keybin
Key-based Binning for Distributed Clustering

Xinyu Chen*, Jeremy Benson† and Trilce Estrada‡
Department of Computer Science, University of New Mexico
Email: *xychen@cs.unm.edu, †jeremybenson@cs.unm.edu, ‡estrada@cs.unm.edu

***Abstract*—Traditional machine learning algorithms often require computations on centralized data, but modern datasets are collected and stored in a distributed way. In addition to the cost of moving data to centralized locations, increasing concerns about privacy and security warrant distributed approaches. We propose *keybin*, a distributed key-based binning clustering algorithm for high-dimensional spaces. keybin locally generates a spatial *key* for each data point across all dimensions without needing knowledge of other data. Then, it performs a conceptual Map-Reduce procedure in the index space to form a global clustering assignment. We present an implementation and a case study on the capabilities and limitations of this approach, showing that this algorithm can learn a global clustering structure with limited communication and can scale with the dimensionality and size of data sets.**

I. Introduction

Modern data acquisition techniques, the need for high performance analysis, and constraints regarding data management are slowly rendering centralized data analysis obsolete. In domains like climate simulations, high-energy physics, astronomy, and remote sensing, data transfer represents a major bottleneck. In medical or financial domains, policies regarding security and privacy impede data openness, even when volume is not a concern.

A true and tried principle in the Big Data era is to minimize data movement. Coupling data analytics with simulations, or placing them alongside data acquisition tasks, yields large performance gains [1], [2], [3], [4]. While some algorithmic approaches have been successfully used to decentralize data analysis in specific domains, they still lack in at least one of the following core aspects of distributed learning: scalability, accuracy, or generality. For example, many traditional machine learning and data mining approaches rely on expensive training phases and often require gathering raw data in a centralized [5], [6], [7] or semi-centralized [8], [9] way. Existing distributed approaches either require synchronized communication [10], [11], [8] or iterate for a long time, incurring large communication costs among parameter servers [12], [13]. Others are tied to a particular domain and do not generalize [14], [15], sacrifice accuracy for the sake of scalability [16], [17], [18], or are affected by the *curse of dimensionality* [19] (i.e., scaling the number of data features negatively impacts an algorithm's predictive capabilities [20]).

In this paper, we present keybin, a scalable and accurate clustering algorithm, suitable for distributed and privacy constrained environments. Our method is scalable horizontally (i.e., every data sample can be processed in parallel without requiring any direct knowledge of other data samples). It is also, to a certain extent, scalable vertically (i.e., every feature, or dimension, can be processed independently). Our clustering method is able to organize large distributed datasets in a single pass and requires minimum communication.

The idea behind keybin is to map every data point into a multidimensional key in space. Keys represent bins, whose densities reflect a particular spatial data agglomeration. Multiple bins are built for every dimension and are associated with only that dimension. Assuming a number of distributed sites, each with their own local data, the clustering process is:

1) Every data point in a distributed site updates its corresponding bins in space in a fully parallel way; data points and features can be processed locally and independently.
2) Distributed sites communicate with each other or with a master to consolidate a global view of bin densities.
3) As every distributed site updates its model, local statistics are computed, noisy or uninformative dimensions are collapsed, and final clustering assignments are made.

Communication among sites in the second and third phases is performed only once, and transferred data contains only aggregated information that is considerably smaller than the raw datasets. The time complexity of our algorithm is linear with respect to both the number of points and the number of dimensions. More specifically, the processing of each point is done in constant time. Unlike most distance-based clustering methods, our algorithm can benefit from a large number of dimensions and data volume. Our contributions are:

- A distributed linear time complexity clustering approach that is able to group data across multiple dimensions, even with a very limited view of the data.
- A probabilistic method to identify and discard noisy or uninformative dimensions.
- An experimental study of our algorithm's scalability and accuracy using stress-testing and comparison with other clustering algorithms.

The remainder of this paper is organized as follows: in Section 2, we summarize some influential algorithms and related approaches. Section 3 presents our method. In Section 4 we cover experimental results. Section 5 presents our discussion regarding our algorithm's limitations, reasoning behind parameter selection, and overview of ongoing and future work. Finally, Section 6 concludes the paper.

II. Related Work

Lazy learning and nearest neighbors are easily adapted for clustering on Big Data because they adapt automatically when a new sample is received and there is no training cost associated with them. In practice, however, these types of algorithms do not scale well with massive amounts of data, as producing a local model every time a sample needs to be classified is computationally expensive. This problem has been addressed and partially mitigated by Zhang *et.al.* [21], who proposed a lazy tree that dynamically maintains high-level summaries of historical stream records and classifies new samples in sub-linear time complexity.

Other optimizations for the nearest neighbors algorithm include flexible distance NN [22], meaningful NN [23], and approximate KNN [20]. Lazy and nearest neighbor approaches are affected by storage and memory constraints, and while the problem is mitigated with hybrid approaches, a lazy approach still fails for systems collecting massive amounts of data in a distributed way. Semantic Hashing [6], is a deep graphical model used as a variation of nearest neighbors that requires a small portion of additional data for each input. For each query a shortlist of that data which is closest to the particular input is used. The search time is independent of the size of the input collection and linear with respect to the size of the shortlist. Even though training time is done one layer at a time, it can be computationally expensive if the input corpus changes dynamically. Distributed approaches like [11], [12], [24] process data in an iterative way and rely on parameter servers to converge. The main drawback of these approaches is the constant communication required over a long training phase.

Density-based clustering methods are closely related to our approach. DBSCAN [25] can find underlying clustering structure without a-priori knowledge of the number of clusters, a useful property in cases where some sites may only contain a subset of the global structures. It should be noted here that the knowledge of total number of global clusters may even be misleading for algorithms like K-means. Finding the optimal parameter of ϵ and *min support* is a process of trial and error.

BD-CATS [26] and PDSDBSCAN [27] are two recent parallel density-based clustering algorithms that rely on disjoint-set data structures to represent clusters and union-find techniques to build local clusters, merging them across nodes. These algorithms perform well at scale through sophisticated load balancing; they handle clustering for large cosmology and plasma physics simulations. However, both algorithms require global knowledge of the data and require partitioning in the preprocessing steps. Moreover, the inter-node raw data exchange leads away from our focus scenario of distributed analytics under privacy constraints.

KOTree [14], an N-dimensional tree for Knowledge Organization, also performs density-based clustering, but it is tied to a particular domain and does not generalize. Some efficient dimensionality reduction techniques address the issue of scalability by using prior knowledge of the data (e.g., principal components, covariance matrix, or other statistics). Feature selection techniques such as matrix factorization [16], locality sensitive hashing[17] and random projection [18] reduce dimensionality in order to achieve scalability at the expense of accuracy. But when the i.i.d. property cannot be guaranteed, like when data needs to be analyzed as a stream, *in situ* or is stored in geographically distributed locations, these methods fail.

Clustering techniques that consider privacy preservation have been studied for a decade [28], [29], [30]. Efforts in randomization and permutation represent the bulk of anonymization methods, but these operations do not necessarily preserve privacy since the individual data can be recovered from spectral filtering [31]. Using cryptographic techniques in two-party or multi-party communications is another approach, but it burdens communication further. Moreover, these methods still need to move vast datasets to centralized computations and are not able to work at scale.

Index-based [32] and grid-based **hierarchical clustering** algorithms [33], [34] also provide insight towards privacy. Each individual site first builds lower-level clustering hierarchies locally. The global structure can be merged later based on knowledge extracted from lower levels. *Sub-space Clustering* algorithms are closely related and the most influential for us. CLIQUE [35] and MAFIA [36] first find dense regions in lower-dimensional spaces. Then they merge these lower-dimensional regions into bigger higher-dimensional hypercubes if they can find a common-face between two regions. If sites only exchange knowledge of common-faces between regions, the privacy can be well preserved. The limitation of these two algorithms is the expensive combinations of all possible lower-dimensional regions, so scalability drops with rise in dimensionality.

III. Algorithm

Most traditional clustering techniques rely on computing pairwise distances between points or regions to form clusters and are exponentially expensive as the number of points and the number of dimensions grow. When data are collected and stored on distributed sites, these traditional clustering techniques fail to scale. So, our question becomes: *How can we decide whether two points are similar?* We took inspiration from Locality Sensitive Hashing [17] to answer this. We map data points to indices along every dimension (i.e., every feature represents one dimension). By organizing these indices into bins and merging to primary clusters, we generate a list of primary cluster identifiers, which we call a *key*. Then, forming clusters can be done in the space of keys. The a-priori algorithm for mining frequent patterns [37] is behind our final grouping approach; points that belong to one higher-dimensional cluster also stay together in each lower-dimensional region. Points with different keys belong to different clusters. Two crucial byproducts of this approach are:

1) Keys represent a summarized knowledge of the raw data, which allows us to preserve privacy

2) The reduction on keys (instead of distance comparisons) allows us to improve scalability

Assume we want to cluster a dataset d^0 of size $M \times N$, where M is the number of data points and N is the number of features. We denote $d^0 = \{x_{i,j} \mid i < M, j < N\}$, with i being the unique identity of a data point with feature j. Without lack of generality we could assume $M = 1$ for a data stream scenario or multiple d's for a distributed case. We call d our *raw data* for the rest of the paper and we use superscripts to denote distributed datasets ($d^0, d^1, d^2, ..., d^K$, for k distributed sites). The steps of our method are as follows:

1) **Assigning a key to a point**. Using point features, the point is assigned to multiple indices, each corresponding to a bin in the specific dimension. The ordered set of indices represents a key. As points get their indices, bins update their density. This step only involves the point itself and the value range of each dimension. In the distributed scenario, this is only done with local data.
2) **Computing global densities**. In the distributed scenario, distributed sites communicate their computed bin densities either to a master site or among each other in order to consolidate an integrated view of the data. This updated model is broadcast back to the distributed sites.
3) **Collapsing dimensions**. By analyzing the distribution of bin densities per dimension, we determine which dimensions converge into a similar clustering pattern and which fall out of the norm. Noisy or uninformation dimensions are collapsed, improving clustering accuracy.
4) **Building primary clusters**. A primary cluster is a partial clustering assignment viewed from one particular dimension. This step merges adjacent bins into primary clusters if they exceed a density threshold.
5) **Assigning points to final clusters**. Once primary clusters are built, every point can be mapped to a specific set of primary clusters in all of its dimensions, leading to a final global clustering assignment.

When the distribution of data points is skewed, a primary cluster may be under-represented by only a few points and may be assigned as outliers or noise by other algorithms. With the aggregated global knowledge, the corresponding primary cluster can be constructed and the points will not be assigned as noise. Adding a new point can be done in constant time, as it is equivalent to calculating the key and querying the cluster label. Updating clustering assignments on the fly is also trivial, as new clusters can be created in empty regions by updating their respective bin densities while smaller clusters can be merged as they grow in size.

A. Assigning a key to a point

This step is performed with local data; we assume one dataset d. For point $x_i \in d$, $x_{i,j}$ denotes its jth feature. Our method converts a point's coordinates into a list of binary indices (idx), indicating a relative location along each dimension. The number of bits per index is defined by the user as the $depth$ of our algorithm. Every index is associated with a bin, and the $depth$ determines the number of bins per dimension, as $B = 2^{depth}$. $idx_{i,j}$, which is the index for $x_{i,j}$, is in range of $[0, 2^{depth} - 1]$.

The kernel *getKey* (Algorithm 1), intakes the $depth$, an estimated $lower$ and $upper$ boundaries representing the range of the specific dimension, and the point's jth feature. Then, it recursively divides the dimension's range into equal halves for $depth$ number of steps. This procedure is conceptually building a $depth$-deep binary tree to accommodate the range. Each leaf contains a sub region of the bounded dimension and every $x_{i,j}$ is associated to a leaf. The *getKey* kernel is applied to $x_{i,j} \forall i, j$ independently. Thus, it can be efficiently implemented in parallel per data point and feature.

Algorithm 1 getKey

```
1:  procedure GETKEY(max_depth, lower, upper, x_{i,j})
2:      for depth < max_depth do
3:          μ_{j,depth} ← 1/2(lower + upper)
4:          if x_{i,j} ≥ μ_{j,depth} then
5:              append 1 to idx_{i,j}
6:              lower ← μ_{j,depth}
7:          else
8:              append 0 to idx_{i,j}
9:              upper ← μ_{j,depth}
10:     return idx_{i,j}                          ▷ return leaf index
```

getKey converts the value of $x_{i,j}$ to an index $idx_{i,j}$. Given $depth = 6$, points in the upper-right-most cell will have binary indices $(111111, 111111)$ for x and y dimensions respectively, which corresponds to bins $63, 63$. It is worth noting that even though our algorithm needs an estimated lower and upper bounds for each dimension, these values can be simple estimates. If the approximate bounds are tighter than the actual bounds, outermost points will be put into the same bin. This means we could also put outliers into the outermost bins. If the approximate bounds are wider than the actual boundary, inner points will be squeezed into the same bin, meaning we could lose resolution for smaller $depth$ values. However, for most real-world applications, we typically have an appropriate estimation of range values.

The produced indices serve as bin identifiers. The algorithm accumulates densities of bins according to the number of data points falling into them. For a given data point, the concatenation of indices represents its key. Once the local sites calculate a key for each of their data points and compute bin densities, this is the last time that raw data is needed in the clustering process. Note that key calculation is a lightweight, constant-time process that does not exhibit data dependencies. From this point on, all the communication and calculations are done over the key space and bin densities. For a streaming, in-situ, or in-transit clustering algorithm, key generation can be directly coupled with simulations or data acquisition. Bins can be kept in memory and raw data can be immediately sent to secondary storage.

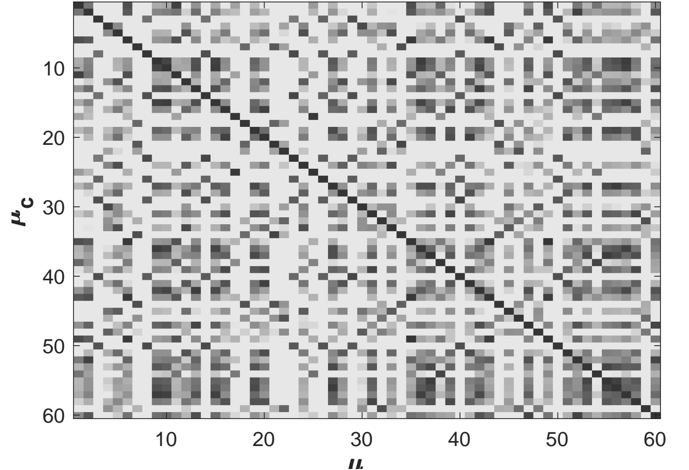
B. Computing global densities

Calculation of bin densities in the local sites can be viewed as a partial view of the underlying cluster structures of the process being studied. In this step, we aggregate densities from all sites into a comprehensive global view for the later clustering task. There are two cases: (1) all the sites worked with the same parameters to build keys and bins or (2) parameters were not consistent across sites. In the first case, integration is trivial; all the corresponding bins are summed up. In the second case the process is just marginally more complex: given the different upper and lower bounds for each site, we need to calculate the overall range, the set of global partitions, and the set of bins that correspond to each partition. After that, bins can be summed up per partition. The updated densities are sent back to all the distributed sites, where the clustering assignment is finalized, as explained in Sections III-D and III-E. Communication per site comprises the list of bins, which is $O(N*2^{depth})$ where N is the number of dimensions and $depth$ is the depth of the indices. If there are K sites, the communication is $O(2K*N*2^{depth}) << O(M*N)$ where M is the number of points. For example, if we had a floating point (i.e. 4 bytes per record) 1000-dimensional dataset distributed among different locations, each with 1 million points, that is a 4 GB dataset per site, assuming a depth of 8, our algorithm would transfer about 2 MB per site. Now, assuming 1 billion points (4 TB) per site, our algorithm would still transfer about 2 MB per site.

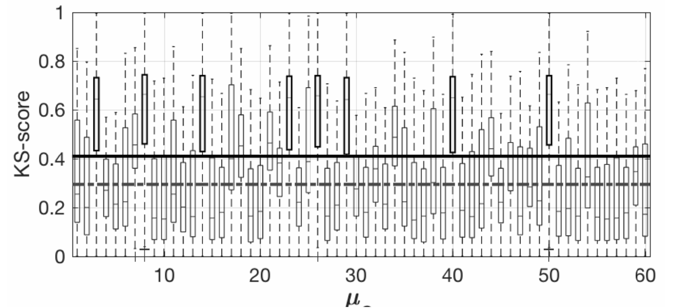
For simplicity, we assume a centralized topology through this paper (i.e., multiple distributed sites and one master site that performs the aggregation of bins). However, keybin does not necessarily rely on a master-worker topology to calculate the global model. A ring topology, for example, would work in a fairly similar way. Assuming sites s are numbered from 0 to $K-1$, site s^0 would send its partial view to s^1. s^1 would integrate both views and send the updated result to s^2, and so on. Two passes around the ring are enough to consolidate a global view of the data. A hierarchical approach is also possible by arranging sites as if they were leaves in a binary tree and communicating densities up until reaching the root. In these scenarios, communication is $O(K)$ and $O(Klog(K))$, respectively (explicitly, $O(4K*N*2^{depth})$ and $O(2K*N*2^{depth}log(K*N*2^{depth}))$). Even in this decentralized approach, where communication is done across all the different distributed sites, privacy is still preserved. Since sites only communicate information in an aggregate form, raw data cannot be reconstructed. keybin saves bandwidth and storage, as communication is not iterative and a few passes are enough to achieve convergence. Also, the amount of information sent is orders of magnitude smaller than a raw or even a compressed dataset.

C. Collapsing dimensions

An important challenge in high-dimensional data is that many dimensions or combinations of dimensions may have uninformative values or noise. These type of dimensions are especially problematic for clustering tasks, since a cluster, by definition, is made up of data points that are similar by some metric. Similarity in a high-dimensional space is affected by the curse of dimensionality. As the number of dimensions increases, the space becomes increasingly sparse. In a sparse high-dimensional space, very similar data points could become distant as meaningless dimensions are added. The opposite is also true - as uninformative dimensions are removed, distant points could become closer, exposing well defined patterns and groupings.



(a) KS Matrix for 60 dimensions (darker colors = higher similarity)



(b) Box and whiskers plot of KS scores used for the rejection rule

Fig. 1: Collapsing uninformative or noisy dimensions using a 2-sample KS test

Our hypothesis is that given an underlying clustering process, it will generate similar grouping patterns across most of its dimensions. We also hypothesize that noisy or uninformative dimensions will exhibit a different behavior than most of the other dimensions. To test our hypothesis, we needed to quantify the relative deviation between every pair of dimensions and exclude those that behave like outliers. Note that, since every distributed site has received the same global information, this process can be done locally.

To determine which dimensions are candidates to be collapsed, we used the computed bins densities to form an empirical distribution per dimension, and the Kolmogorov-Smirnov [38] test (KS-test) to determine when two samples

differ significantly. An important advantage of using the KS-test for our purpose of determining outlier dimensions is that this test does not make any assumption regarding the distribution of data (i.e., is non-parametric and distribution free). Specifically, we use the two-sample KS-test, which is sensitive to differences in both location and shape of the empirical cumulative distribution functions of the two samples. For every pair of dimensions, we compute the KS statistic given by:

$$D_{s_1,s_2} = sup_p |F_{1,s_1}(p) - F_{2,s_2}(p)|$$

where s_1 and s_2 are the sizes and F_{1,s_1} and F_{2,s_2} are the empirical distribution functions of the first and the second sample, (i.e., first and second dimension), and *sup* is the supremum function or *least upper bound* between the two empirical distributions. For each dimension $j < N$, we use its list of bins to compute an empirical distribution F_{j,s_j}. Then, for each subsequent dimension $g \mid g = j+1, j+2, ..., N$, we compute the respective empirical distribution F_{g,s_g}. Afterwards, we calculate the two-sample KS-statistic between F_{j,s_j} and F_{g,s_g} and update positions (j, g) and (g, j) of a symmetric matrix KS of KS-statistics (as seen in Figure 1a for 60 dimensions).

In order to blacklist potentially noisy or uninformative dimensions, we statistically analyze KS. For each column in KS, we compute its mean (i.e., $\mu_c = [\mu_{c,1}, \mu_{c,2}, ..., \mu_{c,N}]$). The expectation of all μ_c's would be analogous to the most common behavior across dimensions. However, since we anticipate that noisy dimensions will behave like outliers, we instead calculate the median (m_{ks}) and standard deviation (σ_{ks}) of μ_c and use it as the canonical expected behavior for all the dimensions. The rejection rule for a particular dimension j is *if* $\mu_{c,j} > m_{ks} + \gamma\sigma_{ks}$, *then* blacklist dimension j. In the rejection rule, γ is a constant factor that determines the rejection boundary. For the traditional definition of outliers γ is equal to 2, but the rule can be more lenient or more strict for larger or smaller values of γ, respectively.

Figure 1b depicts the rejection rule in action, where each box and whisker diagram represents the empirical distribution of each dimension and horizontal lines inside of the boxes are the dimension means (μ_c's). The dotted horizontal line represents the median of the means (m_{ks}) and the black solid line represents the rejection boundary for $\gamma = 1$. Boxes in bold are dimensions for which we added random noise.

D. Building primary clusters

After communicating global bin densities (*bins*), primary clusters are locally built in every distributed site. Primary clusters are sets of bins organized in a partial clustering assignment for a single dimension. Unlike CLIQUE [35] and MAFIA [36], we build primary clusters on each dimension, then directly reduce to the entire dimensional clusters, skipping all intermediate lower-dimensional dense regions. A bin is instantiated only when there is a data point whose index on this dimension is associated to that bin. $bins_{b,j}$ is the density of bin b in the jth dimension.

To build primary clusters, we merge adjacent bins if their density is larger than a predefined threshold *minD*. For very sparse clustering, this threshold can be set to zero. A primary cluster represents an agglomeration of points viewed from one particular dimension. We refer to them as PC where each element PC_q contains a bin identifier, denoting the starting of a bin partition that extends until PC_{q+1} in the specific dimension (e.g., if $PC_1 = 25$ and $PC_2 = 43$ it means that PC_1 contains an agglomeration of bins from 25 to 42).

Algorithm 2 shows the procedure for building primary clusters. It has 2 main sections: a for loop traversing all the dimensions, excluding those that have been blacklisted, and a second for loop merging adjacent bins forming primary clusters; neither have data dependencies, and thus can be done in parallel. The input bin densities are considerably smaller than raw data and have no sensitive information that could compromise privacy.

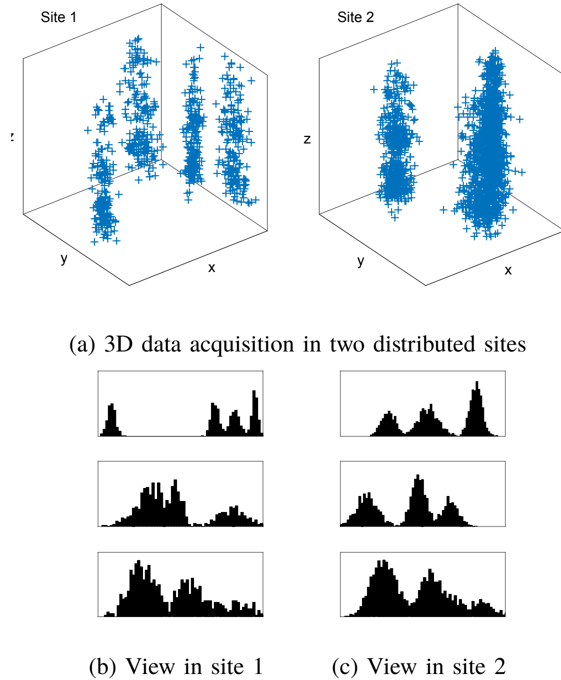
Algorithm 2 Build Primary Clusters

```
1:  N                                          ▷ number of dimensions
2:  minD                                       ▷ minimum density in bin
3:  B = 2^depth                                ▷ number of bins
4:  bins ← updatedDensitiesFromMaster()
5:  blacklist ← collapseDimensions()
6:  procedure BUILDPCs(bins, blacklist, B, N, minD)
7:      PC ← 0                                 ▷ primary clusters
8:      q ← 0                                  ▷ counter of PCs
9:      for j = 1 to N do          ▷ merge bins for every dimension
10:         if j ∉ blacklist then
11:             b ← 1, flag ← false
12:             for b = 1 to B do      ▷ for bins in dimension c
13:                 if bins_{b,j} > minD AND notflag then
14:                     PC_q ← b       ▷ gets first bin in sequence
15:                     flag ← true
16:                 else if bins_{b,j} <= minD AND flag then
17:                     q ← q + 1                ▷ initialize next PC
18:                     flag ← false
19:     return PC                      ▷ return primary clusters
```

E. Assigning points to final clusters

The final step makes a final assignment of points to their respective clusters on each site. The aggregated clusters comprise the entire high-dimensional space, with exception of the blacklisted dimensions. The key for a *final cluster* is the concatenation of PC_q's of each one-dimensional primary cluster. The intuition being that if points belong together in a high dimensional cluster, they will be together with high frequency in lower dimensional clusters. Although the possible combination of primary cluster keys is vast, the real worst case is each point forms a unique "group". So we can bound the maximum number of *final clusters* to be M. Since, in this case, the bin densities that all sites use to build primary clusters and then final clusters is broadcast by the master, then it is guaranteed that the final set of final clusters will be the same across sites. This guarantee holds even if data is highly skewed or even if some clusters are not represented at all in some sites.

We illustrate the whole procedure with a didactic example. Consider a process generating data in a three dimensional space. The data is collected and stored in two different sites. In *Step 1* we compute bin frequencies on each site. The local views are shown in Figure 3. In this case, where data samples in both sites are not identically sampled, the distribution across dimensions is different from one site to the other.

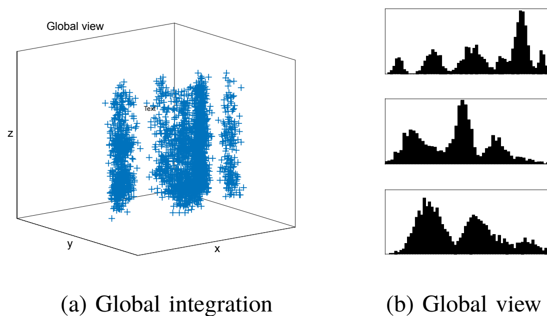


(a) 3D data acquisition in two distributed sites

(b) View in site 1 (c) View in site 2

Fig. 2: Step 1. Computing keys and local bin densities

In *Step 2* a global view is aggregated by combining all partial views from the distributed sites. Figure 3a shows the global 3D process, and Figure 3b shows the global integrated view of the data.



(a) Global integration (b) Global view

Fig. 3: Step 2. Computing an integrated view

In this example, the combined global view is not separable, as dimension z is made up of random Gaussian noise. If dimension z was used during the final clustering assignment, the number of possible partitions would be 54, as there are three different modes. In that case, most of the real clusters will be partitioned on average by a factor of 3. *Step 3* of our algorithm deals with this problem, by statistically analyzing all dimensions and collapsing those that are deemed to be uninformative. Figure 4 shows the 3D process collapsed into two dimensions. Histograms on the sides show the bin densities for the two remainding dimensions. These densities are used in *Steps 4 and 5* to generate primary clusters and to compute the final clustering assignment.

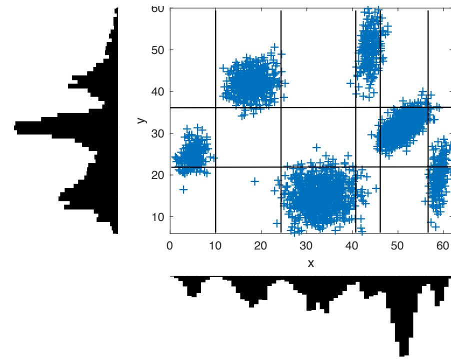


Fig. 4: Steps 3, 4, 5. Clustering formation

IV. Implementation

We implemented keybin using mpi4py and a master-worker topology. The program receives two input parameters: *depth* and *minD*. *depth* decides the bin width per dimension, which is the resolution of the algorithm. As *depth* grows, keybin is able to explore the space at a finer resolution, which decreases performance and in some cases, can over-partition clusters. The optimal value of *depth* for a given dataset is affected by the number and shape of the underlying clustering structures, the number of points, the number of features, and how separable the clusters are. The other parameter, *minD* decides if two adjacent bins can be merged into primary clusters. These two parameters in keybin are analogous to the ϵ (radius) and *minPts* (minimum density in a cluster) parameters in the DBSCAN algorithm. In this sense, our algorithm combines traits of density based and hierarchical clustering algorithms. As there are no universal optimal values of $depth$ and *minD* for all datasets, keybin needs to iterate through a range of values to find out the best clustering pattern. Note that while we implemented keybin in mpy4py, using a master-worker topology, keybin is not constrained to this implementation or topology. Given the appropriate communication channels, keybin can easily work as a wide-area algorithm for remote and geographically distributed sites.

We use a synthetic data generator to create random high dimensional data with noisy features. By using synthetic data, we are able to create large and very high-dimensional datasets with varying levels of noise and separability. This control mechanism can help us avoid ceiling effects while testing keybin and other clustering algorithms under systematic conditions. Source code, settings, and datasets are available at *https://lobogit.unm.edu/datascience/keybin-cluster2017*

V. Evaluation

Although we leverage mpi4py to implement keybin, the algorithm does not depend on specific MPI functions and does not require tightly coupled communication. Our algorithm is general for broader clustering applications, where data are created and stored in a distributed manner. Instead of the term *node* which is more often used in MPI environments, we use the term *site* to refer to a more general concept of a geographically distributed site. Under the restriction of no raw data exchange between sites, we empirically evaluate *keybin* in four steps:

1) First we quantify the effect of collapsing noisy dimensions on the accuracy of keybin.
2) Second, we compare keybin's accuracy and elapsed time with two well-known clustering algorithms: K-means and DBSCAN.
3) Third, we include an extended study about the influence of number of distributed sites and data imbalance on keybin's accuracy.
4) Finally, we evaluate the scalability of keybin as the number of dimensions, points, and distributed sites grow.

We ran the experiments on the Xena cluster at the Center for Advanced Research Computing of the University of New Mexico. Xena is a PowerEdge R730 / Intel Xeon CPU E5-2640 at 2.6 GHZ with 32 nodes, 16 cores per node, Infinibad interconnect, and 4GB of RAM per core.

A. Collapsing noisy dimensions

The goal of these experiments is to provide a quantitative justification regarding *Step 3* in our algorithm. We want to determine whether it is possible to get a gain in accuracy with respect to the baseline by collapsing certain dimensions. In this experiment, we generated $1,000$ 10-dimensional data points in 2 sites. There are 5 global clusters. Among the 10 dimensions, 2 of them are noise. A noisy dimension contains mixed uniformly distributed noise and several *modes*. Modes appear at random in discrete positions. To quantify the clustering accuracy, we report precision, recall, and the f1-score. Precision is the ratio $tp/(tp + fp)$ where tp is the number of true positives (i.e., points assigned to a cluster C that actually belong to C) and fp the number of false positives (i.e., points assigned to a specific cluster C that do not belong to C). The precision is the ability of the clustering not to assign a point to a cluster C that does not belong to it. Recall is the ratio $tp/(tp+fn)$ where fn the number of false negatives (i.e., the number of points that belong to a cluster C but were not identified as part of C). The recall is the ability to find all the data points that belong to a cluster. The f-score is the harmonic mean of precision and recall.

After 30 trials, the collapsing of noisy dimensions increases precision by 24.2% (from 66.9% to 91.1%) but slightly decreases recall by 4.4% (from 95.2% to 90.8%). The overall f1-score accuracy is improved by 14.0% (from 0.751 to 0.891). The reason for the increase in precision and decrease in recall is because noisy dimensions are a mixture of uniformly distributed noise and random agglomerations. In this situation, bin densities are more likely to be clumped together on the noisy dimensions. Our empirical study shows that before collapsing, keybin tends to assign points to larger and more spread agglomerations, reducing the number of clusters found. After removing noise, keybin tends to find smaller and more compact clusters.

B. Comparing keybin with other clustering methods

Traditional distributed approaches assume that data points hold the i.i.d property, but in most real-world processes, this property is too optimistic. Our global clustering does not make this assumption and is shown to work well even when distributed sites are very skewed. In this experiment we compare keybin's performance compared to other well-known and widely-used clustering algorithms: K-means++ (i.e., and optimized version of the popular K-means) and DBSCAN, both from scikit-learn 0.17.1. Also, we compare with the state of the art HPC implementation of DBSCAN, PDSDBSCAN [27], and attempted a comparison with the GPU implementation of MAFIA (GPUMAFIA [39]).

Our first experiment determines the ability of the different algorithms to find the correct number of clusters when data is slightly unbalanced. For this scenario, we generated $80,000$ data points with 100 dimensions. These points are divided in four distributed sites. The global structure consists of five global clusters. Each site has $20,000$ points, but only 4 out of the 5 clusters are dense while 1 is very sparse. While 4 clusters make up for 24.975% each, 1 cluster comprises only 0.1% of the data. For this experiment we compare accuracy and running time of keybin, K-means, DBSCAN, and PDBSCAN. We were unable to include the comparing performance of GPUMAFIA as it stopped converging with as little as 40 dimensions.

We set the number of iterations to 6 for all the three algorithms to get the best accuracy measure. keybin iterates through $depth = 4, 6, 8$, and $minD \in \{0, 1\}$. K-means iterates through 6 values of k, including the optimal value 5. DBSCAN and PDSDBSCAN iterate through $\epsilon \in \{1, 10, 25\}$ and $minPts \in \{5, 10\}$. In total, 30 trials per algorithm. In Table I, we report average number of clusters found, recall, and precision across all sites.

Algorithm	Avg. Clusters	Precision	Recall	Time (s)
keybin	5.00	1.00	0.99	63.68 (63,64)
K-means	4.33	0.97	0.74	127.56 (109,145)
DBSCAN	5.00	1.00	0.75	341.19 (323,358)
PDSDBSCAN	32.69	N/A	N/A	156.12 (89,222)

TABLE I: Average accuracy and time with confidence intervals for 5 clusters in 4 sites with unbalanced data ($80,000$ data points, each with 100 dimensions)

Even with this modest dataset and the same number of iterations, keybin is 2 times faster than K-means and more than 5 times faster than DBSCAN. The execution time of PDSDBSCAN is widely variable, ranging from as little as 12 seconds to more than 700 seconds; it is worth noting though,

that a small number of nodes is not the optimal setup for PDSDBSCAN. In terms of accuracy, keybin is always able to find the 5 different clusters. K-means is tricked into finding less groups for some of the runs. DBSCAN can find the correct number of clusters but with lower accuracy. A possible reason for DBSCAN's lower recall is that large spread clusters may be split and merged into smaller ones, creating more false negative cases. Surprisingly, In 20 out of 30 iterations PDSDBSCAN found zero clusters. In the remaining 10 runs, it found an average of 32 clusters and a maximum of up to 240, thus making unfeasible a direct comparison of accuracy in the 100-dimensions scenario. Other tests show PDSDBSCAN performing at 0.24 to 0.31 precision and 0.54 to 0.63 recall for 10, 20, and 40 dimensions. Table I shows the number of clusters found, recall and precision accuracy. The last row compares the elapsed time.

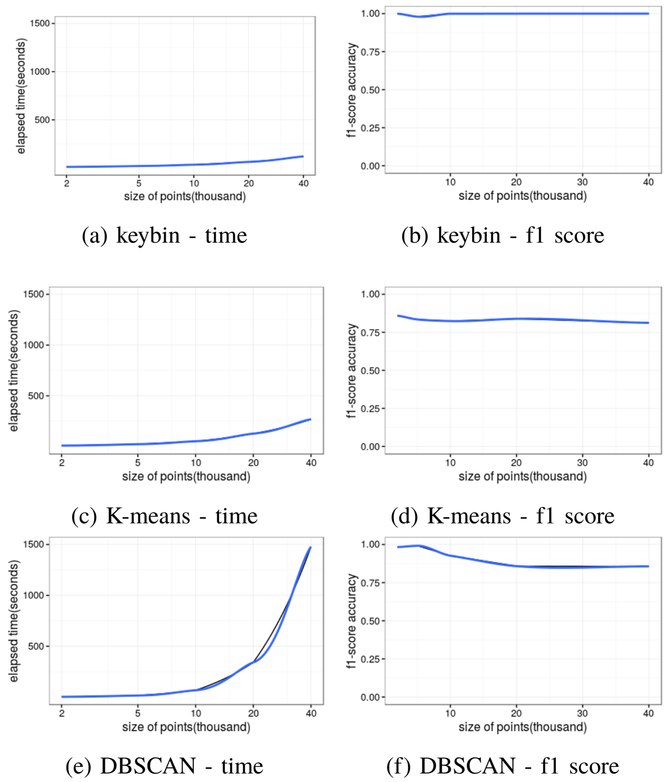
In order to understand the time growth and accuracy of the different algorithms as the number of points increases, we modified the experiment by varying the number of points from 2000, 5000, 10000, 20000, and 40000. Still, we use 100-dimensional data with 5 clusters and 4 sites. Since PDS-DBSCAN was unable to perform comparable to the other algorithms for the 100-dimensional test, we decided to exclude it from this experiment. Figure 5 shows the time and f1-score for the three algorithms. Precision and recall remained close to 1 for keybin, while K-means and DBSCAN kept them in the 0.7 and 0.9 range respectively. In this experiment, the difference is more evident for the time complexity among the three methods. On average, K-means remained 2 to 3 times slower than keybin. DBSCAN showed up to 10 times the slowdown as compared to keybin, and exhibited an exponential trend.

C. Evaluating keybin's on varying levels of unbalanced data

The goal of this experiment is to evaluate how keybin performs under a range of data imbalances on different number of sites. We fixed the number of clusters to 16, we set the number of dimensions to 10, and the number of points per site 64000 (1 million in total). Then, we factor the number of sites in powers of two as $\{2, 4, 8, 16\}$ and we defined three different types of data imbalance *uniform*, each cluster has a 6.25% share of the data; *one out*, there is one cluster missing from each site; *one large* there is one big cluster consisting of 90% points and 15 small clusters with only 0.667% of the data. Table II shows that accuracy becomes more stable when the number of sites increases. This is reasonable because the probability of making an error P_e is the product of such probability for each site $\prod_{i=1}^{N} p_e^i$.

D. Evaluating dimensionality growth and separability

We tested keybin's scalability as a function of dimensionality growth and accuracy as a function of separability. In the following experiments, we fixed the number of sites to 16, and the number of total points to 1.28 million. Each site contains $80,000$ data points. We varied the dimensionality of the dataset by $\{10, 100, 1000\}$. To test the ability of keybin to accurately find clusters at varying degrees of separability, we varied the number of clusters by $\{2, 4, 8, 16, 32, 64, 128\}$ within the same size hypercube. The rationale was to generate a space with sparse and separable clusters for the smaller values, and very dense and non-separable clusters for the higher values.



(a) keybin - time (b) keybin - f1 score

(c) K-means - time (d) K-means - f1 score

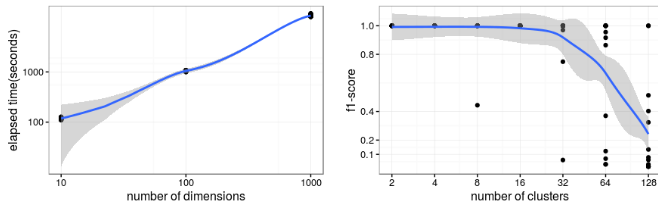
(e) DBSCAN - time (f) DBSCAN - f1 score

Fig. 5: Comparison of time and accuracy as the number of points increase

Experiment	Sites	Clusters found	Recall	Precision
uniform	2	15.67	1.0	0.997
	4	16	1.0	1.0
	8	16	1.0	1.0
	16	16	1.0	1.0
one out	2	15	0.999	0.877
	4	14	0.998	0.866
	8	11.67	0.999	0.736
	16	15	1.0	1.0
one large	2	15.33	0.999	0.997
	4	16	1.0	1.0
	8	16	1.0	1.0
	16	16	1.0	1.0

TABLE II: Average accuracy through growing site number over different patterns of data imbalance

Figure 6a shows a linear trend for keybin's time complexity. When the dimensionality increases, the elapsed time increases linearly. The number of clusters does not affect keybin's elapsed time. As expected, Figure 6b shows that the accuracy of keybin depends on data separability. The dimensionality does not influence accuracy but the number of clusters affects the data separability, as the size of the hypercube does not change with the number of clusters and they all have to share



(a) scalability as a function of dimensionality growth (b) accuracy as a function of separability

Fig. 6: Study of running time and accuracy as dimensions grow and as space becomes non-separable

the same "volume." In this case, keybin tends to assign non-separable data to a big cluster, so recall is high but precision decreases. At 32, there are a few groups that are not separable, and at level 64 or higher, keybin cannot distinguish the highly mixed groups.

VI. Limitations

Orthogonality Assumption. A basic idea behind our algorithm is the conceptual framework from the *a priori* algorithm. The idea is analogous to finding frequent patterns of $\{p, q\}$, where both p and q have to be frequent. Here we assign p and q to the final clustering label only if they belong to the same primary cluster on every dimension. This idea is the backbone of our map-reduce like approach in assigning the final cluster labels, and it needs the assumption that data dimensions are orthogonal. In non orthogonal situations, this does not hold.

Although we do not need pairwise distance computations, the distance between points affects how we put them into bins. We use the distance metric to illustrate how the orthogonal assumption influences our algorithm. The distance between points p and q is $d_x(p, q) = |x_2 - x_1|$ on dimension$_x$. If we include the orthogonal dimension$_y$ for consideration, the actual distance in the 2D space is $d(p, q) = \sqrt[2]{(x_2 - x_1)^2 + (y_2 - y_1)^2} \geq |x_2 - x_1|$. The distance $d(p, q)$ will only be further stretched. We can use the bin number of x_1, x_2 and y_1, y_2 to infer the actual positions of p and q in 2D space.

If we include the nonorthogonal dimension$'_y$ for consideration, the actual distance $d(x', y')$ in the 2D space may be shortened for some cases. That is to say, we cannot rely on the bin number of x'_1, x'_2 and y'_1, y'_2 to infer the actual position of p' and q' in the 2D space. Ongoing work is in identifying nonorthogonal dimensions and collapsing them, which should help address this issue.

Projection overlapping. The bin frequencies on each dimension can be seen as a partial view of the data set projected into particular dimensions. They are *shadows* of the data set. Where the *shadows* of two clusters overlap, the above algorithm cannot separate them. If p_i is the probability of two clusters overlapping on the ith dimension, then the probability that our algorithm cannot separate the two clusters is the product of such overlapping probabilities on all dimensions. As the dimensionality increases, $\prod_{i=1}^{N} p_i$ quickly becomes diminishes. Thus, our algorithm is likely to perform better as the number of dimensions increase.

VII. Conclusion

In this paper, we present a key-based binning clustering algorithm that is able to discover a global clustering structure in scenarios where there is a limited global view or direct access to the data. Our algorithm welcomes large dimensional datasets because it has been implemented without pair-wise point comparisons and can be optimized in a embarrassingly parallel manner. The communication that is needed to learn clustering patterns from a distributed collection is extremely small when compared to the entire original, raw dataset. As our technique leverages the benefits of parallel operations and feeding only need-to-know information from one point to another, individual data points cannot be reproduced, and so an inherent aspect of privacy is maintained.

Our algorithm takes advantage of a filter that collapses uninformative dimensions; this technique leads to a high accuracy in finding global clustering structures, resulting in a performance which is comparable to DBSCAN and one that outperforms k-means, given that these two algorithms are presented the pooled data.

We also discussed the limitations of our algorithm. In particular, its assumption for dimension orthogonality and the probability that it cannot separate groups if their bin frequencies overlap along all dimensions. As the dimensionality grows higher, the probability of poor accuracy due to overlapping drops quickly. Applying domain-specific knowledge can benefit our algorithm by providing guidelines for collapsing, particularly in the context of noisy or non-orthogonal dimensions.

Ongoing work revolves around the idea of gathering and exchanging bin frequencies between distributed sites. In this paper we present an mpi4py version of keybin, but this does not mean we rely on the parallelization power of MPI functions. A more ideal approach will be in using GPU threads to boost performance because the nature of our algorithm does not require pairwise distance computations and again, can be implemented through embarrassing parallelism.

Acknowledgment

This material is based upon work supported by the National Science Foundation for the grant entitled *CAREER: Enabling Distributed and In-Situ Analysis for Multidimensional Structured Data* (NSF ACI-1453430). We thank the UNM Center for Advanced Research Computing for computational resources used in this work.

References

[1] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, “Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, pp. 78:1–78:12.

[2] D. Tiwari, S. S. Vazhkudai, Y. Kim, X. Ma, S. Boboila, and P. J. Desnoyers, “Reducing data movement costs using energy-efficient, active computation on ssd,” in *2012 Workshop on Power-Aware Computing and Systems*. USENIX, 2012.

[3] J. C. Bennett, H. Abbasi, P. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, pp. 49:1–49:9.

[4] M. Dreher and B. Raffin, "A flexible framework for asynchronous in situ and in transit analytics for scientific simulations," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2014, pp. 277–286.

[5] J. Šíma and P. Orponen, "General-purpose computation with neural networks: A survey of complexity theoretic results," *Neural Computing*, vol. 15, no. 12, pp. 2727–2778, 2003.

[6] R. Salakhutdinov and G. Hinton, "Semantic hashing," *Int. J. Approx. Reasoning*, vol. 50, no. 7, pp. 969–978, 2009.

[7] J. Zhou, L. Chen, C. L. P. Chen, Y. Wang, and H. X. Li, "Uncertain data clustering in distributed peer-to-peer networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1–15, 2017.

[8] H. Kargupta, W. Huang, K. Sivakumar, and E. Johnson, "Distributed clustering using collective principal component analysis," *Knowledge and Information Systems*, vol. 3, no. 4, pp. 422–448, 2001.

[9] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, no. 1, pp. 1–22, 2011.

[10] T.-Y. Liu, W. Chen, and T. Wang, "Distributed machine learning: Foundations, trends, and practices," in *Proceedings of the 26th International Conference on World Wide Web Companion*, ser. WWW '17 Companion, 2017, pp. 913–915.

[11] S. Bandyopadhyay, C. Giannella, U. Maulik, H. Kargupta, K. Liu, and S. Datta, "Clustering distributed data streams in peer-to-peer environments," *Information Sciences*, vol. 176, no. 14, 2006.

[12] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, 2016, pp. 265–283.

[13] M. Abadi, "Tensorflow: Learning functions at scale," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2016, 2016.

[14] T. Estrada and M. Taufer, "On the effectiveness of application-aware self-management for scientific discovery in volunteer computing systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, pp. 80:1–80:11.

[15] H. Kawashima, R. R. Sato, and H. Kitagawa, "Models and issues on probabilistic data streams with Bayesian Networks," in *Proc. of the International Symposium on Applications and the Internet (SAINT)*, 2008.

[16] Y. Liu, L. C. Jiao, F. Shang, F. Yin, and F. Liu, "An efficient matrix bi-factorization alternative optimization method for low-rank matrix recovery and completion," *Neural Netw.*, vol. 48, 2013.

[17] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1999, pp. 518–529.

[18] C. C. Aggarwal, J. L. Wolf, P. S. Yu, C. Procopiuc, and J. S. Park, "Fast algorithms for projected clustering," *SIGMOD Rec.*, vol. 28, no. 2, pp. 61–72, 1999.

[19] A. Quiroz, M. Parashar, N. Gnanasambandam, and N. Sharma, "Design and evaluation of decentralized online clustering," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 3, pp. 34:1–34:31, 2012.

[20] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. ACM, 1998, pp. 604–613.

[21] P. Zhang, B. J. Gao, X. Zhu, and L. Guo, "Enabling fast lazy learning for data streams," *IEEE International Conference on Data Mining*, vol. 0, pp. 932–941, 2011.

[22] M. Barrena, E. Jurado, P. Márquez-Neila, and C. Pachón, "A flexible framework to ease nearest neighbor search in multidimensional data spaces," *Data Knowl. Eng.*, vol. 69, no. 1, pp. 116–136, 2010.

[23] D. Omercevic, O. Drbohlav, and A. Leonardis, "High-dimensional feature matching: Employing the concept of meaningful nearest neighbors," in *IEEE 11th International Conference on Computer Vision*, 2007, pp. 1–8.

[24] S. Boyd, "Convex optimization: From embedded real-time to large-scale distributed," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '11, 2011.

[25] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, and Others, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *Kdd*, vol. 96, no. 34, 1996, pp. 226–231.

[26] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, P. Dubey *et al.*, "Bd-cats: big data clustering at trillion particle scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 6.

[27] M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary, "A new scalable parallel dbscan algorithm using the disjoint-set data structure," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 62.

[28] Q. Zhang, L. T. Yang, Z. Chen, and P. Li, "Pphopcm: Privacy-preserving high-order possibilistic c-means algorithm for big data clustering with cloud computing," *IEEE Transactions on Big Data*, vol. PP, no. 99, 2017.

[29] Z. Gheid and Y. Challal, "Efficient and privacy-preserving k-means clustering for big data mining," in *2016 IEEE Trustcom/BigDataSE/ISPA*, Aug 2016, pp. 791–798.

[30] R. R. Gupta, G. Mishra, S. Katara, A. Agarwal, M. K. Sarkar, R. Das, and S. Kumar, "Data storage security in cloud computing using container clustering," in *2016 IEEE 7th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, Oct 2016, pp. 1–7.

[31] H. Kargupta, S. Datta, Q. Wang, and K. Sivakumar, "Random-data perturbation techniques and privacy-preserving data mining," *Knowledge and Information Systems*, vol. 7, no. 4, pp. 387–414, 2005.

[32] P. Jin and Q. Song, "A novel index structure r*q-tree based on lazy splitting and clustering," in *IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, 2011, pp. 405–407.

[33] M. Bendechache, N. A. Le-Khac, and M. T. Kechadi, "Hierarchical aggregation approach for distributed clustering of spatial datasets," in *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, Dec 2016, pp. 1098–1103.

[34] M. Rahmani and G. K. Atia, "In pursuit of novelty: A decentralized approach to subspace clustering," in *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Sept 2016, pp. 447–451.

[35] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic subspace clustering of high dimensional data for data mining applications," *ACM SIGMOD Record*, vol. 27, no. 2, pp. 94–105, 1998.

[36] S. Goil, H. Nagesh, and A. Choudhary, "MAFIA: Efficient and scalable subspace clustering for very large data sets," … *Discovery and Data Mining*, vol. 5, pp. 443–452, 1999. [Online]. Available: http://mrl.cecsresearch.org/Resources/papers/goil99mafia.pdf

[37] R. Agrawal, R. Srikant, and Others, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[38] H. W. Lilliefors, "On the kolmogorov-smirnov test for normality with mean and variance unknown," *Journal of the American statistical Association*, vol. 62, no. 318, pp. 399–402, 1967.

[39] A. Adinetz, J. Kraus, J. Meinke, and D. Pleiter, *GPUMAFIA: Efficient Subspace Clustering with MAFIA on GPUs*. Springer Berlin Heidelberg, 2013, pp. 838–849.