Scalable Clustering Over High Dimensional Vector Streams

HAN HAN, University of Arizona, USA
BEICHUAN ZHANG, University of Arizona, USA
LEI CAO, University of Arizona, USA

As a data preprocessing step, clustering high-dimensional vector streams plays a critical role in many applications such as recommendation, community detection in social networks, and anomaly detection in large-scale computer networks. In this work, we propose a scalable framework, `Suffice`, to cluster high-dimensional vectors in streaming environments that continuously experience three fundamental types of data update including entry values, vectors, and dimensions. `Suffice` features a new clustering definition that is native to the dynamic and high-dimensional nature of vector streams. Moreover, it incorporates two key optimizations. First, to minimize the frequency of pairwise similarity comparisons between vectors, we design an adaptive multiple reference strategy that dynamically selects multiple reference vectors per cluster. Each reference vector in a cluster is associated with a *safe region* such that any vector falling into it is guaranteed to belong to this cluster. We design a reward function to select reference vectors that jointly maximize the size of the safe region, thus minimizing the chance of conducting extra similarity comparisons when determining the cluster membership of incoming vectors. Second, we introduce a double-hash structure that efficiently supports both nearest neighbor-based and furthest neighbor-based similarity searches, critical to selecting and searching reference vectors. Using both real-world and synthetic streaming datasets, our comprehensive experimental studies confirm that `Suffice` outperforms baselines with a speedup from 18× to 120× in run time and demonstrate the superiority of our proposed cluster definition in cluster quality.

CCS Concepts: • **Information systems** → **Data cleaning**.

Additional Key Words and Phrases: High Dimensional Vector, Stream, Clustering

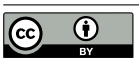
ACM Reference Format:
Han Han, Beichuan Zhang, and Lei Cao. 2026. Scalable Clustering Over High Dimensional Vector Streams. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 80 (February 2026), 26 pages. https://doi.org/10.1145/3786694

1 Introduction

Applications in different domains, such as e-commerce, social media, and computer networks, are generating high-volume data streams. In e-commerce [18], users generate streams of behavioral data through browsing, clicking, and purchasing activities, reflecting their evolving interests and preferences on various items. In social media [40], user interactions with friends or posts, such as comment, like, and share, form dynamic relationships and latent community structure. In computer networks [21], routing nodes frequently update the path information, reflecting the connectivity and routing dynamics of the nodes. The data instances in these *behavior-driven* streaming applications share common characteristics: they are *high-dimensional* and *highly dynamic*.

Clustering, which organizes these behavior-driven data streams into user or node groups based on their evolving behavior patterns, serves as a critical preprocessing step for downstream tasks such as recommendation systems [20], community discovery [33], and anomaly detection [10].

Authors' Contact Information: Han Han, University of Arizona, USA, hanhan1@arizona.edu; Beichuan Zhang, University of Arizona, USA, bzhang@cs.arizona.edu; Lei Cao, University of Arizona, USA, lcao@csail.mit.edu.



ACM 2836-6573/2026/2-ART80
https://doi.org/10.1145/3786694

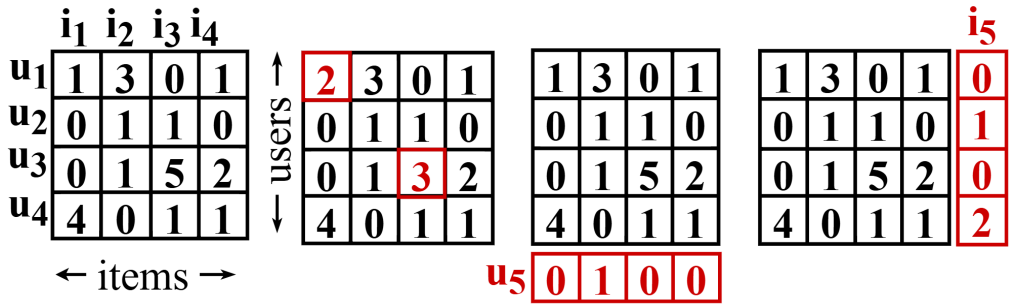


Fig. 1. Three Types of Date Update.

Next, we use real applications to discuss the characteristics of these streams and the importance of clustering.

In *recommendation systems*, user behavior is typically represented as a user–item matrix [35]. Each row is a *high-dimensional* vector corresponding to a user, while each column corresponds to an item. The value in each entry captures a measure of user-item interaction, such as purchase count or user rating score. As shown in Fig. 1, these user vectors continuously evolve over time, exhibiting *three major types* of dynamic changes. (a) *Value update*: the values of the existing user vectors change overtime as a user may purchase certain items more or less frequently; (b) *Vector update*: new users arrive, introducing new vectors, while some users become inactive, leading to the expiration of the corresponding vectors. (c) *Dimension update*: the platform adds or removes items, resulting in increases or decreases of the dimensionality of all user vectors. Combined with the fact that the number of users and items can be very large, this leads to a data stream that is *high-volume*, *high-dimensional*, and *highly dynamic*. Clustering these high-dimensional streaming user vectors into behaviorally similar groups can be used as a pre-filtering layer in a recommendation system [9]. Instead of comparing a target user against the entire item catalog or user base, the system can restrict candidate items to those that are highly rated within the same cluster. This improves the efficiency and effectiveness of the recommendations.

In *social networks*, behavior between users and posts can be encoded as a user–post interaction matrix [27], where each row represents a user as a high-dimensional vector, each column, i.e., a dimension, is a post, and each entry reflects the frequency of interaction between a user and a post. In *computer networks*, the Border Gateway Protocol (BGP) records how Autonomous System (AS) nodes, i.e., the routing nodes, continuously update the valid paths to reach a specific destination [30]. Each AS node can be encoded as a *high-dimensional* vector, where each dimension corresponds to an observed path. The entries in the vector may reflect statistics such as the frequency of each path used by the AS node. As users (AS nodes) join and leave the network and posts (paths) continuously appear and become stale, these high-dimensional vectors show *three types* of update, i.e., *value updates*, *vector updates*, and *dimension updates*, similar to recommendation systems.

In these applications, clustering serves as a pre-processing mechanism for downstream tasks such as community detection [38] and anomaly detection [4]. For example, a social network could first cluster users into coarse-grained groups and then detect fine-grained communities within or across these clusters using machine learning models such as graph neural networks [15]. In a computer network, a BGP hijack anomaly triggers a sudden surge of new paths, increasing the dimensionality of AS vectors [4]. By clustering these AS vectors and monitoring temporal group

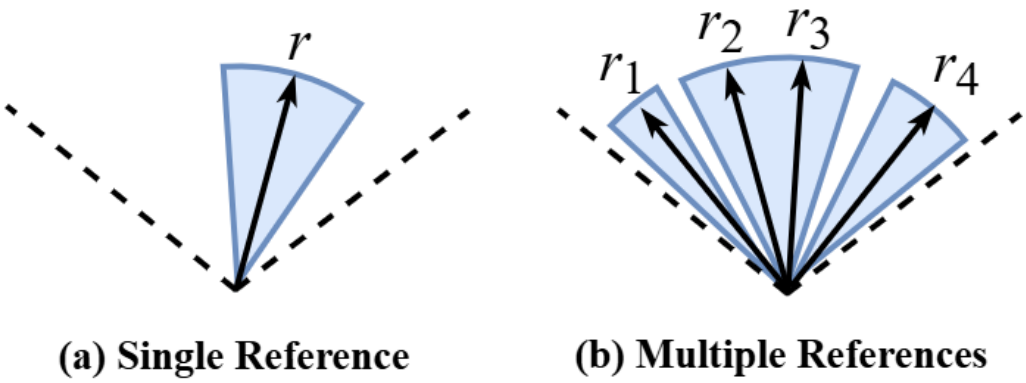


Fig. 2. Single Reference v.s. Multiple References

features (e.g. node migration ratio, density, and size) over time, the network could capture sudden feature changes caused by BGP hijack as anomalies due to group splits or emerging groups [36].

In summary, all the above applications can represent their data as high-dimensional vector streams that continuously update in three major ways: *value update*, *vector update*, and *dimension update*. Table 1 in Sec 5.1 presents the numbers of these types of update in the real datasets w.r.t. these applications.

The Limitations of State-of-the-Art. Researchers have begun investigating the problem of clustering high-dimensional vectors in streaming environments. However, most existing stream clustering methods assume fixed feature spaces and focus primarily on handling point insertions or deletions [16]. Dynamic feature spaces, where new dimensions emerge or expire over time, are rarely considered, despite being common in behavioral vector streams. Furthermore, optimization techniques in the literature often rely on dimensionality reduction [12, 24] or sparse coding [13] to map high-dimensional data to low-dimensional representation. These techniques are computationally expensive, thus ineffective in high-volume, frequent update settings.

Focused on low-dimensional data, other streaming clustering methods [2, 17] use the classical outlier definitions, such as distance- or density-based outliers. These works do not nicely fit high-dimensional vector streams, because they use Euclidean distance for clustering, which becomes ineffective for high-dimensional *sparse* spaces due to the curse of dimensionality [3], where distance or density metrics lose discriminative power and density estimation becomes unreliable [39].

Proposed `Suffice` Approach. To address these limitations of existing methods, we propose `Suffice`, a scalable clustering framework that efficiently handles high-dimensional vector streams with frequent update in *values*, *vectors*, and *dimensions*.

First, `Suffice` features a new clustering definition, *native* to high-dimensional vector streams. It ensures that every pair of vectors in a cluster has a small angle. Because similarity measured by angle is not subject to the curse of dimensionality, it is robust to high-dimensional vectors. Moreover, it is amenable to the dynamic nature of streaming data, as *expiring vectors*, one of the most expensive operations in stream clustering [41], can be handled in *constant time*. Therefore, we model the operations triggered by all three types of update in vector streams as an *expiration operation* and an *insert operation*, offering us opportunities to design a unified framework to continuously and efficiently update the cluster structure.

Our key technical innovations are in optimizing the performance of continuously inserting vectors into clusters, which constitutes the performance bottleneck of `Suffice`. The key insight is to select a *reference vector* w.r.t. each cluster. As long as a new vector falls into an angular region

surrounding a reference, namely *safe region*, depicted as a blue cone around a reference r in Fig. 2. (a), this vector is guaranteed to belong to the associated cluster. This avoids exhaustively conducting pairwise comparison between the new vector and each existing vector.

However, maintaining a single reference within each cluster tends to lead to a safe region with limited coverage. Therefore, we propose the optimized `Suffice` algorithm, which, by selecting multiple references, jointly maximizes the total coverage of safe regions, as depicted in Fig. 2 (b).

Moreover, we design a `Double Hash` data structure to speed up the search for a *nearest reference* whose safe region most likely covers an incoming vector, as well as the search for the *max angle* of each cluster, i.e., the largest angle between any pair of vectors in a cluster, which determines the coverage of a safe region. Therefore, `Double Hash` efficiently supports searching for both the nearest and farthest neighbors, which seem to have contradictory requirements. The key idea is to compactly model the proximity distribution of the vectors in the angular space based on their angles to some references. Furthermore, using two hash tables that complement each other, it overcomes the problem that two vectors having similar angles to a reference are not necessarily close to each other.

Contributions. The key contributions of this work include:

- We propose a new clustering definition, native to the high-dimensional and dynamic nature of vector streams.
- We propose a scalable clustering framework that supports all three major types of update in high-dimensional vector streams.
- We propose a multi-reference strategy to minimize the frequency of pairwise comparison among vectors during clustering.
- Our `Double Hash` effectively speeds up the selection, maintenance, and search of reference vectors.
- We have conducted extensive experiments on both synthetic and real datasets, including Reddit, RetailRocket, and BGP. The results confirm that `Suffice` continuously produces high quality clusters and is 18 to 120× faster than the baselines.

2 The Clustering Problem Over High-dimensional Vector Streams

We first discuss the key concepts of high-dimensional vector streams in Sec. 2.1. In Sec. 2.2, we present our cluster definition. In Sec. 2.3, we overview our `Suffice` approach that dynamically maintains the cluster structure over time.

2.1 High-dimensional Vector Streams

The problem is to cluster high-dimensional vectors in a *sliding window*-based stream. We start with defining high-dimensional streaming vectors and the sliding window.

High-dimensional Streaming Vectors: At time t, the set of vectors is represented as $\mathcal{V}_t = \{v_1^{(t)}, v_2^{(t)}, \ldots, v_{n_t}^{(t)}\}$, where each $v_i^{(t)} \in \mathbb{R}^{d_t}$ is a d_t-dimensional vector. Each entry $v_i^{(t)}[d]$ represents the value of dimension d at time t.

Sliding Window: A time-based sliding window $[t - w, t]$ specifies the vectors whose timestamps fall within this interval, where w is the size of the window and t is the current time.

`Suffice` uses time-based sliding window to update $v_i^{(t)}[d]$, which represents the frequency of items d that user i purchases within a window. Both the number of vectors n_t and the dimensionality d_t change over time as new users or items arrive.

Various Types of Update. The streaming nature of the data introduces the following dynamics:

Value Update: Between two consecutive time windows t and $t + 1$, a vector $v_i^{(t)}$ may update at any specific dimension d. More specifically, at time $t + 1$, $v_i^{(t)}$ updates to a new vector $v_i^{(t+1)}$:

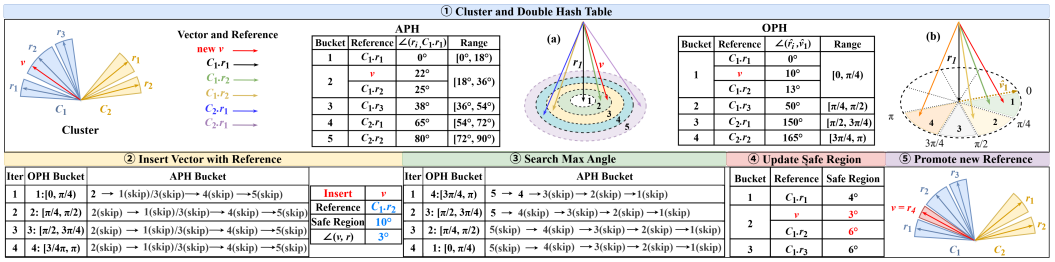


Fig. 3. Overview of Suffice.

$$v_i^{(t+1)}[d] = \begin{cases} v_i^{(t)}[d] + \Delta_{i,d}^{(t)} & \text{if dimension } d \text{ changes} \\ v_i^{(t)}[d] & \text{otherwise.} \end{cases}$$

where $\Delta_{i,d}^{(t)}$ denotes the update of the value w.r.t. the dimension d of vector i. As illustrated in Figure 1(a), vector u_1 increases its value at dimension i_1 from 1 to 2 due to its new interaction with item 1, while u_3 decreases its values at dimension i_3 from 5 to 3 due to its expired interaction with item 3.

Vector Update: A sliding window continuously receives new vectors and expires old vectors. A new vector $v_i^{(t)}$ is inserted into the window when at least one entry $v_i^{(t)}[d]$ becomes a nonzero value. In contrast, a vector $v_i^{(t)}$ expires when $v_i^{(t)}[d] = 0 \; \forall \; d$. As shown in Figure 1(b), a new vector u_5 arrives, which represents a new user and has a non-zero value at dimension i_2.

Dimension Update: The number of dimensions d_t in the sliding window could increase or decrease. A sliding window gets a new dimension when at least one vector $v_i^{(t)}$ gets a non-zero value $v_i^{(t)}[d]$ at dimension d. A dimension expires when its values in all vectors become zero. Figure 1(c) shows that a new item corresponding to dimension i_5 arrives, at which u_2 and u_4 have non-zero values.

Consider the high-dimensional vector stream at time t as a matrix $\mathcal{V}_t$. In the future, at any time step t_i, Suffice will continuously introduce new vectors or dimensions into $\mathcal{V}_t$, corresponding to new rows or columns that expand the current matrix. Suffice removes the expired vectors and dimensions from the matrix.

2.2 Clustering High-dimensional Vector Streams

Next, we define the problem of clustering high-dimensional vector streams. Given a set of high-dimensional vectors $\mathcal{V}_t = \{v_1^{(t)}, \ldots, v_{n_t}^{(t)}\} \subset \mathbb{R}^{d_t}$, where both the number of vectors n_t and the dimensionality d_t vary over time, our goal is to maintain a cluster structure $\mathcal{C}_t = \{C_1^{(t)}, \ldots, C_{k_t}^{(t)}\}$ at each time step t, where each cluster $C_j^{(t)}$ is a subset of vectors defined as follows.

Definition 2.1. Given a set of vectors $\mathcal{V}_t = \{v_1^{(t)}, \ldots, v_{n_t}^{(t)}\} \subset \mathbb{R}^{d_t}$ observed at time t, a cluster $C \subseteq \mathcal{V}_t$ is valid if it satisfies the following constraint:

$$\forall v_a, v_b \in C, \quad \angle(v_a, v_b) \leq \theta,$$

$\angle(v_a, v_b)$ denotes the angle between two vectors, and θ is a threshold w.r.t. the maximum angle between any pair of vectors in C.

The Merits of Our Cluster Definition. By Def. 2.1, a cluster remains valid as long as the pairwise similarity of any two vectors is larger than the threshold. Therefore, the removal of any individual vector does not affect the validity of a cluster. This property makes vector expiration a constant-time

operation, while it is broadly recognized that handling expiration is typically the most expensive operation in clustering streams [41].

2.3 The Suffice Framework

We present Suffice that continuously clusters a high-dimensional vector stream, guaranteeing that the clusters satisfy our cluster definition at any time. In each window, as *vector*, *value*, or *dimension* updates, Suffice will update the cluster structure. For vector update, new vectors continuously arrive, and existing vectors may expire. The new arrivals will be inserted incrementally into cluster. The expired vectors are simply removed because deleting a vector from a cluster does not affect its validity. Value update changes the values at specific dimensions of existing vectors. When such changes occur, Suffice first removes affected vectors from their current clusters and reinserts them into the appropriate clusters. Dimension update, which introduces new dimensions or expires existing ones, influences a set of vectors whose values lie in the affected dimensions. Similar to value update, these vectors are removed and then re-inserted due to the change of the feature space.

Because removal of vectors does not trigger any additional similarity comparison, the key is thus to optimize the efficiency of continuously inserting vectors into the cluster structure. Given a new vector, the *insert operation* either inserts it into an existing cluster if it does not violate our cluster definition or creates a new cluster. Next, we use Fig. 3 to illustrate the process that Suffice inserts a newly arrived vector v, highlighting the innovative ideas including the adaptive multi-reference strategy and the double hash data structure. Sec. 4 presents the technical details.

(1) Clusters and Double Hash. Suffice stores all existing vectors and references in our Double Hash structure, which consists of an Angular Proximity Hash table (APH) and a Orthogonal Projection Hash table (OPH). APH maps each vector to a bucket according to its angle with the base reference $C_1.r_1$. OPH orthogonally projects vectors and references onto the base plane of $C_1.r_1$ and then maps each vector to a bucket based on its angle to the projection $\hat{v}_1$ of a vector v_1. As shown in Fig. 3 (Step 1), the current window contains 2 clusters: C_1 with 3 references and C_2 with 2 references. The 5 buckets in APH correspond to the rings in Fig. 3(a), each representing an angle interval between a vector/reference and the base reference $C_1.r_1$. The 4 buckets in OPH correspond to the sectors in Fig. 3(b), where vectors and references are organized by their angles to $\hat{v}_1$ in the projected space.

(2) Insert Vector with Multi-Reference. Given an incoming vector v, Suffice first finds the cluster that is most likely to host it. Suffice achieves this efficiently by finding the nearest reference to v with Double Hash. As illustrated in Fig. 3 (Step 2), v is in Bucket 1 of OPH. The search starts from the nearest bucket (Bucket 1) and iteratively processes other buckets (Order: 1 $\rightarrow$ 2 $\rightarrow$ 3 $\rightarrow$ 4), because the references in Bucket 1 tend to have smaller angles with v than those in other buckets. When processing the vectors in Bucket 1, Suffice prioritizes the order based on the buckets they belong to in APH. Since v is in Bucket 2 of APH, Suffice starts with the references in the nearest bucket (Bucket 2) and proceeds to search further buckets (Order: 2 $\rightarrow$ 1/3 $\rightarrow$ 4 $\rightarrow$ 5) if necessary. This maximizes the chance of finding the small angles earlier. Suffice estimates a lower bound of the angle between v and any reference in the bucket it is searching. If this bound exceeds the current min angle, Suffice skips all remaining references to save pairwise comparisons. In this case, $C_1.r_2$ is the nearest to v. Since v falls into its safe region, Suffice inserts v into C_1.

(3) Search Max Angle. Next, Suffice computes the maximum angle between v and all vectors within this cluster. This *max angle* determines the boundary of the safe region if v is promoted to be a reference. Similar to the nearest neighbor search described above, Suffice uses the Double Hash to speed up this *farthest-neighbor* search operation, but searches buckets in a reverse order. The goal is to find the large angles as early as possible and skip the buckets if the angles between their vectors and v have no chance to exceed the current max angle. As illustrated in Fig. 3 (Step 3),

the search follows an order from the farthest bucket (Bucket 4) in OPH to the nearest (Bucket 1). When searching the vectors in an OPH bucket, `Suffice` prioritizes them based on their buckets in APH, i.e., from the outer to the inner rings ($5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$).

(4) Update Safe Regions. `Suffice` uses the computed max angle to create the safe region of v and updates that of each existing reference by checking if its angle with v is larger than its current max angle. In Fig. 3 (Step 4), $C_1.r_2$ updates its safe region.

(5) Promote New Reference. `Suffice` determines whether to promote a new vector as a reference or not based on how largely it could expand the safe regions of existing references. As shown in Fig. 3 (Step 5), `Suffice` elects v as a new reference r_4, as its safe region does not overlap with that of any current reference, making the safe regions of cluster C_1 clearly larger.

Next, in Sec. 3 we introduce our basic idea that uses single reference to speed up clustering and reveal its limitation; we then present in detail our optimized multi-reference strategy in Sec. 4.

3 Single Reference Optimization

Given our cluster definition in Def. 2.1, where all vectors in a cluster must maintain pairwise cosine similarity within a threshold θ, a Naive algorithm is to compare a new vector v with all existing vectors in all clusters and insert it into a cluster C_j if the constraint still holds. When inserting the i-th vector, Naive has to compare it with $(i-1)$ existing vectors, and each comparison requires $O(d)$ time, where d is the vector dimensionality. Otherwise, it creates a new cluster $C_{\text{new}} = \{v\}$. The total time complexity of inserting n vectors into clusters is thus $O((1+2+\cdots+(n-1)) \cdot d)$ $= O(n^2 \cdot d)$.

To reduce complexity, we present an optimization that uses *reference vectors* to reduce the frequency of similarity comparisons. Next, we first introduce the basic idea of this optimization and then discuss how to identify the *optimal* reference vectors.

3.1 The Reference Vector-based Algorithm

To mitigate the performance issue of the Naive method, we propose to maintain one *reference* vector within each cluster, namely Fix. Given any cluster C_j, when a new vector v arrives, it first compares v with the reference vector r_j of C_j. As long as v falls into the **safe region** associated with r_j, we can safely add v into C_j, without having to compare against any other vector in C_j.

Next, we introduce the concept of *safe region*. We start by defining the *boundary angle* θ_{r_j} w.r.t. a reference r_j.

Definition 3.1. Boundary Angle.

$$\theta_{r_j} = \theta - \max_{v_i \in C_j} \angle(v_i, r_j)$$

In Def. 3.1, $\max_{v_i \in C_j} \angle(v_i, r_j)$ represents the maximal angle between r_j and any vector in C_j, which we denote by A_{max}. Boundary angle refers to the gap between the clustering threshold θ and A_{max}, which is used in the following Lemma 1.

Lemma 1. *Let r_j denote the reference of cluster C_j. If*

$$\angle(v, r_j) \leq \theta - \max_{v_i \in C_j} \angle(v_i, r_j) = \theta_{r_j}$$

then for any $v_i \in C_j$, $\angle(v, v_i) \leq \theta$

Lemma 1 shows that as long as the angle between a new vector v and the reference r_j is smaller than the *boundary angle*, inserting v into C_j does not violate the constraint of our cluster definition.

Next, we give a brief proof.

Proof. We prove Lemma 1 based on the angular triangle inequality: for any three vectors v_a, v_b, v_c, we have

$$\angle(v_a, v_b) \leq \angle(v_a, v_c) + \angle(v_b, v_c)$$

This ensures: $\angle(v, v_i) \leq \angle(v, r_j) + \angle(v_i, r_j) \leq \theta$. □

Now we are ready to define *safe region*.

Definition 3.2. Safe Region. The safe region w.r.t. reference r_j of a cluster C_j is the spherical cone centered at the reference r_j, where the aperture of this cone equals the boundary angle θ_{r_j}.

The cone defined in Def. 3.2 is called a safe region because any vector v in this region can be safely inserted into the cluster without violating the cluster definition. As shown in Fig. 2(a), the blue cone represents the safe region of a reference r.

The probability that a new vector v can be directly inserted into cluster C_j is proportional to the boundary angle θ_{r_j}, i.e., the size of the safe region. For example, assume the cluster definition specifies a threshold angle $\theta = 30°$. According to Lemma 1, if the max angle A_{max} between reference r_j and any vector in cluster C_j is 10°, the boundary angle is $\theta - A_{max} = 30° - 10° = 20°$. When a new vector v arrives, if $\angle(v, r_j) \leq 20°$, then v lies within the safe region and can be directly inserted into cluster C_j.

Time Complexity. Let n be the number of vectors in the existing clusters, k be the number of clusters, d be the dimensionality of the vectors. Then for each incoming vector v, it performs k comparisons with the references, while each computation costs $O(d)$. If none of the k references satisfies the acceptance threshold, it conducts n pairwise comparisons to find a cluster for v. The total number of comparisons per insert is $O\left(kd + (1-\alpha)nd\right)$, where α denotes the probability that v falls in the safe region of a reference. Assuming vectors do not have negative values and are randomly distributed in angular space, the maximal possible angle between a reference and a new vector is $\frac{\pi}{2}$. Consequently, α is proportional to the size of a reference's safe region, which can be approximated by $\alpha = \frac{2(\theta - \max_{v_i \in C_j} \angle(v_i, r_j))}{\pi}$. Clearly, if α is large, it is much faster than Naive, as $k \ll n$.

3.2 Selecting Optimal Reference

Given a cluster C_j, if we could find a reference r_j that has the largest boundary angle, its safe region would be the largest, indicating r_j has the highest chance to directly admit a new vector into C_j. Therefore, r_j is *optimal*. Selecting a reference vector that maximizes its safe region is equivalent to minimizing the max angular between the reference and any vector within the cluster.

$$r^* = \arg\min_{r_j \in C_j} \max_{v_i \in C_j} \angle(v_i, r_j)$$

This equals to finding the minimal enclosing cone (MEC) [28] on the unit hypersphere that encloses the set of vectors C_j, centered at some r^*. In a three-dimensional space, computing the spherical Voronoi diagram [25] achieves an exact solution with a time complexity $O(n \log n)$. A randomized incremental algorithm can approximate it in $O(n)$ time [7]. It maintains a set of support vectors that determine the apex and aperture of the cone. The support vectors correspond to those vectors lying on the boundary of the cone. At each step, it checks whether a newly inserted vector lies within the current cone. If so, the cone remains unchanged. Otherwise, it recursively computes the MEC of the current support set plus v_i, where v_i is forced to be on the boundary of the resulting cone.

Time Complexity. In a three-dimensional space, the maximum size of the support set is 3, and the cone computation can be completed in linear time. The algorithm in [7] naturally generalize

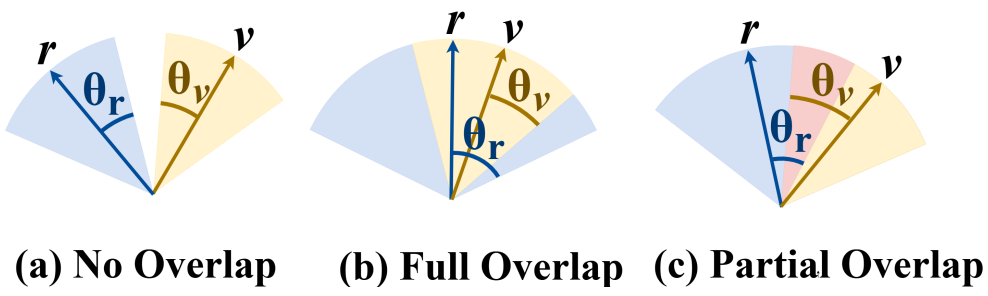


Fig. 4. Three Cases of Reference Candidates.

to arbitrary number of dimensions. However, the original work [7] does not provide complexity bounds in high-dimensional space. Next, we derive this bound by extending its analysis. In d-dimensional space, the maximum number of support vectors is d. The time complexity can be expressed recursively as:

$$T(n,d) = T(n-1,d) + T(n-1,d-1) + \cdots + T(n-1,0) = O(d! \cdot n)$$

where $T(n,d)$ denotes the expected time to compute the MEC of n vectors in $\mathbb{R}^d$ and $T(n-1,d-k)$ represents the recursive subproblem where the current vector is added to the support set containing k support vectors. In a high-dimensional space, this complexity is extremely high, potentially even higher than the naive approach that exhaustively computes the pairwise similarity.

4 Suffice: A Multi-reference Strategy

Next, we propose Suffice to overcome the problem that finding an optimal reference vector is too expensive. It is inspired by the observation that a single reference vector can only cover a safe region with limited size, even if we are able to find the optimal reference regardless of the cost. Therefore, we propose to maintain multiple references within each cluster. These references, each with its own adaptive *boundary angle*, jointly expand the coverage of the *safe regions* and increase the probability of accepting vectors.

As depicted in Fig.2(b), the safe regions of r_1, r_2, r_3 and r_4 act jointly as the *global* safe region of cluster C_1, significantly expanding the safe region covered by the single reference r.

Next, we present *adaptive multiple reference* (Sec. 4.1) and *double hash-based search* (Sec. 4.2), which make Suffice effective and efficient in electing and searching for references.

4.1 Adaptive Multiple Reference

As a cluster evolves, Suffice continuously evaluates whether a newly inserted vector $v \in C_j$ should be promoted to a reference. However, Suffice does not expire a reference for the following reason: even if a reference expires or is updated, it is still effective in admitting new vectors as long as we update its boundary angle and in turn the safe region. Our *reward-based* mechanism ensures that Suffice only elects references that clearly expand the global safe region of a cluster. This avoids maintaining redundant references.

Let $\mathcal{R}_j = \{r_j^{(1)}, r_j^{(2)}, \ldots, r_j^{(L)}\}$ denote the current reference vectors in cluster C_j. Whether a vector v could be a new reference or not is decided by *its reward*. To estimate the reward, we first compute the *boundary angle* θ_{new} (Def. 3.1) that v would have as a reference, as it represents the size of the safe region w.r.t. v. Next, we show how to use θ_{new} to estimate the reward.

As shown in Fig.4, we classify each reference candidate into three cases based on its overlap with existing references.

(1) No Overlap. The newly inserted vector v lies outside the angular coverage of all existing references in the cluster C_j. For each existing reference $r \in \mathcal{R}_j$, let θ_r denote its boundary angle.

$$\angle(v, r) \geq \theta_r + \theta_{\text{new}}, \quad \forall r \in \mathcal{R}_j \tag{1}$$

If *all* existing references r satisfy this condition, then the safe region that v covers as a reference is completely disjoint from those of existing references. `Suffice` will promote v to a new reference, because it largely expands the overall coverage of safe regions, leading to a large reward. In Fig. 4(a) the blue and yellow cones represent the safe regions of the existing reference r and the new vector v. By Eq.1, `Suffice` will elect v as a reference.

(2) Full Overlap. v lies entirely within the safe region of at least one existing reference. That is, suppose that each existing reference $r \in \mathcal{R}_j$ has a boundary angle θ_r, the following condition holds:

$$\angle(v, r) \leq \theta_r - \theta_{\text{new}} \tag{2}$$

This indicates that the entire safe region that v covers as a reference has been fully enclosed within the safe region of r. Because v does not provide any new coverage, it will not be promoted to a reference due to the low reward. Fig. 4(b) shows this case.

(3) Partial Overlap. The newly inserted vector v exhibits *partial overlap* with one or more existing references of cluster C_j in safe region, corresponding to the following condition:

$$|\theta_r - \theta_{\text{new}}| < \angle(v, r) < \theta_r + \theta_{\text{new}} \tag{3}$$

This indicates that the safe region of v partially overlaps with that of r, as shown in Fig. 4(c). The reward depends on the size of the overlapping region. Next, we propose a method to estimate it.

Angle-based Estimation. Suppose v has a boundary angle θ_{new}. As precisely computing the intersection of the safe regions of v and existing references is expensive [22], we approximate the degree of intersection by estimating their *angular overlap* with Eq. 4.

$$\theta_{\text{overlap}}^{(v,r)} = \theta_r + \theta_{\text{new}} - \angle(v, r) \tag{4}$$

In Equation 4, the angular overlap between v and r increases as the intersection increases. We then sum the estimated angular overlaps across all intersecting references.

$$\Theta_{\text{total}} = \sum_{\substack{r \in \mathcal{R}_j \\ |\theta_r - \theta_{\text{new}}| < \angle(v,r) < \theta_r + \theta_{\text{new}}}} \theta_{\text{overlap}}^{(v,r)}. \tag{5}$$

If this *total angle overlap* Θ_{total} exceeds v's own boundary angle θ_{new}, we consider v to be sufficiently covered, leading to a very small reward. `Suffice` thus does not promote it.

Data Distribution Drift. `Suffice` introduces references to accelerate the clustering process, by avoiding the expensive pairwise comparisons of Naive and high computation in searching optimal-reference. The selection of references thus matters to the efficiency of `Suffice`. However, no matter what reference `Suffice` chooses, all clusters strictly satisfy our cluster definition. `Suffice` effectively handles data distribution drift, because it continuously elects new references. We decide not to expire older references, because potentially they expand the size of the safe regions, thus enlarging the chance of directly admitting vectors to clusters. Our experiments in Sec 5.2 and 5.3 confirm the advantage of not expiring references.

4.2 Double Hash-based Search

`Suffice` has to search for a cluster most likely to host a new vector as well as the max angle. Both can be expensive.

Search Most Promising Cluster. When a new vector v arrives, `Suffice` searches for a cluster where one of its references has the smallest angle with v. A naive approach would be to compute the angle between v and every reference and find the closest reference. Even worse, when `Suffice` cannot determine whether v belongs to a cluster by only comparing v against the closest reference, it has to compare v with every vector in every cluster, which becomes extremely expensive in a large volume vector stream.

Search Max Angle. To decide whether a vector v should be a reference, `Suffice` maintains a boundary angle, which is determined by the *max angle* among all vectors within its cluster C_j. To compute it, a naive approach would be to compare v to every vector in C_j, resulting in $O(|C_j|)$ operations per update. This is expensive.

`Double Hash` reduces the number of pairwise comparisons in both scenarios. It consists of a data structure and two algorithms to search for the most promising cluster and the max angle.

4.2.1 *Double Hash Structure*

Similar to Locality Sensitive Hashing (LSH) [37], `Double Hash` maps the vectors close to each other to the same bucket. However, unlike LSH, which employs random projection and thus provides probabilistic guarantees, `Double Hash` is deterministic. This is because we map a high-dimensional vector into a one-dimensional space according to its angle w.r.t. references, therefore no need to conduct random projection. It consists of 2 hash tables, which together effectively encode the angle-based similarity of vectors.

Because the goal is to efficiently search vectors in the angular space, `Double Hash` hashes the angle between a vector and a reference, equivalent to hashing one-dimensional vectors. This produces the *angular proximity hash table* (APH). However, potentially vectors in the same bucket can still have large angles. For example, two vectors symmetrically offset an angle α_i from a reference r_1 may have an angle up to $2\alpha_i$, as shown in Figure 5. Thus, building APH alone is insufficient to capture the true proximity structure in angular space. To solve this problem, we introduce an auxiliary *orthogonal projection hash table* (OPH) that captures *how* vectors deviate from the direction of reference r_1.

Angular Proximity Hash (APH). This hash table captures the proximity distribution of all vectors w.r.t. the first reference vector $r_1 \in \mathcal{R}$. We define h_{aph} as a hash function that maps a vector v_i to a bucket based on its angle to the reference vector r_1. For any vector v_i in the cluster, we compute its angle to r_1: $\alpha_i = \angle(v_i, r_1)$.

Let $\mathcal{T}_{\text{aph}}$ be the hash table that maps bucket indices to sets of vectors with a hash function h_{aph}.

Definition 4.1. Angular Proximity Hash (APH)

$$h_{\text{aph}} : \mathbb{R}^d \to [\![0, \tau_1 - 1]\!], \quad h_{\text{aph}}(v_i) = \left\lfloor \frac{\alpha_i}{\pi/2\tau_1} \right\rfloor$$

where τ_1 is the size of $\mathcal{T}_{\text{aph}}$.

As illustrated in Fig. 5(a), when $\tau_1 = 5$, APH consists of 5 buckets, each corresponding to one angular ring. A vector v_i is mapped to the third bucket since its angle to the base reference r_1 ($\alpha_i = 40°$) falls within the interval $[36°, 54°)$.

Orthogonal Projection Hash (OPH). `OPH` hashes vectors based on the *orthogonal component* of each vector w.r.t. r_1. For any vector v_i, its orthogonal projection is:

$$\hat{v_i} = v_i - (\langle v_i, r_1 \rangle \cdot r_1),$$

We select a fixed base direction $\hat{v_1}$, typically the normalized orthogonal projection of the first inserted vector v_1:

$$\gamma_i = \angle(\hat{v_i}, \hat{v_1}).$$

Let $\mathcal{T}_{\text{oph}}$ be the hash table that maps bucket indices to sets of vectors with a hash function h_{oph}.

Definition 4.2. Orthogonal Projection Hash (OPH)

$$h_{\text{oph}} : \mathbb{R}^d \to [\![0, \tau_2 - 1]\!], \quad h_{\text{oph}}(v_i) = \left\lfloor \frac{\gamma_i}{\pi/\tau_2} \right\rfloor$$

where τ_2 is the size of $\mathcal{T}_{\text{oph}}$.

OPH represents how each vector deviates from the reference direction. In particular, even if two vectors in the same bucket of $\mathcal{T}_{\text{aph}}$ have the same angle to r_1, their orthogonal projections may point in different directions. Therefore, based on γ_i, we are able to separate vectors that are not directionally close within the bucket. As illustrated in Fig. 5(b), when $\tau_2 = 4$, OPH is divided into 4 buckets, each corresponding to one angular sector on the projection plane: $[0, \pi/4)$, $[\pi/4, \pi/2)$, $[\pi/2, 3\pi/4)$, and $[3\pi/4, \pi)$. Vector v_i is mapped to the 4th bucket since its projected angle γ_i falls within $[3\pi/4, \pi)$.

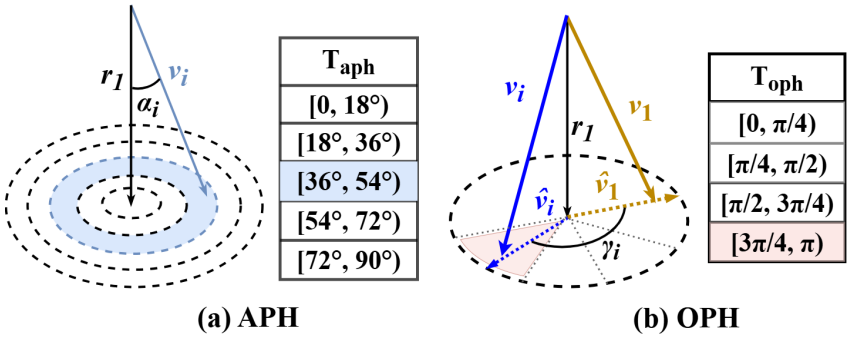


Fig. 5. The Construction Process of Double Hash.

4.2.2 *Search the Most Promising Cluster*

Given an incoming vector v, `Suffice` uses `Double Hash` to search its nearest reference. It starts with searching the bucket in OPH that is nearest to v to prioritize references whose orthogonal deviation directions are most close to that of v. Then it searches in APH, also starting from the bucket nearest to v, because the references in it are likely to have a small angle with v. For each bucket, we estimate the *lower bound* of the angle between v and the references in it, and skip it if this bound exceeds the smallest angle found so far.

The nearest-cluster search proceeds as follows:

(1) Compute the bucket indices of an incoming vector v in both hash tables(Lines 1 - 2, Alg. 1):

$$b_{\text{aph}}(v) = h_{\text{aph}}(v), \quad b_{\text{oph}}(v) = h_{\text{oph}}(v)$$

(2) Perform a nearest neighbor search in $\mathcal{T}_{\text{oph}}$. Starting from the bucket index $b_{\text{oph}}(v)$, `Suffice` searches neighboring $\mathcal{T}_{\text{oph}}$ buckets in increasing offset order, i.e., $b_{\text{oph}}(v) \pm 1, \pm 2, \ldots$. From each bucket, it retrieves candidate references $\mathcal{R}_{\text{oph}} \subseteq \mathcal{R}$. (Lines 5 - 6, Alg. 1).

(3) Perform a nearest neighbor search in $\mathcal{T}_{\text{aph}}$. Starting from the bucket index $b_{\text{aph}}(v)$, `Suffice` searches neighboring $\mathcal{T}_{\text{aph}}$ buckets containing $\mathcal{R}_{\text{oph}}$ in increasing offset order, i.e., $b_{\text{aph}}(v) \pm 1, \pm 2, \ldots$. b_{aph} denotes the bucket searched currently. (Lines 7 - 9, Alg. 1).

(4) Estimates the **lower bound** on the angle $\angle(v, r_k)$ between retrieved reference r_k in b_{aph} and v. By triangle inequality, angle $\angle(v, r_k)$ is large than the difference between angle $\angle(v, r_1)$ and angle $\angle(r_1, r_k)$. Given the searched bucket index b, angle $\angle(r_k, r_1)$ is in the range of $[b\frac{\theta}{\tau_1}, (b+1)\frac{\theta}{\tau_1}]$.

Algorithm 1: PromisingClusterSearch(v, cluster set C)

Input: Incoming vector v, current cluster set C, two hash tables $\mathcal{T}_{\text{aph}}$ and $\mathcal{T}_{\text{oph}}$
Output: Most promising cluster c^* and reference r^* for inserting v

1 $b_{\text{aph}}(v) \leftarrow h_{\text{aph}}(v)$, $b_{\text{oph}}(v) \leftarrow h_{\text{oph}}(v)$;
2 $\mathcal{R}_{\text{oph}} \leftarrow \emptyset$ $\mathcal{R}_{\text{aph}} \leftarrow \emptyset$;
3 $\Theta_{min} \leftarrow \infty$, $r^* \leftarrow$ None;
4 $offset1 \leftarrow 0$, $offset2 \leftarrow 0$;
5 **for** $b_{oph} \in \{ b_{oph}(v) - offset1,\ b_{oph}(v) + offset1 \}$ **do**
6 $\mathcal{R}_{\text{oph}} \leftarrow \mathcal{T}_{\text{oph}}[\, b_{\text{oph}} \,]$;
7 **for** $b_{aph} \in \{ b_{aph}(v) - offset2,\ b_{aph}(v) + offset2 \}$ **do**
8 $\mathcal{R}_{\text{aph}} \leftarrow \mathcal{T}_{\text{aph}}[\, b_{\text{aph}} \,] \cap \mathcal{R}_{\text{oph}}$;
9 $b \leftarrow b_{\text{aph}}$;
10 $\Theta_{\text{lower}} \leftarrow \min\left(0,\ \left|\angle(v, r_1) - b\frac{\theta}{\tau_1}\right|,\ \left|\angle(v, r_1) - (b+1)\frac{\theta}{\tau_1}\right|\right)$;
11 **if** $\widehat{\Theta}_{lower} \geq \Theta_{\text{min}}$ **then**
12 **continue**;
13 **foreach** $r_k \in \mathcal{R}_{aph}$ **do**
14 $\Theta_{r_k} \leftarrow \angle(v, r_k)$;
15 **if** $\Theta_{r_k} < \Theta_{\text{min}}$ **then**
16 $\Theta_{\text{min}} \leftarrow \Theta_{r_k}$;
17 $r^* \leftarrow r_k$;
18 $c^* \leftarrow C[\, r^*.cid \,]$;
19 **return** c^*, r^*

Therefore, the lower bound can be estimated by the following Equations(Lines 10 - 17, Alg. 1):

$$\Theta_{lower}(v, r_k) = \min\left(0,\ \left|\angle(v, r_1) - b\frac{\theta}{\tau_1}\right|,\ \left|\angle(v, r_1) - (b+1)\frac{\theta}{\tau_1}\right|\right)$$

If this lower bound $\Theta_{lower}(v, r_k)$ is smaller than the currently maintained min angle, we skip the current bucket in $\mathcal{T}_{\text{aph}}$, because the angle between any reference in this bucket and v is guaranteed to be larger than current min. We then continue to search the next nearest bucket in $\mathcal{T}_{\text{oph}}$(Step (2)).

(5) Once the search stops, we choose the cluster associated with r^* that has the min angle with v to insert v (Lines 18 - 19, Alg. 1).

4.2.3 *Search Max Angle*

To determine whether a newly inserted vector v should be promoted as a reference vector, `Suffice` computes the max angle between any vector v_i in cluster C_j and v. To avoid comparing with all vectors in C_j, we efficiently approximate this max angle using `Double Hash`. This approximated max angle is guaranteed to be larger than the actual value. Therefore, the safe region derived from it will *never erroneously* admit a vector into a cluster.

`Suffice` performs the inverse of the nearest cluster search. Instead of starting from nearby hash buckets, it begins from the most distant ones. This process shown in Fig. 6 is as follows:

(1) Compute the bucket indices of the incoming vector v in both hash tables (Lines 1 - 4, Alg. 2):

$$b_{\text{aph}}(v) = h_{\text{aph}}(v), \quad b_{\text{oph}}(v) = h_{\text{oph}}(v)$$

In Fig. 6(a), all existing vectors have already been mapped into hash tables $\mathcal{T}_{\text{oph}}$ and $\mathcal{T}_{\text{aph}}$. $R_3.O_3$

(2) Perform a farthest neighbor search in $\mathcal{T}_{\text{oph}}$ (Lines 4 - 6, Alg. 2). Starting from the most distant bucket index $b^{far}_{\text{oph}}(v)$, `Suffice` searches in decreasing order to $b_{\text{oph}}(v)$. For each non-empty bucket, it retrieves a set of candidate vectors $\mathcal{V}_{\text{oph}}$ in the current bucket. Let b_{oph} be the current bucket of $\mathcal{T}_{\text{oph}}$. In Fig. 6(a) step 1, it searches the OPH bucket $b^{far}_{\text{oph}}(v)$ farthest from the bucket containing the incoming vector v. It contains v_3 and v_5 in the candidate vectors $\mathcal{V}_{\text{oph}}$. In Fig. 6(a) step 1, v is in the $R_3.O_3$

Algorithm 2: MaxAngleSearch(v, cluster $C_|$)

Input: Inserted vector v, cluster C_j, hash tables $\mathcal{T}_{\text{aph}}$, $\mathcal{T}_{\text{oph}}$
Output: Estimated maximum angle $\max_{v_j \in C_j} \angle(v, v_j)$

1 $\alpha \leftarrow \angle(v, r_1)$, $\Theta_{\max} \leftarrow 0$;
2 $b_{\text{aph}}(v) \leftarrow h_{\text{aph}}(v)$, $b_{\text{oph}}(v) \leftarrow h_{\text{oph}}(v)$;
3 **for** $b_{oph} \in \mathcal{T}_{oph}$ *sorted in* $|b_{oph} - b_{oph}(v)|$ **do**
4 **if** $\mathcal{T}_{oph}[b_{oph}] = \emptyset$ **then**
5 **continue**;
6 $\mathcal{G} \leftarrow \emptyset$;
7 **foreach** $v_j \in \mathcal{T}_{oph}[b_{oph}] \cap C_j$ **do**
8 $b_{\text{aph}}(v_j) \leftarrow h_{\text{aph}}(v_j)$;
9 $\mathcal{G}[\, b_{\text{aph}}(v_j)\,] \leftarrow \mathcal{G}[\, b_{\text{aph}}(v_j)\,] \cup \{v_j\}$;
10 **foreach** *bucket* b_{aph} *in sorted keys of* $\mathcal{G}$ **do**
11 $\beta_{\max} = |b_{\text{aph}} - b_{\text{aph}}(v)| \cdot \delta_1$;
12 $\phi_{\max} = |b_{\text{oph}} - b_{\text{oph}}(v)| \cdot \delta_2$;
13 **if** $\phi_{\max} \geq \pi/2$ **then**
14 $\widehat{\Theta}_{\text{up}} \leftarrow \arccos\left(\cos\alpha \cdot \cos\beta_{\max} + \sin\alpha \cdot \sin\beta_{\max} \cdot \cos\phi_{\max}\right)$;
15 **else**
16 $\widehat{\Theta}_{\text{up}} \leftarrow \alpha + \beta_{\max}$;
17 **if** $\widehat{\Theta}_{up} \leq \Theta_{\max}$ **then**
18 **continue**;
19 **foreach** $v_j \in \mathcal{G}[b_{aph}]$ **do**
20 $\Theta_{v_j} \leftarrow \angle(v, v_j)$;
21 $\Theta_{\max} \leftarrow \max(\Theta_{\max}, \Theta_{v_j})$;
22 **return** $\Theta_{\max}$

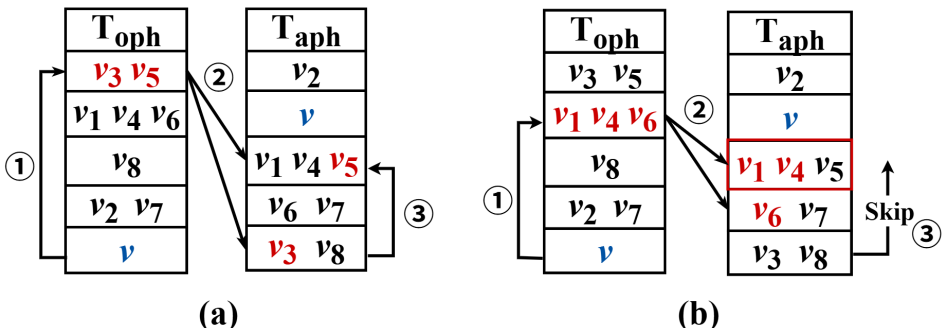


Fig. 6. Search Max Angle.

OPH Bucket 5. `Suffice` searches the OPH bucket furthest from the incoming vector v, which is Bucket 1 containing v_3 and v_5 from $\mathcal{V}_{\text{oph}}$.

(3) Perform a farthest neighbor search in $\mathcal{T}_{\text{aph}}$ (Lines 7 - 10, Alg. 2). For each vector $v_j \in \mathcal{V}_{\text{oph}}$, `Suffice` determines its APH bucket $b_{\text{aph}}(v_j)$. Starting from the most distant bucket index $b^{far}_{\text{aph}}(v_j)$, it searches in an order from outer to inner rings. Let b_{aph} be the current bucket of $\mathcal{T}_{\text{aph}}$. In Fig. 6(a) Step 2, candidate vector v_3 and v_5 from $\mathcal{V}_{\text{oph}}$ are mapped to APH buckets 3 and 5, respectively. The search starts from the distant bucket of v_3 and proceeds to the bucket of v_5, as shown in Fig. 6(a) Step 3.

(4) For these candidate vectors in current bucket b_{oph} and b_{aph}, `Suffice` estimates the **upper bound** on the angle $\angle(v, v_j)$ between any vector v_j in these two buckets and v using the following equation (Lines 11 - 13, Alg. 2):

$$\cos\angle(v, v_j) = \cos(\alpha)\cos(\beta) + \sin(\alpha)\sin(\beta)\cos(\phi) \tag{6}$$

where $\alpha = \angle(v, r_1)$, $\beta = \angle(v_j, r_1)$ and $\phi = \angle(\hat{v}, \hat{v_j})$.

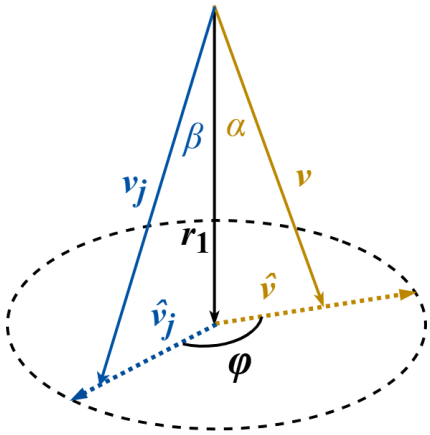


Fig. 7. Upper Bound of the Angle Between v_j And v.

Fig.7 visualizes Eq. 6 in the angle space. Note that α is fixed for a given vector v and r_1, making Eq. 6 a function of β and ϕ. In Fig. 6(a) step 3, when we search vector v_j lying in current bucket $b_{\text{aph}}(v_5)$, given the bucket assignments of both APH and OPH, we can deduce the possible ranges of angle β and angle ϕ, which bound the angle between x and v. For example, if we have $\beta \in (10°, 15°]$ and $\phi \in (160°, 165°]$, choosing $\beta = 15°$ and $\phi = 165°$ obviously yields the largest possible angle $\angle(v, v_j)$, providing an upper bound.

If the upper bound function were monotonic w.r.t. β and ϕ, then we could safely use their boundary values to compute a conservative upper bound. If this bound is already lower than the current max angle, we can skip evaluating the reference in this current bucket entirely. Lemma 2 below establishes this monotonicity.

Lemma 2. *Given a fixed α, and variable $\beta \in [0, \pi/2]$, $\phi \in [\pi/2, \pi]$, the function $f(\beta, \phi) = \cos(\alpha)\cos(\beta) + \sin(\alpha)\sin(\beta)\cos(\phi)$ monotonically decreases with respect to β and ϕ.*

Proof. Since α is fixed, we differentiate w.r.t. each variable.

For ϕ: $\frac{\partial f}{\partial \phi} = -\sin(\alpha)\sin(\beta)\sin(\phi)$. Given that $\phi \in [\pi/2, \pi]$, we have $\sin(\phi) \geq 0$. In our cases, α and β are the angles between two vectors v, x and reference r_1 within a cluster. Therefore, the max angle is the clustering threshold θ, which is much smaller than $\pi/2$. Since α,$\beta \in [0, \pi/2]$, then $\sin(\alpha), \sin(\beta) \geq 0$, therefore: $\frac{\partial f}{\partial \phi} \leq 0$. Thus, f is monotonically decreasing with respect to ϕ.

For β: $\frac{\partial f}{\partial \beta} = -\cos(\alpha)\sin(\beta) + \sin(\alpha)\cos(\beta)\cos(\phi)$. Since $\cos(\phi) \leq 0, \sin(\beta) \geq 0$, and $\cos(\beta) \geq 0$, the first and second terms are non-positive. Hence: $\frac{\partial f}{\partial \beta} \leq 0$. This shows that $f(\beta, \phi)$ monotonically decreases w.r.t both β and ϕ when ϕ is obtuse. □

We show how to leverage Lemma 2 to upper bound $\angle(v, v_j)$ (Lines 11 - 20, Alg. 2). We consider the buckets to which vectors v and v_j belong within the two hash tables. Specifically, $b_{\text{aph}}(v)$ and $b_{\text{aph}(v_j)}$ correspond to the current bucket indices of v and v_j in $\mathcal{T}_{\text{aph}}$, where each bucket corresponds to an angular range with a width of $\delta_1 = \theta/\tau_1$. Hence, β can be upper bounded by $\beta_{\max} = |b_{\text{aph}}(v_j) - b_{\text{aph}}(v)| \cdot \delta_1$.

Similarly, $b_{\text{oph}}(v)$ and $b_{\text{oph}(v_j)}$ represent the indices of the current buckets of v and v_j in $\mathcal{T}_{\text{oph}}$. Each bucket covers an angular range of width $\delta_2 = \pi/\tau_2$. The angular deviation ϕ is upper bounded by $\phi_{\max} = |b_{\text{oph}}(v_j) - b_{\text{oph}}(v)| \cdot \delta_2$.

Substituting β and ϕ with their upper bounds, the max angle between v and any candidate vector v_j is upper bounded by:

$$\begin{aligned}\widehat{\theta}_{\text{upper}}(v, v_j) = \ & \arccos\big(\cos(\alpha)\cos(\beta_{\max}) \\ & + \sin(\alpha)\sin(\beta_{\max})\cos(\phi_{\max})\big)\end{aligned} \tag{7}$$

If this upper bound $\widehat{\theta}_{\text{upper}}(v, v_j)$ is smaller than the currently maintained max angle, we skip the remaining comparisons of the bucket in $\mathcal{T}_{\text{oph}}$. We then continue to search the next farthest bucket in $\mathcal{T}_{\text{oph}}$ (step (2)).

(5) Because Eq. 6 is not monotonic when $\phi \in [0, \pi/2]$, the upper bound in Eq. 7 is not valid. In this case, we apply the triangle inequality–based approximation to derive an upper bound and use it to reduce search cost (Lines 21 - 24, Alg. 2).

$$\widehat{\theta}_{\text{upper}}(v, v_j) = \alpha + \beta_{max}$$

(6) Once the search stops, it uses the final approximation as the max angle w.r.t. vector v.

Time and Space Complexity. We analyze the time and space complexity of `Suffice`, covering Algorithm 1 and Algorithm 2.

Because Alg. 1 effectively skips buckets, the number of buckets that have to be searched, denoted as b, is much smaller than the total number of buckets. Within these b buckets, the average number of references is $\frac{m}{\tau_1\tau_2}$, where m is the total number of references($m \ll n$) and τ_1, τ_2 denote the number of buckets in the two tables of `Double Hash`, respectively. Therefore, the overall expected time complexity of Alg. 1 is $O\Big(\frac{bm}{\tau_1\tau_2} \cdot d\Big)$, which is significantly smaller than scanning all references. The complexity analysis of Alg. 2 is similar to that of Alg. 1. Within c buckets that have to be searched, the average number of vectors is $\frac{n}{\tau_1\tau_2}$, where n is the total number of vectors. Therefore, the expected time complexity of Alg. 2 is $O\Big(\frac{cn}{\tau_1\tau_2} \cdot d\Big)$.

The overall per-insertion cost is thus $O(\lambda d)$, where $\lambda = \frac{cn}{\tau_1\tau_2} + \frac{bm}{\tau_1\tau_2} \ll n$. The time complexity of inserting n vectors is $O(\lambda nd)$, much smaller than the $O(n^2 d)$ complexity of Naive.

`Suffice` stores each active vector in global memory, requiring $O(nd)$ space. `Double Hash` maintains $\tau_1 \times \tau_2$ buckets covering all n vectors, and each bucket only stores the indices of vectors. This contributes an additional space of $O(n + \tau_1 + n + \tau_2)$. Hence, the total space complexity of `Suffice` is $O(nd + 2n + \tau_1 + \tau_2) = O(nd)$.

5 Experiments

Our experiments focus on answering the following key questions:

- How efficient is `Suffice` compared to baselines?
- How scalable is `Suffice` to the volume of vector streams?
- How effective is `Suffice` in clustering quality?
- How does `Suffice` handle different types of updates?
- How effective is our cluster definition compared to others?
- How effective is `Suffice` in handling data distribution drift?
- How sensitive is `Suffice` to the parameters τ_1 and τ_2?

5.1 Experiment Setup and Methodologies

Experimental Infrastructure. We conduct all experiments on an instance hosted on Google Cloud Platform (GCP). It is configured with the `e2-medium` machine type, which includes 2 virtual CPUs and 4 GB of memory. The operating system is Debian 12.

Table 1. Update Dynamics of the real-world datasets

Dataset	Vector Updates	Dimension Updates	Value Updates
Retailrocket	11.8K	12.1K	42.5K
Reddit	6.3K	7.2K	67.4K
BGP	2.2K	15.5K	782K

Real Datasets. We evaluate our proposed methods on three real-world datasets: Retailrocket [31], Reddit [8] and BGP [32]. We summarize their key statistics in Table 1, including data volume, dimensionality, and the number of three types of updates across windows. Each stream starts with an empty window. Therefore, the number of vector/dimension updates is equivalent to the total number of vectors/dimensionality in each dataset.

Retailrocket. It is an e-commerce dataset collected from the Retailrocket platform [31]. It captures the user interactions with items on a commercial website over a 4 month period, including views, cart additions, and purchases. It contains 11,769 users and 12,055 items. We construct user vectors based on their purchase behavior and update them accordingly, as discussed in Sec. 1.

Reddit. We collect all users, posts and comments under the `r/uwaterloo` subreddit from the year 2015, using the official Reddit API. The dataset comprises a total of 6,366 users, 80,252 comment records, and 7,244 original posts. We construct user vectors based on commenting behavior, as discussed in Sec. 1.

BGP. We use RIB snapshots from the RIPE RIS project[32] to construct this dataset. Each snapshot records the global BGP routing table as observed from multiple vantage points on the Internet. We collect RIB snapshots from one of the vantage points over 90 days and extract Autonomous Systems(AS) nodes and their AS-PATHs. This dataset comprises a total of 2,213 AS nodes and 15,508 AS-PATHs. We represent each AS node as a vector, where each dimension represents an AS-PATH.

Synthetic Datasets. We develop a data generator to produce vector streams with a controlled number of clusters, data update types, and update frequencies. We first generate cluster centers by randomly sampling vectors from a multivariate Gaussian distribution. We then generate vectors w.r.t. each cluster by applying a random angular perturbation to its center. We generate three types of streaming datasets, each targeting one specific update type, namely value update, vector update, and dimension update. Within each window, it produces different numbers of updates.

Metrics. First, we measure processing time. As a vector stream is processed with a sliding window, by default we measure the *cumulative processing time* across all windows. Depending on the experimental context, it can be reported as the *average processing time* per window. We also measure clustering quality using widely adopted metrics, including *silhouette score*, *intra-cluster similarity*, and *inter-cluster dissimilarity* [29, 34].

Methods. We evaluate the following methods. The baseline methods represent the major categories of clustering algorithms discussed in Sec.6, focusing on those suitable for streaming and high-dimensional settings. We adapt baselines to comply with our cluster definition (Def.2.1), because most baseline algorithms, originally designed for low-dimensional data, rely on Euclidean distance, which becomes ineffective in high-dimensional spaces due to the curse of dimensionality. Specifically, for algorithms without a representative reference or centroid, we conduct pairwise similarity checks to ensure that all vectors in a cluster satisfy our cluster criterion. For centroid-based methods, we incorporate our safe region idea (Def.3.2) to enhance efficiency during incremental updates. For each method, we also report the corresponding hyperparameter configurations. In general, we follow the authors' recommendations, test multiple parameter settings, and select the one that achieves the best performance on validation data. The parameter choices for our proposed methods remain consistent across different datasets.

(1) D-Stream [14]: It is a popular density-based stream clustering algorithm originally designed for low-dimensional data. We replace its grid-partitioning with an angular partitioning over sphere space. We estimate the density of each grid and prune spare grids with the idea in the original paper.

Hyperparameters: Following the guideline of the original paper, we find the dense-grid threshold $\mu_{dense} = 3$ and spare-grid threshold $\mu_{sparse} = 0.5$ yield the most stable clustering results across datasets.

(2) DenStream [11]: DenStream is a two-phase density-based stream clustering algorithm. During the online phase, it maintains a set of micro-clusters and assigns a new vector to an existing micro-cluster with our safe region idea. At offline, it links nearby micro-clusters to generate the final clusters.

Hyperparameters: We set parameter $N = 1.0$ to disable scaling the clustering threshold θ. We set $weight = 1.0$ that assigns an equal density contribution to all vectors.

(3) Links [23]: Links maintains multiple subclusters, each represented by a center. It determines whether to insert a new vector into an existing subcluster with our safe region idea. It connects two subclusters if their centers are sufficiently similar, forming a graph to represent the final clusters.

Hyperparameters: T_c controls the minimum similarity between a vector and a subcluster centroid. We set T_c to the boundary angle defined in Sec. 3.1. T_s is the subcluster connecting threshold, fixed to our clustering threshold θ.

(4) HSRC [13]: Starting from a root node representing all vectors, it hierarchically merges high-dimensional vectors into a clustering tree with spherical k-means. A vector is assigned to a subcluster if it satisfies our cluster definition.

Hyperparameters: After testing a bunch of combinations, we set the parameter k in spherical k-means clustering to 10 and the parameter s to 0.2 that determines the proportion of representative points at each hierarchy level.

(5) Naive: It conducts exhaustive pairwise similarity comparisons to insert a new vector into clusters (Sec. 3).

(6) Fix: It uses one reference per cluster to speed up vector insertion; it does not update references as window slides (Sec. 3.1).

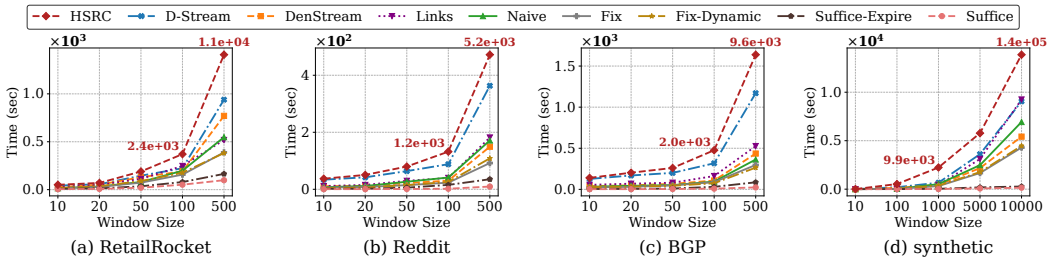
(7) Fix-Dynamic: It extends Fix to dynamically select a new reference for each cluster when window slides. The new reference is chosen as the vector that has the smallest average angular distance to all other vectors within the cluster.

(8) `Suffice`: our optimized approach discussed in Sec. 4.

(9) `Suffice`-Expire: Unlike Suffice, it expires old references as window moves.

D-Stream, DenStream, Links, and HSRC do not naturally support dimension update. To run experiments, we assume that they are aware of all dimensions in a vector stream in advance and recluster all affected vectors when dimension update occurs. *HSRC*, originally designed as a static algorithm, is super slow in handling dynamic streams. To ensure readability, we mark its processing time with a scale different from other methods.

Experimental Parameter Settings. For real-world datasets, we set the window size to 500, constrained by the available data volume. For synthetic datasets, we use a larger window size of 1000, as the data volume can be freely controlled. Moreover, in Sec. 5.2, we conduct scalability experiments with window sizes varying from 10 to 100K, demonstrating the ability of `Suffice` to efficiently handle high-volume data streams. For all experiments, we set parameters τ_1, and τ_2, i.e., the number of buckets in our APH and OPH tables, to 50. For the real-world Reddit, RetailRocket, and BGP datasets, we set the cluster threshold θ as 35°, 30°, and 30° respectively, as this value provides a good trade-off between compactness and separability. For synthetic datasets, we use $\theta = 35°$ to make the inter-cluster boundaries more discernible under controlled synthetic conditions.



(a) RetailRocket (b) Reddit (c) BGP (d) synthetic

Fig. 8. Evaluation on Different Window Sizes.

5.2 Evaluation on Varying Window Sizes

This experiment evaluates the scalability of `Suffice` to large volume of data. We evaluate 9 methods on three real-world datasets and a synthetic dataset. We vary the window size from 10 to 500 on real datasets due to the limitation of data volume. We vary the window size from 10 to 100K on synthetic datasets demonstrating the scablibity of `Suffice` to high-volume data streams. We run 200 windows per experiment. Fig. 8 shows `Suffice` consistently outperforms other methods, achieving up to 937× speedup on HSRC and 96× speedup on others. This performance gain becomes more evident as the window size increases. Larger windows require more similarity comparison operations, which results in substantial overhead for the baselines.

The advantage of `Suffice` comes primarily from the joint effect of the `Adaptive Mult Ref` and `Double Hash` optimizations. `Adaptive Mult Ref` accelerates clustering by maintaining multiple reference vectors per cluster to enlarge the coverage of *safe region*, thereby avoiding unnecessary similarity checks. In contrast, the single-reference strategy such as Fix leads to a much smaller safe region and therefore often requires many pairwise similarity comparisons to find an appropriate cluster for each new vector.`Double Hash` further boosts efficiency. It quickly identifies the most promising cluster for each to each new vector, avoiding similarity comparisons with all references in all clusters. Additionally, `Double Hash` accelerates the computation of the cluster's *maximum angle* by narrows the search space to a small candidate vector set.

Although Fix-Dynamic continuously updates reference vectors, its single reference design limits the coverage of the *safe region* compared to `Adaptive Mult Ref`. As a result, Fix and Fix-Dynamic frequently fallback to pairwise comparison when the single reference fails to absorb the vector. Suffice-Expire removes expired references. This results in a smaller global safe region per cluster, in turn more pairwise comparisons when inserting new vectors. It is thus slower than `Suffice`. However, since `Double Hash` accelerates cluster search and promotion of new reference, its total processing time still outperforms other baselines.

HSRC, Links, D-Stream and DenStream methods are even slower than Naive because they incur heavy computational overhead when handling high-dimensional vector streams. HSRC is extremely expensive, because as a static method, it has to recluster all vectors from scratch by performing spherical k-means to rebuild the hierarchical tree when the sliding window moves. Links has to incrementally construct similarity graphs by performing frequent pairwise similarity comparisons among vectors, making it very slow. Similarly, D-Stream and DenStream suffer from pairwise checking the angle between grids or micro-clusters.

5.3 Evaluation of Clustering Quality

We evaluate clustering quality with the following three classical metrics: Silhouette Score [34], IntraSim [29], InterSim [29]. For Silhouette Score and IntraSim, a higher score indicates better clustering quality, while for InterSim, a smaller score is better.

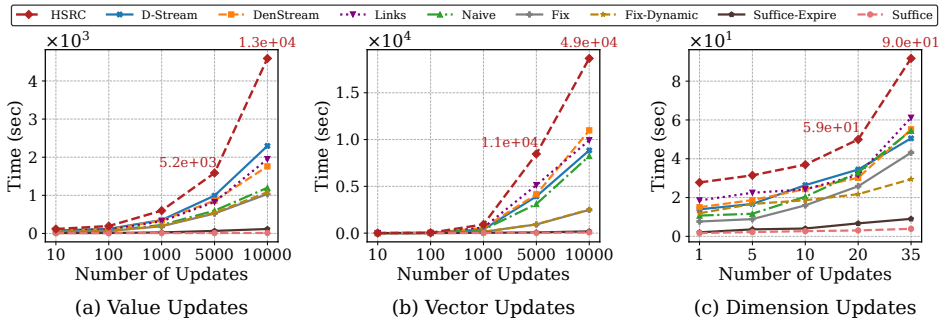
Table 2. Evaluation of Clustering Effectiveness on Three Real Datasets.

(a) **RetailRocket**

Method	Silhouette	IntraSim.	InterSim.
Naive	**0.8241**	**0.9743**	**0.0214**
Fix	0.8138	0.9444	0.0232
Fix-Dynamic	0.8140	0.9433	0.0245
Suffice-Expire	0.7823	0.9371	0.0359
Suffice	**0.8139**	**0.9529**	**0.0228**
Links	0.7790	0.9189	0.0327
HSRC	0.7527	0.8922	0.0362
DenStream	0.7960	0.9388	0.0298
D-Stream	0.7891	0.9354	0.0245

(b) **Reddit**

Method	Silhouette	IntraSim.	InterSim.
Naive	**0.7586**	**0.8931**	**0.0242**
Fix	0.7207	0.8326	0.0290
Fix-Dynamic	0.7132	0.8239	0.0275
Suffice-Expire	0.7045	0.8244	0.0307
Suffice	**0.7321**	**0.8534**	**0.0285**
Links	0.6759	0.8251	0.0806
HSRC	0.6546	0.8202	0.1014
DenStream	0.6826	0.8294	0.0518
D-Stream	0.6946	0.8340	0.0680



(a) Value Updates (b) Vector Updates (c) Dimension Updates

Fig. 9. Evaluation on Different Update Types.

Due to space limit, we report the results on two real-world datasets in Table 2. All methods produce high quality clusters, confirming the effectiveness of our cluster definition. Naive performs the best across all three metrics since it conducts exhaustive pairwise comparisons within clusters, ensuring maximal separation and compactness. However, `Suffice` is very close to Naive, and is superior to the other 7 methods in average. `Suffice` adopts multiple references per cluster to cover a broader safe region and thus accommodates more vectors with tight clusters. It adaptively selects non-overlapping references via a reward mechanism, which enhances intra-cluster compactness while preserving inter-cluster separability. Fix is slightly inferior to `Suffice` due to its limited coverage with a single reference vector. Based on multiple experimental runs, the marginal InterSim improvement of Fix over `Suffice` arises primarily from the randomness of the clustering process. Fix-Dynamic, selecting the vector that has smallest average angle with other vectors within a cluster as its reference, performs similarly to Fix. For Suffice-Expire, the removal of expired references sometimes cause overlapping clusters. Links, HSRC, D-stream and Denstream yield lower silhouette scores, indicating less consistent intra-cluster structure. Links constructs clusters through pairwise linkages based on local connectivity enforcing pairwise global threshold, leading to loose internal cohesion and overlapping clusters. HSRC constructs clusters via sparse representation enforcing strict boundary constraints, leading to ambiguous cluster boundaries in high-dimensional space. DenStream relies on single centroid, which may cause neighboring micro-clusters to overlap. D-Stream suffers from rigid grid alignment and lacks flexibility in adapting to dynamic clusters.

5.4 Evaluation on the Types of Data Update

We evaluate the influence of *value*, *vector*, and *dimension* update. We run this set of experiments on both synthetic and real-world datasets. Since HSRC is designed for static data, and Links, DenStream,

and D-Stream do not natively support value or dimension updates, we adapt these baselines under our cluster definition (Def.2.1) so that all methods can process three types of updates. We fix the window size to 1k and run 1k windows. We report the average processing time per window. `Suffice` achieves consistent performance gains across all scenarios.

5.4.1 *Synthetic Datasets: Varying the Number of Updates*

Value Update. Fig 9(a) illustrates the results under different numbers of value updates, ranging from 10 to 10k in a setting where each stream only updates vector values. `Suffice` consistently outperforms HSRC by up to 870× and all other baselines by around 120× across all volumes of update. Moreover, this performance advantage becomes increasingly significant as the number of updates grows. For HSRC, frequent re-insertions trigger reclustering and hierarchy reconstruction, leading to significant computational overhead. In other baselines, re-inserting an updated vector back into the cluster structure requires either exhaustive pairwise comparisons with all existing vectors(Naive), pairwise check with all vectors when merging subcluster (D-Stream, DenStream, Links) or fallback to pairwise comparisons when a vector falls out of the safe region of any references(Fix, Fix-Dynamic), which are expensive. In contrast, `Suffice` reduces the complexity of insert operation through `Adaptive Mult Ref` and `Double Hash`.

Vector Update. We vary the number of vector updates from 10 to 10k, namely only vector insertions and expirations. As discussed in Sec. 2.3, all algorithms simply remove an expired vector from its cluster. For each new vector, the standard insertion process is applied. As shown in Fig.9(b), `Suffice` consistently outperforms all methods in all settings by 118×.

Dimension Update. We vary the number of dimension updates from 1 to 35, namely only inserting or expiring dimensions. As shown in Fig.9(c), `Suffice` consistently outperforms HSRC by 30× and other baseline by 18 × in all settings. Dimension updates often impact a large number of vectors. HSRC reinserts all vectors, leading to prohibitive recomputation cost. Naive, Fix and Fix-Dynamic incur high computational costs due to repeated pairwise comparisons when re-inserting these vectors. Denstream, D-stream and Links suffers from pairwise comparisons when merging grids, micro-clusters or similarity graphs. `Suffice` efficiently handles dimension updates by leveraging its core optimizations: multiple reference-based insertion and fast vector search via `Double Hash`. As the number of updated dimensions increases, the performance advantage of `Suffice` becomes more pronounced, highlighting its scalability in evolving feature spaces.

5.4.2 *Real datasets*

We separate each type of update from the real datasets and run experiments separately. The data volumes for each dataset under different update types are summarized in Table 1.

We report the average processing time per window. Due to space limit, we only report the results on the RetailRocket dataset in Fig. 10. The results on the other two datasets show similar trend. `Suffice` achieves consistent performance gains across all three types of updates and across all different datasets, for the same reasons discussed in Sec. 5.4.1. As the number of value updates grows from 6.37K to 728K, the speedup of `Suffice` over all other baselines increases from 15× to 78×, indicating that `Suffice` scales better to the growing update volume.

5.5 Evaluation on Cluster Definition

We evaluate the clustering quality of four baseline methods under both their original cluster definitions and our proposed definition (Def.2.1). We also compare them with our `Suffice`. We run this set of experiments on the real Reddit dataset with the clustering threshold θ set to 35°. We fix the window size to 500 and run 200 windows. As shown in Table 3, our cluster definition substantially improves the clustering quality of the four baselines, while `Suffice` still consistently achieves superior clustering effectiveness. For Links, the original definition uses cosine similarity but only

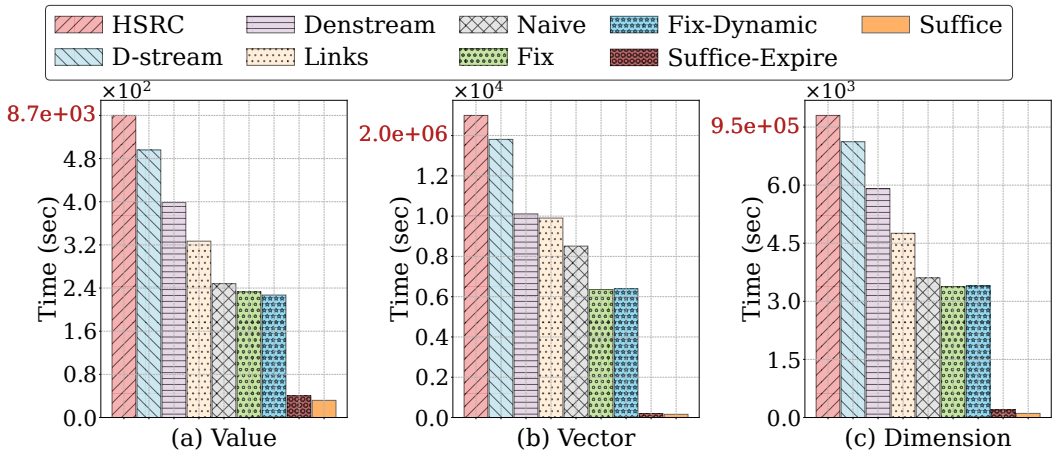


Fig. 10. Processing Time (RetailRocket): Update Types.

Table 3. Comparison of Clustering Effectiveness under Original and Proposed Definitions on Reddit.

Method	Our Definition			Oringinal Definition		
	Silhouette	IntraSim.	InterSim.	Silhouette	IntraSim.	InterSim.
Links	0.6759	0.8251	0.0806	0.5815	0.7626	0.0121
HSRC	0.6546	0.8202	0.1094	0.6062	0.7241	0.0036
D-Stream	0.6946	0.8340	0.0680	0.0542	0.3459	0.2765
DenStream	0.6826	0.8294	0.0518	-0.0209	0.2538	0.2184
Suffice	**0.7321**	**0.8534**	**0.0285**	/	/	/

checks the similarity between a vector and the subcluster centroid. This admits many vectors that violate pairwise constraints into the same cluster, leading to a higher IntraSim. Meanwhile, under our definition, pairwise checks restrict loose linking on similarity graph, effectively mitigating this problem. Similarly, HSRC originally merges some vectors that do not satisfy pairwise similarity, resulting in lower separation between clusters compared to our proposed criterion. For D-Stream and DenStream, their original definitions are designed for low-dimensional data and rely on Euclidean distance. In high-dimensional space, this metric becomes ineffective due to the curse of dimensionality.

5.6 Evaluation of Parameters τ_1 and τ_2

We run this experiment on synthetic datasets to evaluate the sensitivity of `Suffice` to the sizes τ_1, τ_2 of the two hash tables in `Double Hash`. We use a window size of 1000. Fig. 11 shows the cumulative processing time under varying parameter τ_1 and τ_2 from 10 to 100. The results clearly demonstrate that `Suffice` exhibits low sensitivity to changes in these parameters. Specifically, as τ_1 and τ_2 increase, the number of buckets within our `Double Hash` structure grows accordingly. Although `Suffice` might have to search more buckets, this results in fewer vectors per bucket. Therefore, the search cost remains stable. Note that `Suffice` uses `Double Hash` to speed up vector search. Therefore, τ_1 and τ_2 primarily affect the search efficiency and have little impact on clustering quality.

Practical Guideline: The above experiment indicates that τ_1 and τ_2 are not hard to tune. We recommend choosing τ_1 and τ_2 based on the volume of the vector streams, i.e, making sure that each bucket in average does not have too many vectors, e.g, no more than 100 vectors. That is, if

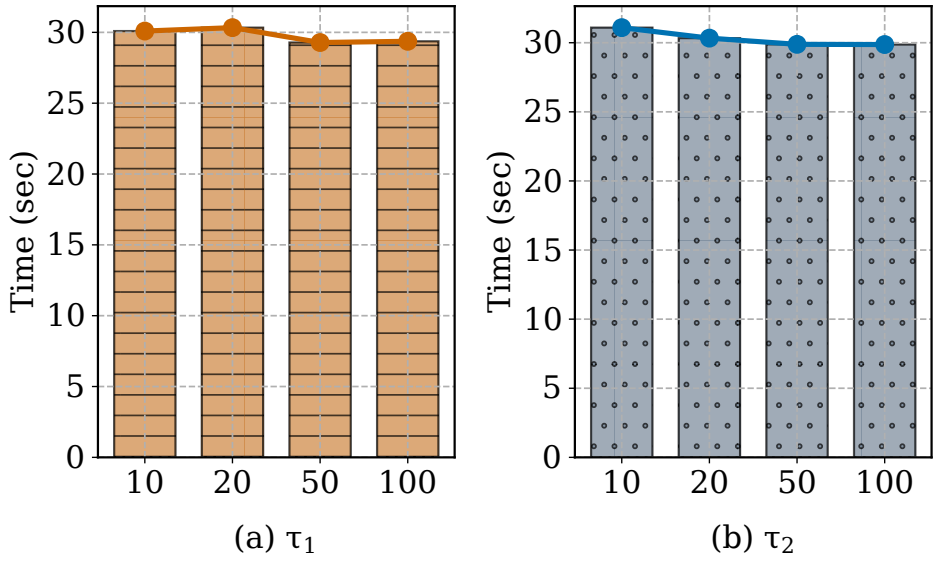


Fig. 11. Evaluation of Varying Parameters τ_1 and τ_2

there are in average N vectors in each window, $\tau_1, \tau_2 \approx \frac{N}{100}$. This is because a large bucket tends to lead to a upper/lower bound that is less effective in skipping buckets during nearest/farthest neighbor search.

6 Related Work

Density-based Clustering. D-Stream [14] partitions the feature space into a fixed grid structure and assigns each incoming data point to a grid. It continuously maintains a density coefficient for each grid through an exponential decay function, allowing the model to favor recent data. Adjacent grids with a density larger than a threshold are merged to form clusters. HDDSTREAM [26] introduces projected subspaces to address high dimensionality. It dynamically determines relevant subspaces for clustering by monitoring feature distributions, and applies density-based clustering within those subspaces. However, these methods often struggle with the exponential number of grid combinations in high-dimensional data. Our `Suffice` adopts an angle-based clustering strategy and uses a double hash structure to map each vector to a one-dimensional angular space, replacing high-dimensional grids with hash buckets. DenStream [11] avoids using the exponential number of grids by adopting a two-phase strategy. In the online phase, it maintains small but tight micro-clusters, each represented by a centroid. Each micro-cluster's weight is continuously updated through an exponential decay function, where outdated clusters gradually were removed. In the offline phase, DenStream periodically aggregates micro-clusters whose centroids lie within a density threshold, forming macro-clusters. Our experiments show that `Suffice` is orders of magnitude faster than D-Stream in clustering high-dimensional vector streams. Note that we do not compare against HDDSTREAM, because we do not target clustering in subspaces.

Distance-based Clustering. Links [23] maintains each cluster represented by a centroid. When a new data point arrives, Links treats it as a node. If its similarity to the existing cluster centroid exceeds a predefined threshold, a link is established between the node and cluster. Clusters are also connected by a link in the similarity graph if the similarity between their centroids is above a merging threshold. The connected clusters in the graph form the final cluster. HSRC [13] is a distance-based hierarchical clustering framework. It begins by treating each data point as an

individual cluster and computing a sparse representation for representative points. HSRC then iteratively merges clusters with the highest sparse affinities computed by sparse representation. This process continues until no cluster will be merged or a target hierarchy depth is reached. However, these methods often incur higher computational overhead under frequency update settings due to the complex maintenance of the connectivity graph and costly sparse representation. `Suffice` employs a double-hash structure that avoids the computation of insertions and deletions in the graph and hierarchical tree.

Subspace Clustering. StreamRPHash [12] leverages random projection to reduce dimensionality of incoming data streams and applies LSH [37] to map points to hash buckets for approximate clustering. In [5] the authors demonstrate that random projections can reduce dimensionality while still maintaining the separation between clusters with high probability. However, random projection inherently introduces information loss. Moreover, these methods often require recomputing projections in streaming data, which increases the computational cost. NGPCA [24] performs stream clustering by projecting high-dimensional data onto local subspaces using PCA [1], and then clustering these lower-dimensional representations. However, PCA is computationally expensive, especially as the dimensionality and the volume of the data increase. More recently, in [19] the authors propose a kernel-based subspace clustering algorithm. It introduces the Euler kernel as a similarity function to project data points in an implicit nonlinear space. However, it lacks efficient mechanisms to handle streaming data, since recomputing the kernel function can be computationally expensive.

Deep Learning-Based Clustering. AdapVAE [42] combines a variational autoencoder (VAE) with a nonparametric clustering prior to learn low-dimensional representations of the streaming data. As new data arrives, the model can dynamically add new clusters when necessary, without predefining the number of clusters. ADCN [6] is a deep learning framework that adjusts its neural network structure over time to keep up with changes in data, such as tracking when user behavior or patterns in the stream shift. Deep learning-based approaches are effective in extracting nonlinear representations from high-dimensional data. However, their high computational cost, training complexity, and sensitivity to hyperparameter tuning make them difficult to deploy in real-time streaming environments.

7 Conclusion

We propose `Suffice`, a scalable framework to cluster high-dimensional vector streams. It features a cluster definition native to the dynamic and high-dimensional nature of vector streams. The multi-reference and `Double Hash` optimizations jointly make it scalable to high-speed, large-volume vector streams. Experimental results on real-world datasets demonstrate its superiority over alternative approaches. A potential limitation of `Suffice` stems from its reliance on angular similarity. Although widely used in measuring the similarity of high-dimensional vectors, it disregards the magnitude of vectors. Consequently, `Suffice` may not be the best fit when the vector magnitude matters. Another limitation is that `Suffice` performs less efficiently when incurring highly frequent dimension updates. Each dimension update triggers the expiration and reinsertion of all affected vectors, resulting in frequent vector re-insertions and hence heavy computational overhead.

Acknowledgments

This research was supported in part by NSF under grants DBI-2327954 and CCF-2106621 and by Amazon Research Award (2023 Fall).

References

[1] Hervé Abdi and Lynne J Williams. 2010. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics* 2, 4 (2010), 433–459.

[2] Marcel R Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. 2012. Streamkm++ a clustering algorithm for data streams. *Journal of Experimental Algorithmics (JEA)* 17 (2012), 2–1.

[3] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. 2001. On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*. Springer, 420–434.

[4] Bahaa Al-Musawi, Philip Branch, and Grenville Armitage. 2016. BGP anomaly detection techniques: A survey. *IEEE Communications Surveys & Tutorials* 19, 1 (2016), 377–396.

[5] Laura Anderlucci, Francesca Fortunato, and Angela Montanari. 2022. High-dimensional clustering via random projections. *Journal of Classification* (2022), 1–26.

[6] Andri Ashfahani and Mahardhika Pratama. 2022. Unsupervised continual learning in streaming environments. *IEEE transactions on neural networks and learning systems* 34, 12 (2022), 9992–10003.

[7] Gill Barequet and Gershon Elber. 2005. Optimal bounding cones of vectors in three dimensions. *Inform. Process. Lett.* 93, 2 (2005), 83–89.

[8] Jason Baumgartner, Savvas Zannettou, Brian Keegan, Megan Squire, and Jeremy Blackburn. 2020. The pushshift reddit dataset. In *Proceedings of the international AAAI conference on web and social media*, Vol. 14. 830–839.

[9] Irina Beregovskaya and Mikhail Koroteev. 2021. Review of clustering-based recommender systems. *arXiv preprint arXiv:2109.12839* (2021).

[10] Monowar H Bhuyan, Dhruba Kumar Bhattacharyya, and Jugal K Kalita. 2013. Network anomaly detection: methods, systems and tools. *Ieee communications surveys & tutorials* 16, 1 (2013), 303–336.

[11] Feng Cao, Martin Estert, Weining Qian, and Aoying Zhou. 2006. Density-based clustering over an evolving data stream with noise. In *Proceedings of the 2006 SIAM international conference on data mining*. SIAM, 328–339.

[12] Lee A Carraher, Philip A Wilsey, Anindya Moitra, and Sayantan Dey. 2016. Random projection clustering on streaming data. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. IEEE, 708–715.

[13] Jie Chen, Hua Mao, Yuanbiao Gou, and Xi Peng. 2024. Hierarchical Sparse Representation Clustering for High-Dimensional Data Streams. *arXiv preprint arXiv:2409.04698* (2024).

[14] Yixin Chen and Li Tu. 2007. Density-based clustering for real-time stream data. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 133–142.

[15] Zhengdao Chen, Xiang Li, and Joan Bruna. 2017. Supervised community detection with line graph neural networks. *arXiv preprint arXiv:1705.08415* (2017).

[16] Mohammed Ghesmoune, Mustapha Lebbah, and Hanene Azzag. 2016. State-of-the-art on clustering data streams. *Big Data Analytics* 1, 1 (2016), 13.

[17] Michael Hahsler and Matthew Bolaños. 2016. Clustering data streams based on shared density between micro-clusters. *IEEE transactions on knowledge and data engineering* 28, 6 (2016), 1449–1461.

[18] Sergio Hernandez, Pedro Alvarez, Javier Fabra, and Joaquin Ezpeleta. 2017. Analysis of users' behavior in structured e-commerce websites. *IEEE Access* 5 (2017), 11941–11958.

[19] Ruohe Huang, Ruliang Xiao, Weifu Zhu, Ping Gong, Jinhui Chen, and Imad Rida. 2021. Towards an efficient real-time kernel function stream clustering method via shared nearest-neighbor density for the IIoT. *Information Sciences* 566 (2021), 364–378.

[20] Hyeyoung Ko, Suyeon Lee, Yoonseo Park, and Anna Choi. 2022. A survey of recommendation systems: recommendation models, techniques, and application fields. *Electronics* 11, 1 (2022), 141.

[21] Thomas Krenc, Robert Beverly, and Georgios Smaragdakis. 2020. Keep your communities clean: exploring the routing message impact of bgp communities. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*. 443–450.

[22] Yongjae Lee and Woo Chang Kim. 2014. Concise formulas for the surface area of the intersection of two hyperspherical caps. *KAIST Technical Report* (2014).

[23] Philip Andrew Mansfield, Quan Wang, Carlton Downey, Li Wan, and Ignacio Lopez Moreno. 2018. Links: A high-dimensional online clustering method. *arXiv preprint arXiv:1801.10123* (2018).

[24] Nico Migenda, Ralf Möller, and Wolfram Schenck. 2024. NGPCA: Clustering of high-dimensional and non-stationary data streams. *Software Impacts* 20 (2024), 100635.

[25] Hyeon-Suk Na, Chung-Nim Lee, and Otfried Cheong. 2002. Voronoi diagrams on the sphere. *Computational Geometry* 23, 2 (2002), 183–194.

[26] Irene Ntoutsi, Arthur Zimek, Themis Palpanas, Peer Kröger, and Hans-Peter Kriegel. 2012. Density-based projected clustering over high dimensional data streams. In *Proceedings of the 2012 SIAM international conference on data mining*. SIAM, 987–998.

[27] A James O'malley and Peter V Marsden. 2008. The analysis of social networks. *Health services and outcomes research methodology* 8, 4 (2008), 222–269.
[28] Bibekananda Patra and Sandipan Bandyopadhyay. 2024. Determination of the Minimum Enclosing Cone of a Finite Collection of Cones Sharing the Same Vertex. *Available at SSRN 4801268* (2024).
[29] Bhavani Raskutti and Christopher Leckie. 1999. An evaluation of criteria for measuring the quality of clusters. In *Ijcai*, Vol. 99. 905–910.
[30] Yakov Rekhter, Tony Li, and Susan Hares. 2006. *A border gateway protocol 4 (BGP-4).* Technical Report.
[31] RetailRocket. 2017. RetailRocket Recommender System Dataset [Online]. https://www.kaggle.com/datasets/retailrocket/ecommerce-dataset.
[32] RIPE NCC. 2014. RIPE RIS Raw Data and RIB Snapshots. https://ris.ripe.net/.
[33] Giulio Rossetti and Rémy Cazabet. 2018. Community discovery in dynamic networks: a survey. *ACM computing surveys (CSUR)* 51, 2 (2018), 1–37.
[34] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65.
[35] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. 2007. Collaborative filtering recommender systems. In *The adaptive web: methods and strategies of web personalization*. Springer, 291–324.
[36] Ben A Scott, Michael N Johnstone, Patryk Szewczyk, and Steven Richardson. 2025. BGP anomaly detection as a group dynamics problem. *Computer Networks* 257 (2025), 110926.
[37] Malcolm Slaney and Michael Casey. 2008. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal processing magazine* 25, 2 (2008), 128–131.
[38] Meng Wang, Chaokun Wang, Jeffrey Xu Yu, and Jun Zhang. 2015. Community detection in social networks: an in-depth benchmarking study with a procedure-oriented framework. *Proceedings of the VLDB Endowment* 8, 10 (2015), 998–1009.
[39] Zhipeng Wang and David W Scott. 2019. Nonparametric density estimation for high-dimensional data—Algorithms and applications. *Wiley Interdisciplinary Reviews: Computational Statistics* 11, 4 (2019), e1461.
[40] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna PN Puttaswamy, and Ben Y Zhao. 2009. User interactions in social networks and their implications. In *Proceedings of the 4th ACM European conference on Computer systems*. 205–218.
[41] Di Yang, Zhenyu Guo, Elke A Rundensteiner, and Matthew O Ward. 2011. Clues: a unified framework supporting interactive exploration of density-based clusters in streams. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. 815–824.
[42] Tingting Zhao, Zifeng Wang, Aria Masoomi, and Jennifer G Dy. 2019. Streaming adaptive nonparametric variational autoencoder. *arXiv preprint arXiv:1906.03288* (2019).

Received July 2025; revised October 2025; accepted November 2025