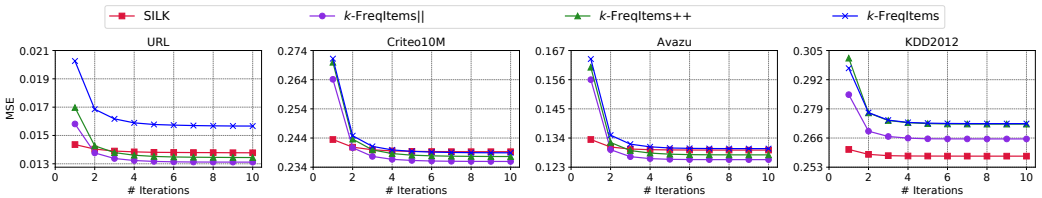


Fig. 10. Convergence results of different methods ($k = 10,000$).

We have two interesting findings from Figure 10. First, the MSEs of all methods decrease monotonically as # iterations increases, and their values approach a constant after a few iterations. These results empirically validate that k-FreqItems can converge to a local optimum. Moreover, the same claim can be confirmed for k-FreqItems++, k-FreqItems||, and SILK. Second, the convergence speed might be relevant to the seeding performance. For example, as the SILK seeding obtains more effective seeds (e.g., smaller MSEs) than k-FreqItems++ seeding, k-FreqItems|| seeding, and random seeding, the convergence time of SILK is earlier than that of the three baselines by about 1~3 iterations.

5.8 Scalability to Multiple GPUs

To verify the effectiveness of the distributed implementation, we study the performance of SILK under different # GPUs. We vary the # GPUs in a range of $\{1, 2, 4, 8\}$. To be concise, we only show the seeding results of SILK on Criteo10M and Avazu. Similar trends can be observed from other data sets.

We have two observations from Figure 11. (1) The running time of SILK is asymptotically linear to # GPUs, which indicates that the workload of each GPU is approximately balanced, and the communication cost is vastly reduced. (2) The curves of MSE vs. k for different # GPUs are very close to each other. This finding shows that with proper settings of local α and local β, the approximation of the FreqItem and the set of frequent data can approach their ground truths. The two observations also validate the effectiveness of the distributed implementation of SILK and show its potential scalability to tackle massive data sets with multiple GPUs. Note that the running time of SILK does not increase smoothly, especially on 1 and 2 GPUs. The reason might be that we select different MinHash parameters (e.g., L_1 and L_2) for different k to smooth MSE. The larger L_1 (or L_2) leads to longer running time.

We also study the performance of k-FreqItems|| under different # GPUs. Due to space limitation, we only report its seeding results on Criteo10M and Avazu in Figure 12. Similar to SILK, we observe that (1) the running time of k-FreqItems|| is asymptotically linear to # GPUs; (2) the curves of MSE vs. k for different # GPUs are very close to each other. Observation (2) further shows that with a proper setting of local α, the approximation of the FreqItem can approach its ground truth. These results also validate the effectiveness of our adaptation of k-FreqItems|| (i.e., adapt k-Means|| for Jaccard distance) and our distributed implementation of k-FreqItems||.

5.9 Performance on Billion-Scale Data Set

We study the performance of the k-FreqItems series methods on the billion-scale data set Criteo1B. We set the same global α and local α as Criteo10M. The results are depicted in Figure 13.

Compared with k-FreqItems, k-FreqItems++, and k-FreqItems||, SILK shows more elevated efficiency superiority on this billion-scale data set, especially for the seeding results. For example, SILK uses only 12.6 hours to obtain $k = 10,000$ seeds, while k-FreqItems|| and k-FreqItems++ need about 40.7 and 53.1 hours, respectively. In other words, SILK is more than 3.2× and 4.2× faster

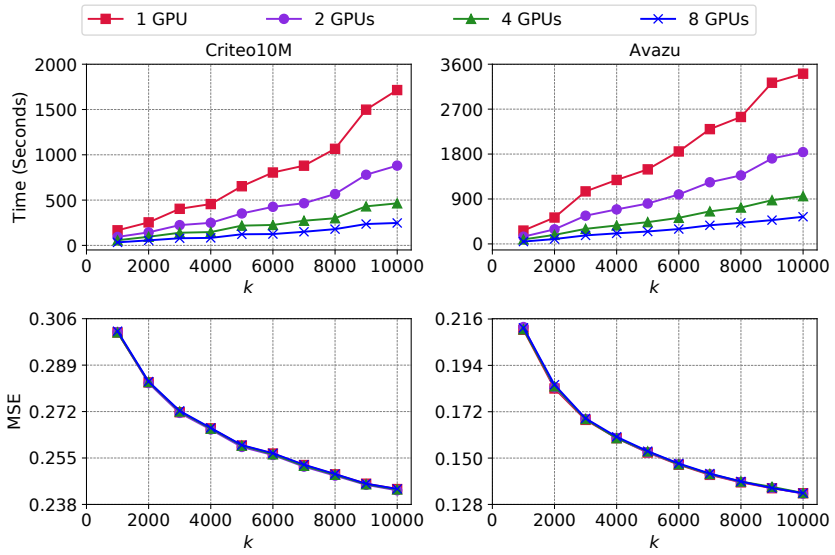


Fig. 11. SILK seeding on different # GPUs.

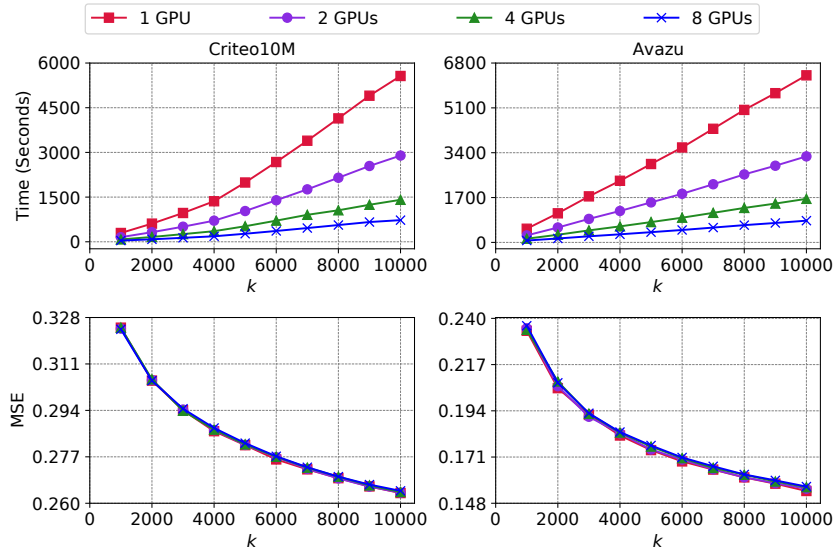


Fig. 12. k-FreqItems|| seeding on different # GPUs.

than k-FreqItems|| and k-FreqItems++, respectively. The speedup ratio is the largest among the five large data sets.

Besides, we discover that the MSEs of SILK after seeding are very close to their convergence values. For example, for $k = 10,000$, the MSE of SILK after seeding is 0.291, while its corresponding value after convergence is 0.287. The difference is 0.004 only. If a small estimated error is allowed, we can directly use the SILK seeding results as outputs without extra iteration. As such, SILK only takes 12.6 hours. As for a comparison, the MSE of k-FreqItems|| after convergence is 0.281, but it takes 47.9 hours.

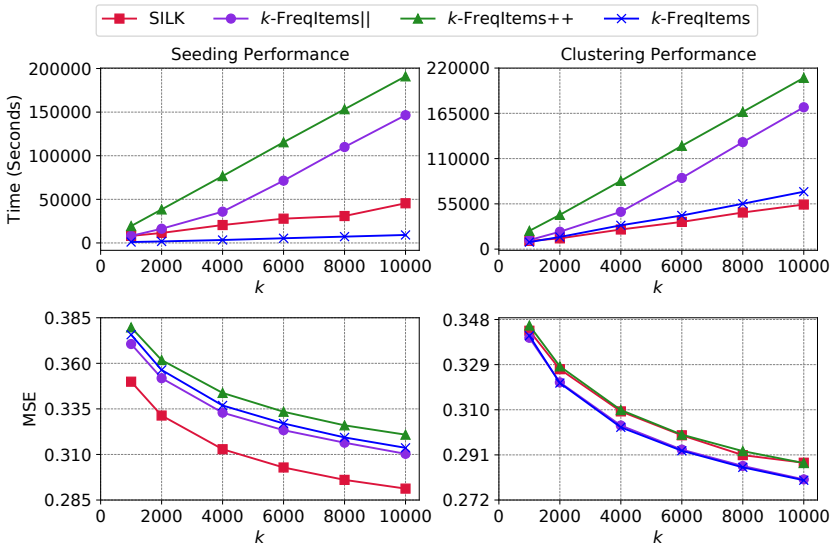


Fig. 13. Seeding and clustering results on Criteo1B.

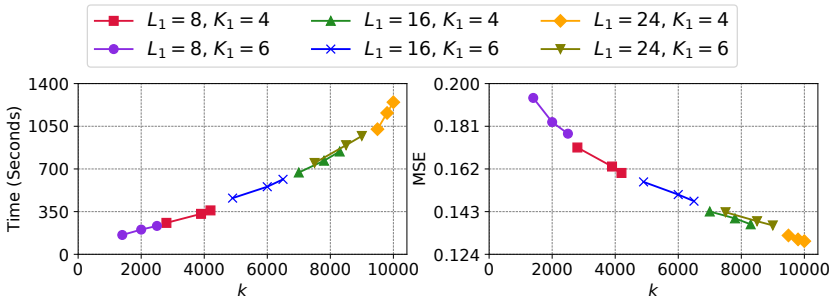


Fig. 14. Impacts of MinHash parameters.

5.10 Impacts of MinHash Parameters

Finally, we study the impacts of MinHash Parameters in SILK, i.e., L_1, K_1, L_2, and K_2. To be concise, we only report the results on Avazu. Similar trends can be observed from other data sets. In Figure 14, we show the clustering results of SILK with $L_1 \in \{8, 16, 24\}$ and $K_1 \in \{4, 6\}$. For each line we fix $K_2 = 4$ and vary $L_2 \in \{6, 8, 10\}$.

First of all, the increment of both L_1 and L_2 would lead to larger k. L_1 affects the number of buckets in the first level MinHash and thus has a significant impact on k. As L_2 affects the number of bins in the second level MinHash, it has a small factor contributed to k. K_1 and K_2 control the quality of buckets. Increasing them usually leads to smaller k as many infrequent points have been filtered, and the number of buckets/bins becomes smaller.

Parameter Tuning. To find the proper parameters for a specific k, we first search L_1 in coarse granularity to identify suitable values that produce a larger number of seeds than k. Then, we vary L_2 in a small range to approach the best results. As for K_1 and K_2, we empirically set them within $\{4, 6\}$ as the very large K_1 and K_2 would deteriorate the results rapidly by producing scarce buckets/bins.

6 RELATED WORK

Clustering is a very fundamental problem with extensive studies. A comprehensive survey can be referred to [92, 94]. Below, we first review the representative works regarding sparse data clustering with frequent items. Then, we discuss the pioneer works relevant to k-FreqItems and SILK. Finally, we discuss and analyze other popular clustering methods for sparse data.

6.1 Clustering with Frequent Items

The concept of frequent itemset was first proposed in Association Mining [1] and later has been adapted for sparse data clustering. In the frequent term-based text clustering [13], Beil et al. first applied Apriori [1] to find the frequent itemsets and then utilized the mutual overlap of frequent itemsets to cluster documents. Later, Fung et al. [32] combined the intermediate results of finding the frequent itemsets with hierarchical clustering to construct a topic hierarchy for greater interpretability of the results. Nevertheless, both methods need to find all frequent itemsets with different cluster supports in advance. Recently, Zhang et al. [101] proposed Maximum Capturing that employed the Minimum Spanning Tree (MST) to cluster documents based on co-occurrences of frequent itemsets. However, its scalability might be limited as the MST requires the complete similarity matrix of the data sets. As for k-FreqItems and SILK, we use the frequent items to represent the cluster center, which can be done in $O(n\bar{d})$ time.

6.2 Partitioning-based Methods

k-Means is a classic partitioning-based clustering algorithm initially introduced by MacQueen [75], which works on dense, numerical data. The most famous version is the *Lloyd's* algorithm [66]. As the data volume has grown swiftly, many k-Means variants have been developed to support large n and d [23, 29, 30, 38, 80, 81]. Recently, researchers also leverage modern hardware such as GPU [19, 63] and many-core supercomputers [36, 61] for acceleration. k-Modes [47, 83] is another prevalent partitioning-based clustering algorithm that targets dense, categorical data. To better handle the border of clusters, researchers also adopted the fuzzy theory for deciding the central and boundary objects of clusters [48, 56].

Spherical k-Means [27] is a popular k-Means variant for sparse data. It adopts cosine similarity as the metric. Researchers show its utility on sparse data with a few significant features [26, 40]. Nonetheless, recent methods [53, 79] observe that as the iterations proceed, its cluster centers will become dense as many infrequent features are averaged to the centers. This phenomenon might incur noisy features and cause more time and space overhead. To alleviate this problem, Kim et al. [53] preserved the center sparsity by applying a threshold to filter infrequent features. These works justify the importance of the center sparsity for sparse data clustering and support our center representation by frequent items.

6.3 Seeding Methods

As is well-known, a good seeding method is vital for the quick convergence of the k-Means type methods. A pioneer work is k-Means++ [7, 84], which takes only $\Theta(ndk)$ time. However, the k sequential iterations make it hard to parallel. Bahmani et al. [11] proposed k-Means||, which contains an overseeding step to reduce the k sequential iterations to $O(\log n)$ rounds. Later, Bachem et al. developed two fast and provably good seeding algorithms [8, 9] based on the Markov chain Monte Carlo sampling to approximate the D^2-sampling distribution. They also provided a comprehensive analysis of k-Means|| that bounds the quality of k seeds even for a small, constant number of rounds [10]. Recently, Cohen-Addad et al. [22] proposed a near-linear time method for k-Means++ seeding with rejection sampling.

The seeding methods discussed above are all focused on dense data. Recently, Kim et al. [53] proposed an overseeding approach for cosine similarity on sparse data. They first randomly select μk points from the data set as initial centers; then, they randomly choose a seed and prune those seeds close to it within a heuristic threshold; this pruning procedure is repeated until the size of remaining seeds is less than λk ($1 \leq \lambda < \mu$). This approach reduces the distance computation cost to $O(\mu k^2)$, but its clustering quality might be highly impacted by the random selection of the μk initial centers. SILK is inspired by k-Means||, but we utilize LSH functions for overseeding and obtain the seeds from similar buckets.

6.4 Other Clustering Methods for Sparse Data

Hierarchical clustering [35, 100] is one of the most widely used methods for sparse data, such as text data. Their results can form dendrograms to facilitate understanding these clusters, providing superior interpretability for applications like topic modeling. Nonetheless, building the hierarchy is usually expensive, thus limiting their scalability to tackle a large corpus of data directly. Spectral clustering [55, 82, 86] can also support sparse data since they work on the Laplacian graph that is extracted from the similarity matrix. The downside of spectral clustering is that constructing the similarity matrix is time-consuming for large-scale data and usually requires domain experts to build the matrix [55]. Besides, density-based clustering methods [6, 31, 34] can support sparse data as they work on the neighbor graph, but their performance may deteriorate rapidly due to the curse of dimensionality. In a recent study, Lulli et al. [72] introduced a novel ε-graph and proposed NG-DBSCAN, which employs a distributed implementation to construct the graph and circumvent the costly ε-neighborhood queries, making it feasible for large cardinality and various (symmetric) distance/similarity functions. However, NG-DBSCAN has limitations in handling high-dimensional, sparse data with more than one thousand dimensions, and its scalability is still inferior to partitioning-based methods. Thus, while these three kinds of methods exhibit commendable performance under their respective assumptions/conditions, they may not be well-suited for scenarios involving enormous cardinality and/or ultra-high dimensionality, which are the focus of our paper.

7 CONCLUSION

In this paper, we have addressed the challenge of clustering sparse data with large cardinality and ultra-high dimensionality. We have proposed a novel sparse center representation called FreqItem and introduced k-FreqItems for sparse data clustering based on Jaccard distance. We have validated the choice of α and demonstrated that FreqItem is a practical and efficient center representation for set data. Furthermore, we have utilized LSH functions for oversampling and developed SILK, a seeding technique for k-FreqItems. Our empirical results showcase the rapid convergence of SILK. We have also implemented SILK on a distributed platform and demonstrated its efficient scalability with an asymptotically linear relationship to the number of GPUs employed. In our experiments, SILK outperforms seven competitors and showcases excellent scalability even with 1 billion set data on a commodity machine equipped with 4 GPUs.

ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore under its Strategic Capability Research Centres Funding Initiative and the National University of Singapore Centre for Trusted Internet and Community Industry Collaborative Projects (CTIC-CP-21-01). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast algorithms for mining association rules. In *VLDB*. 487–499.
[2] Salem Alelyani, Jiliang Tang, and Huan Liu. 2018. Feature selection for clustering: A review. *Data Clustering* (2018), 29–60.
[3] Alexandr Andoni. 2005. E2LSH 0.1 User Manual. *http://web.mit.edu/andoni/www/LSH/index.html* (2005).
[4] Alexandr Andoni and Piotr Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*. 459–468.
[5] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. In *NIPS*. 1225–1233.
[6] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: ordering points to identify the clustering structure. *SIGMOD Record* 28, 2 (1999), 49–60.
[7] David Arthur and Sergei Vassilvitskii. 2007. k-means++: The advantages of careful seeding. In *SODA*. 1027–1035.
[8] Olivier Bachem, Mario Lucic, Hamed Hassani, and Andreas Krause. 2016. Fast and provably good seedings for k-means. In *NIPS*. 55–63.
[9] Olivier Bachem, Mario Lucic, S Hamed Hassani, and Andreas Krause. 2016. Approximate k-means++ in sublinear time. In *AAAI*. 1459–1467.
[10] Olivier Bachem, Mario Lucic, and Andreas Krause. 2017. Distributed and provably good seedings for k-means in constant rounds. In *ICML*. 292–300.
[11] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Scalable k-means++. *PVLDB* 5, 7 (2012), 622–633.
[12] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. 2005. LSH forest: self-tuning indexes for similarity search. In *WWW*. 651–660.
[13] Florian Beil, Martin Ester, and Xiaowei Xu. 2002. Frequent term-based text clustering. In *KDD*. 436–442.
[14] Ron Bekkerman and Martin Scholz. 2008. Data weaving: Scaling up the state-of-the-art in data clustering. In *CIKM*. 1083–1092.
[15] Aditya Bhaskara and Maheshakya Wijewardena. 2018. Distributed clustering via lsh based data partitioning. In *ICML*. 570–579.
[16] Mohamed Bouguessa and Shengrui Wang. 2008. Mining projected clusters in high-dimensional spaces. *TKDE* 21, 4 (2008), 507–522.
[17] Andrei Z Broder. 1997. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences*. 21–29.
[18] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. 1998. Min-wise independent permutations. In *STOC*. 327–336.
[19] Feng Cao, Anthony KH Tung, and Aoying Zhou. 2006. Scalable clustering using graphics processors. In *WAIM*. 372–384.
[20] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *STOC*. 380–388.
[21] Xiaojun Chen, Xiaofei Xu, Joshua Zhexue Huang, and Yunming Ye. 2011. TW-k-means: Automated two-level variable weighting clustering algorithm for multiview data. *TKDE* 25, 4 (2011), 932–944.
[22] Vincent Cohen-Addad, Silvio Lattanzi, Ashkan Norouzi-Fard, Christian Sohler, and Ola Svensson. 2020. Fast and accurate k-means++ via rejection sampling. In *NeurIPS*. 16235–16245.
[23] Ryan R Curtin. 2017. A dual-tree algorithm for fast k-means clustering with large k. In *SDM*. 300–308.
[24] Sanjoy Dasgupta. 2008. *The hardness of k-means clustering*. Department of Computer Science and Engineering, University of California, San Diego.
[25] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*. 253–262.
[26] Inderjit S Dhillon, Yuqiang Guan, and Jacob Kogan. 2002. Iterative clustering of high dimensional text data augmented by local search. In *ICDM*. 131–138.
[27] Inderjit S Dhillon and Dharmendra S Modha. 2001. Concept decompositions for large sparse text data using clustering. *Machine Learning* 42, 1 (2001), 143–175.
[28] Chris Ding, Tao Li, Wei Peng, and Haesun Park. 2006. Orthogonal nonnegative matrix t-factorizations for clustering. In *KDD*. 126–135.
[29] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In *ICML*. 579–587.
[30] Charles Elkan. 2003. Using the triangle inequality to accelerate k-means. In *ICML*. 147–153.
[31] 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*. 226–231.

[32] Benjamin CM Fung, Ke Wang, and Martin Ester. 2003. Hierarchical document clustering using frequent itemsets. In *SDM*. 59–70.

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

[34] Junhao Gan and Yufei Tao. 2015. DBSCAN revisited: Mis-claim, un-fixability, and approximation. In *SIGMOD*. 519–530.

[35] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. 1998. CURE: An efficient clustering algorithm for large databases. *SIGMOD Record* 27, 2 (1998), 73–84.

[36] Ali Hadian and Saeed Shahrivari. 2014. High performance parallel k-means clustering for disk-resident datasets on multi-core CPUs. *The Journal of Supercomputing* 69, 2 (2014), 845–863.

[37] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. 2009. Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions. (2009).

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

[39] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. 2012. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of Computing* 8, 1 (2012), 321–350.

[40] Kurt Hornik, Ingo Feinerer, Martin Kober, and Christian Buchta. 2012. Spherical k-means clustering. *Journal of Statistical Software* 50 (2012), 1–22.

[41] Qiang Huang, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2017. Two efficient hashing schemes for high-dimensional furthest neighbor search. *TKDE* 29, 12 (2017), 2772–2785.

[42] Qiang Huang, Jianlin Feng, Qiong Fang, Wilfred Ng, and Wei Wang. 2017. Query-aware locality-sensitive hashing scheme for l_p norm. *VLDBJ* 26, 5 (2017), 683–708.

[43] 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.

[44] Qiang Huang, Yifan Lei, and Anthony KH Tung. 2021. Point-to-Hyperplane Nearest Neighbor Search Beyond the Unit Hypersphere. In *SIGMOD*. 777–789.

[45] 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.

[46] Qiang Huang, Yanhao Wang, and Anthony KH Tung. 2022. SAH: Shifting-aware Asymmetric Hashing for Reverse k-Maximum Inner Product Search. *arXiv preprint arXiv:2211.12751* (2022).

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

[48] Zhexue Huang and Michael K Ng. 1999. A fuzzy k-modes algorithm for clustering categorical data. *IEEE Transactions on Fuzzy Systems* 7, 4 (1999), 446–452.

[49] Lawrence Hubert and Phipps Arabie. 1985. Comparing partitions. *Journal of Classification* 2, 1 (1985), 193–218.

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

[51] Liping Jing, Michael K Ng, and Joshua Zhexue Huang. 2007. An entropy weighting k-means algorithm for subspace clustering of high-dimensional sparse data. *TKDE* 19, 8 (2007), 1026–1041.

[52] Leonard Kaufman and Peter J Rousseeuw. 2009. *Finding groups in data: an introduction to cluster analysis*. Vol. 344. John Wiley & Sons.

[53] Hyunjoong Kim, Han Kyul Kim, and Sungzoon Cho. 2020. Improving spherical k-means for document clustering: Fast initialization, sparse centroid projection, and efficient cluster labeling. *Expert Systems with Applications* 150 (2020), 113288.

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

[55] Peeyush Kumar, N Narasimhan, and Balaraman Ravindran. 2013. Spectral clustering as mapping to a simplex. In *2013 ICML Workshop on Spectral Learning*.

[56] Ren-Jieh Kuo, YR Zheng, and Thi Phuong Quyen Nguyen. 2021. Metaheuristic-based possibilistic fuzzy k-modes algorithms for categorical data clustering. *Information Sciences* 557 (2021), 1–15.

[57] Ken Lang. 1995. NewsWeeder: learning to filter netnews. In *ICML*. 331–339.

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

[59] Yifan Lei, Qiang Huang, Mohan Kankanhalli, and Anthony KH Tung. 2020. Locality-Sensitive Hashing Scheme based on Longest Circular Co-Substring. In *SIGMOD*. 2589–2599.

[60] David D Lewis, Yiming Yang, Tony Russell-Rose, and Fan Li. 2004. RCV1: A new benchmark collection for text categorization research. *JMLR* 5 (2004), 361–397.

[61] 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 *Proceedings of the International*

Conference for High Performance Computing, Networking, Storage, and Analysis. 160–170.
[62] Qiuhong 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.
[63] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. 2013. Speeding up k-means algorithm by GPUs. *J. Comput. System Sci.* 79, 2 (2013), 216–229.
[64] Weiwei Liu, Xiaobo Shen, and Ivor W Tsang. 2017. Sparse embedded k-means clustering. In *NIPS*. 3321–3329.
[65] Wanqi Liu, Hanchen Wang, Ying Zhang, Wei Wang, Lu Qin, and Xuemin Lin. 2021. EI-LSH: An early-termination driven I/O efficient incremental c-approximate nearest neighbor search. *The VLDB Journal* 30 (2021), 215–235.
[66] Stuart Lloyd. 1982. Least squares quantization in PCM. *TIT* 28, 2 (1982), 129–137.
[67] Kejing Lu, Yoshiharu Ishikawa, and Chuan Xiao. 2022. MQH: Locality Sensitive Hashing on Multi-level Quantization Errors for Point-to-Hyperplane Distances. *PVLDB* 16, 4 (2022), 864–876.
[68] Kejing Lu and Mineichi Kudo. 2020. R2LSH: A Nearest Neighbor Search Scheme Based on Two-dimensional Projected Spaces. In *ICDE*. 1045–1056.
[69] 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.
[70] Meilian Lu, Zhen Qin, Yiming Cao, Zhichao Liu, and Mengxing Wang. 2014. Scalable news recommendation using multi-dimensional similarity and Jaccard–Kmeans clustering. *Journal of Systems and Software* 95 (2014), 242–251.
[71] Carlos B Lucasius, Adrie D Dane, and Gerrit Kateman. 1993. On k-medoid clustering of large data sets with the aid of a genetic algorithm: background, feasiblity and comparison. *Analytica Chimica Acta* 282, 3 (1993), 647–669.
[72] Alessandro Lulli, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. 2016. NG-DBSCAN: scalable density-based clustering for arbitrary data. *PVLDB* 10, 3 (2016), 157–168.
[73] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*. 950–961.
[74] 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.
[75] James MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*. 281–297.
[76] Ryan McConville, Xin Cao, Weiru Liu, and Paul Miller. 2016. Accelerating large scale centroid-based clustering with locality sensitive hashing. In *ICDE*. 649–660.
[77] Nicholas Meisburger and Anshumali Shrivastava. 2020. Distributed Tera-Scale Similarity Search with MPI: Provably Efficient Similarity Search over billions without a Single Distance Computation. *arXiv preprint arXiv:2008.03260* (2020).
[78] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *JMLR* 17, 1 (2016), 1235–1241.
[79] Arman Khadjeh Nassirtoussi, Saeed Aghabozorgi, Teh Ying Wah, and David Chek Ling Ngo. 2015. Text mining of news-headlines for FOREX market prediction: A Multi-layer Dimension Reduction Algorithm with semantics and sentiment. *Expert Systems with Applications* 42, 1 (2015), 306–324.
[80] James Newling and François Fleuret. 2016. Fast k-means with accurate bounds. In *ICML*. 936–944.
[81] James Newling and François Fleuret. 2016. Nested mini-batch k-means. In *NIPS*. 1352–1360.
[82] Andrew Y Ng, Michael I Jordan, and Yair Weiss. 2001. On spectral clustering: Analysis and an algorithm. In *NIPS*. 849–856.
[83] Michael K Ng, Mark Junjie Li, Joshua Zhexue Huang, and Zengyou He. 2007. On the impact of dissimilarity measure in k-modes clustering algorithm. *TPAMI* 29, 3 (2007), 503–507.
[84] Rafail Ostrovsky, Yuval Rabani, Leonard J Schulman, and Chaitanya Swamy. 2006. The Effectiveness of Lloyd-Type Methods for the k-Means Problem. In *FOCS*. 165–176.
[85] David Sculley. 2010. Web-scale k-means clustering. In *WWW*. 1177–1178.
[86] Jianbo Shi and Jitendra Malik. 2000. Normalized cuts and image segmentation. *TPAMI* 22, 8 (2000), 888–905.
[87] Anshumali Shrivastava and Ping Li. 2014. Asymmetric LSH (ALSH) for sublinear time Maximum Inner Product Search (MIPS). In *NIPS*. 2321–2329.
[88] Anshumali Shrivastava and Ping Li. 2014. Densifying one permutation hashing via rotation for fast near neighbor search. In *ICML*. 557–565.
[89] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *PVLDB* 8, 1 (2014), 1–12.
[90] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*. 563–576.
[91] Yao Tian, Xi Zhao, and Xiaofang Zhou. 2022. DB-LSH: Locality-Sensitive Hashing with Query-based Dynamic Bucketing. In *ICDE*. 2250–2262.

[92] 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.
[93] Nguyen Xuan Vinh, Julien Epps, and James Bailey. 2010. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *JMLR* 11 (2010), 2837–2854.
[94] Dongkuan Xu and Yingjie Tian. 2015. A comprehensive survey of clustering algorithms. *Annals of Data Science* 2, 2 (2015), 165–193.
[95] Hui Yan, Siyu Liu, and S Yu Philip. 2019. From joint feature selection and self-representation learning to robust multi-view subspace clustering. In *ICDM*. 1414–1419.
[96] Wei Ye, Samuel Maurus, Nina Hubig, and Claudia Plant. 2016. Generalized independent subspace clustering. In *ICDM*. 569–578.
[97] Rong Yin, Yong Liu, Weiping Wang, and Dan Meng. 2020. Extremely sparse Johnson-Lindenstrauss transform: From theory to algorithm. In *ICDM*. 1376–1381.
[98] Kevin Y Yip, David W Cheung, and Michael K Ng. 2005. On discovery of extremely low-dimensional clusters using semi-supervised projected clustering. In *ICDE*. 329–340.
[99] Hwanjo Yu, Jiong Yang, and Jiawei Han. 2003. Classifying large data sets using SVMs with hierarchical clusters. In *KDD*. 306–315.
[100] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: an efficient data clustering method for very large databases. *SIGMOD Record* 25, 2 (1996), 103–114.
[101] Wen Zhang, Taketoshi Yoshida, Xijin Tang, and Qing Wang. 2010. Text clustering using frequent itemsets. *Knowledge-Based Systems* 23, 5 (2010), 379–388.
[102] 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.
[103] Xi Zhao, Bolong Zheng, Xiaomeng Yi, Xiaofan Luan, Charles Xie, Xiaofang Zhou, and Christian S Jensen. 2023. FARGO: Fast Maximum Inner Product Search via Global Multi-Probing. *PVLDB* 16, 5 (2023), 1100–1112.
[104] 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.
[105] Yuxin Zheng, Qi Guo, Anthony KH Tung, and Sai Wu. 2016. LazyLSH: Approximate Nearest Neighbor Search for Multiple Distance Functions with a Single Index. In *SIGMOD*. 2023–2037.

Received April 2022; revised July 2022; accepted August 2022