VStream: A Distributed Streaming Vector Search System

Shenghao Gong
Zhejiang University
gongshenghao@zju.edu.cn

Haobo Sun
Zhejiang University
haobosun@zju.edu.cn

Ziquan Fang
Zhejiang University
zqfang@zju.edu.cn

Liu Liu
Zhejiang University
liu2@zju.edu.cn

Lu Chen
Zhejiang University
luchen@zju.edu.cn

Yunjun Gao
Zhejiang University
gaoyj@zju.edu.cn

ABSTRACT

Vector search is widely employed in recommendation systems, search engines, etc. With the explosive growth of online data and streaming processing engines, streaming vector search has attracted increasing research attention. However, prevailing vector search systems like Vearch, Vespa, and Milvus typically operate as external batch services for streaming processing requirements, resulting in sub-optimal performance for streaming processing scenarios.

In this paper, we propose VStream, a distributed streaming vector search system. Implementing such a system is non-trivial, raising three technical challenges in streaming adaptability, system scalability, and real-time response. Specifically, VStream offers a dynamic partitioner that adapts to data distribution changes in vector streams. Additionally, VStream features an effective hierarchical storage architecture facilitated by streaming state management, enabling a hybrid of four-level storage media with diverse access speeds and targets. Furthermore, VStream utilizes dynamic hot-cold patterns, such as access frequency, in the streaming vector data, incorporating a specialized hot-cold separation mechanism to enhance query efficiency. Extensive experiments prove that VStream outperforms existing vector search systems, e.g., achieving 251–373× improvements in query efficiency, 2.2–2.5× savings in CPU usage, and 1.5–2.0× reductions in memory overhead.

PVLDB Reference Format:
Shenghao Gong, Haobo Sun, Ziquan Fang, Liu Liu, Lu Chen, Yunjun Gao. VStream: A Distributed Streaming Vector Search System. PVLDB, 18(6): 1593 - 1606, 2025.
doi:10.14778/3725688.3725692

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ZJU-DAILY/VStream

1 INTRODUCTION

The proliferation of digital technologies and web services has generated vast volumes of unstructured data, encompassing text, images, and audio formats. It is challenging to process unstructured data due to its high dimensionality. Recently, learning-based embedding models have emerged as mainstream solutions. They transform unstructured data into high-dimensional vectors, showing widespread applications in recommendation systems [59, 61] (e.g., Wide&Deep [33] and YouTube DNN [36]), search engines [25, 73] (e.g., DSSM [53] and ColBERT [58]), and group analysis [43, 71]. The current success of large-scale models such as GPT-4 [28] and LLaMA [81] further solidify their position at the forefront. Among these, ***vector search*** plays a pivotal role by seeking the top-k nearest vectors [21] to a query vector, which has drawn increasing attention [22, 41, 55, 57, 68]. To tackle the storage and computation overhead of vector search, a multitude of vector-oriented systems [10, 12, 15–17, 64, 83] have been developed.

Recently, the explosive growth of online data in social media, e-commerce, and other fields, alongside the development of streaming processing engines (SPEs) such as Kafka [60], Flink [29, 30], and Spark Streaming [91], has led to a gradual shift in vector processing mode from ***batch manner*** to ***streaming manner***. Actually, streaming vector searches serve a wide range of applications, including instant search [50], online product recommendation [48], running log analysis [51], and online processing of emails [37] and news [82], require higher processing throughput and lower latency.

However, to perform streaming vector searches, existing well-known vector-oriented systems, such as Milvus [49, 83], Vearch [64], Vespa [16], Weaviate [17], Vald [15], Qdrant [12], and Pinecone [10], primarily operate as external services, separate from the streaming processing architecture. As shown in Fig. 1, data generated by users in the APP/WEB clients is collected into messaging systems such as Kafka [60], generating data streams. Based on that, SPEs like Flink [30] subscribe to the data streams from the upstream messaging systems. Then, as highlighted by the orange shaded area, the SPE invokes vector search services from external systems

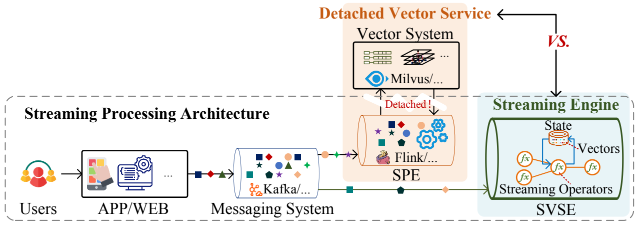


Figure 1: Research Motivation. Orange Block: Existing detached streaming vector search service. Green Block: Our streaming vector search engine (SVSE).

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725692

like Milvus [83] to retrieve target content. The detachment of SPEs and the vector systems has led to two limitations.

① **Inefficient update.** To support the connection with SPE, vector systems [15–17, 49, 83] provide interfaces for streaming read and write but lack optimized indexing and updating. This leads to significantly increased query latency in the streaming processing architecture which involves frequent small-batch or single-record data writes. For example, a Milvus instance housing 100 million vectors can maintain query latency under 100ms when no writes occur. However, with concurrent writes at 8000 QPS (queries per second), query latency quickly rises to 1.5 seconds.

② **Cross-node data transmission.** Although some vector systems have partially fulfilled the dynamic update requirements of vectors through offline and online index separation [15–17, 64] or LSM tree [49, 83], they typically employ a store-compute separation architecture. As a result, insertion and query requests, as well as query results, must be transmitted across nodes, leading to a significant increase in processing latency.

In contrast to existing studies that design another detached vector service for streaming vector searches, we choose to directly build an engine integrating streaming processing and vector search, termed as the streaming vector search engine (SVSE). In SVSE, the above two limitations can be solved. As depicted by the green block in Fig. 1, by utilizing state management capabilities of SPEs for vector storage, SVSE inherently supports streaming read and write operations of vectors. Moreover, streaming operators can efficiently access the vector data locally, eliminating the necessity for frequent cross-node data transfers during queries. However, designing an SVSE faces three specific challenges in streaming scenarios.

Challenge I: How to achieve dynamic partitioning to adapt to real-time vector data streams? Existing vector systems [10, 12, 15–17, 64, 83] mainly utilize an ID-based Partitioner [69] to evenly partition vector data across computation nodes for index construction and aggregate query results from all partitions during queries. Although this approach is simple to implement, it leads to queries being performed across the entire dataset, resulting in decreased efficiency. Furthermore, in streaming vector searches, the distribution of data and query streams changes continuously over time. For instance, search patterns on social media vary throughout the day and can shift multiple times within hours or even minutes [72]. Log analysis and virus detection, which involve fast-changing data, may experience distribution changes within minutes or seconds [94]. As a result, these rapid data shifts can significantly cause the load imbalance of existing vector systems [10, 12, 15–17, 64, 83] due to their static partitioning strategies. And the imbalance becomes more severe as the rate of change in data distribution increases. To address this, we develop a new dynamic partitioner based on Local-Sensitive Hashing (LSH) [63] and the space filling curves. It assigns neighboring vectors to the same partition and non-neighboring vectors to different partitions. During queries, only a few partitions near the query vector are queried, thereby avoiding queries across all partitions. More importantly, to adapt to real-time data distribution changes, we further devise a method to dynamically adjust partition boundaries based on the workload in each partition, achieving real-time load balancing.

Challenge II: How to support multi-type hybrid storage to improve the system's scalability? Existing vector systems [10, 12, 15–17, 64, 83] typically support a singular type of storage, such as main memory or disk, which led to a gap between storage scalability and read/write efficiency. On the one hand, solely relying on memory often proves insufficient to store the extensive vector data. On the other hand, utilizing disk storage mandates loading data from disk to memory in batches for querying, leading to an obvious increase in latency, particularly for streaming queries. To tackle this challenge, we design a hierarchical storage mechanism considering the varying read-write speeds and storage capacities of different storage media. This hierarchical bottom-up approach encompasses remote disk, local disk, local memory, and state memory, with recent data placed in the upper levels and older data in the lower levels. This arrangement ensures that the recently accessed data crucial for streaming vector search primarily resides in the state memory and local memory of the upper levels, enhancing search speed. Moreover, to alleviate the pressure on local and remote disks, we design a new compression mechanism based on Gorilla algorithm [75] for vectors that leverages their proximity within the same partition.

Challenge III: How to realize fast response in high-speed vector data streams to improve efficiency? Streaming vector search imposes stringent requirements on query latency, especially in instant search and online recommendation applications, where a millisecond-level response time is anticipated [62]. Despite the utilization of indexing techniques, conducting large-scale vector searches in existing vector systems [10, 12, 15–17, 64, 83] involves segmenting the substantial volume of data into blocks for sequential querying, hindering their ability to meet the low-latency requirements. On the other hand, in the streaming vector search scenarios, continuous queries demonstrate clustering characteristics in the vector space, resulting in frequent retrieval of hot data. For example, search engines exhibit search trends during different periods, leading to repeated searches of highly correlated vectors. Motivated by this, we propose dynamic hot-cold data separation based on query frequency and vector insertion time. This mechanism segregates the data segments of vectors and indexes into hot and cold segments, facilitating the swift promotion of hot data to the forefront of the search queue dynamically. This dynamic adjustment enables reducing the query latency during the search process.

To address the above challenges, we develop an effective and efficient streaming vector search engine named **VStream**. Overall, we make the following main contributions.

- VStream stands as the first streaming processing engine for streaming vector search tasks (Sec. 2).
- VStream proposes a dynamic partitioner to dynamically partition vector streams to achieve real-time load balancing (Sec. 3).
- VStream designs a hierarchical storage mechanism, comprising state memory, local memory, local disk, and remote disk (Sec. 4).
- VStream provides two storage optimizers: vector compression and hot-cold separation. The former utilizes vector proximity for compression, while the latter adjusts vector search order based on hot-cold patterns (Sec. 5).
- An experimental study shows VStream's superiority in efficiency, scalability, and effectiveness. Moreover, VStream demonstrates substantial advantages in system designs, including CPU usage, memory usage, and disk throughput (Sec. 6).

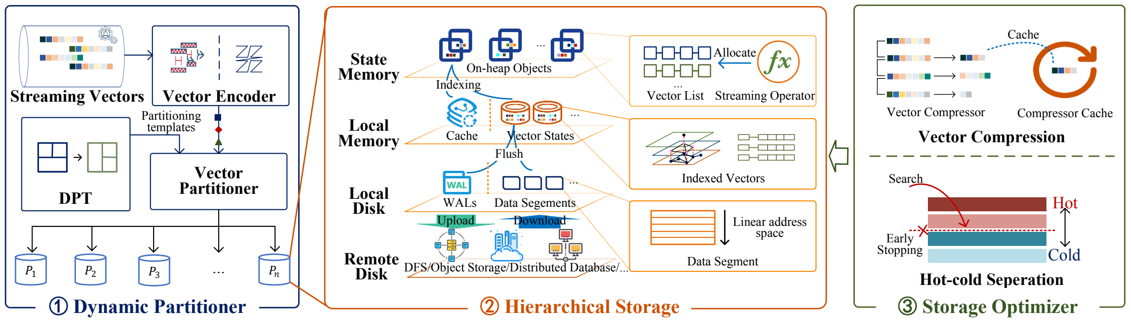


Figure 2: The Architecture of VStream

2 SYSTEM OVERVIEW

As depicted in Fig. 2, VStream comprises three main components: dynamic partitioner, hierarchical storage, and storage optimizer. Streaming vectors are partitioned by the dynamic partitioner and stored in hierarchical storage. The storage optimizer enhances the storage structure to further improve vector read and write speed.

① **Dynamic Partitioner.** It partitions incoming vector data from the upstream messaging system based on vector proximity and adjusts to dynamic data distribution shifts. As shown in Fig. 2, initially, vectors are encoded in the **Vector Encoder** based on LSH [63] and space filling curves, e.g., z-order curve [35] and Hilbert curve [52], then flowed into the **Vector Partitioner**. It partitions vectors according to their distribution on the space filling curves. And **Dynamic Partitioning Templates (DPT)** maintain and continuously update the partitions based on the workload in each partition, providing the real-time partitioning templates (see Sec. 4.2) to the vector partitioner. Utilizing LSH and the space filling curves, neighboring vectors are grouped within the same partition, while non-neighboring vectors are separated into different partitions. The query vector only enters limited nearby partitions, skipping distant partitions to improve search efficiency. The dynamic adjustment strategies in the DPT enable adaptation to changes in data distribution, thereby achieving real-time load balancing.

② **Hierarchical Storage.** It stores all vector data and serves as the state backend of streaming operators for vector search. As shown in Fig. 2, it encompasses four storage levels: **State Memory**, **Local Memory**, **Local Disk**, and **Remote Disk**, thereby ensuring the scalability of the system. Specifically, i) state memory is allocated by the streaming operator instance and thus can be read into on-heap memory for fast computation; ii) local memory is located in off-heap memory on the same physical node. The vector list in state memory is inserted into vector indexes, and then stored as vector states. And there is also a cache in local memory to accelerate the I/O of local disk. When I/O occurs on the disk, it is necessary to access the cache; iii) when the number of states in local memory reaches its limit, they are flushed to local disk after being first written to the write-ahead logs (WALs), and stored in the format of data segment (see Sec. 5.2), the fundamental structure of the vector storage in a linear address space provided by VStream; iv) when the number of data segments in the local disk reaches its limit, they will be uploaded to remote disks such as DFS, object storage, or distributed databases, and can be downloaded when needed.

③ **Storage Optimizer.** It optimizes the storage structure of the aforementioned hierarchical storage. As illustrated in Fig. 2, VStream incorporates two types of optimizers: **Vector Compression** and **Hot-cold Separation**. The vector compression optimizer extends the Gorilla algorithm to compress vectors, which are later decompressed during vector searches. On the other hand, the hot-cold separation optimizer assigns hot-cold scores to data segments in the hierarchical storage based on features such as the access frequency of vector data.

3 DYNAMIC PARTITIONER

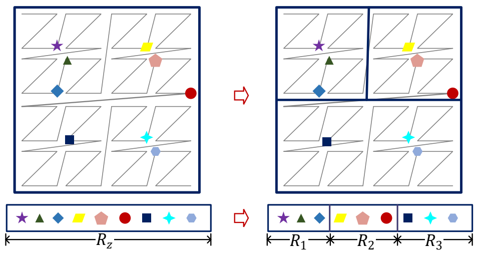
Behind Idea. Existing vector search systems [10, 12, 15–17, 64, 83] typically use the IDPartitioner [69], which hashes each vector and assigns them across partitions using a round-robin method. Therefore, VStream partitions vectors based on LSH [63] and space filling curves, and places neighboring vectors in the same partition and distant ones in different partitions. Then each query vector is assigned to limited nearby partitions for querying, resulting in improved search efficiency.

3.1 Vector Encoder

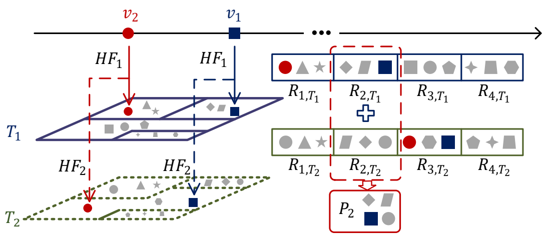
VStream initially provides a vector encoder based on LSH and space filling curves. This encoder transforms high-dimensional vectors into low-dimensional hash value vectors, and subsequently converts them into one-dimensional encoded values, which maintain their positional characteristics in the vector space, thereby enabling assigning neighboring vectors to the same partition.

Vector Hashing. Firstly, the vector encoder encode vectors using LSH, a technique commonly used to search nearest neighbors. Its main idea is to map similar vector to the same hash bucket, enabling quick searches for similar vectors within these buckets without needing to search the entire dataset comprehensively. Specifically, the vector encoder employs a hash family HF with d hash functions, which encode each vector v into a d-dimensional encoding $\hat{v}$ consisting of d hash values, i.e., $\hat{v} = HF(v) \in \mathbb{R}^d$. Under this encoding scheme, vectors that are close to each other will also have close proximity in this d-dimensional space.

Space Filing Curve Encoding. Since the aforementioned vector hashing maps vectors to a d-dimensional space, partitioning unbalancedly distributed data within this space remains challenging. Therefore, we utilize a d-dimensional space filing curve to



(a) Partitioning Templates



(b) Vector Partitioning

Figure 3: Vector Partitioner

further encode the hash values into a 1-dimensional encoded values. VStream supports various space-filling curves, including the z-order curve [35], Hilbert curve [52], Peano curve [74], and Sierpiński curve [76]. Here, the z-order [35] curve is presented as an example. Specifically, for any vector v hashed into $\hat{v}$ by vector hashing, the encoding function $\mathcal{Z}$ provided by z-value encoding converts it into an integer $\tilde{v}$, i.e., $\tilde{v} = \mathcal{Z}(\hat{v}) \in \mathbb{Z}$. The range of z-values is denoted as R_z, i.e., $\tilde{v} \in R_z$. The z-values preserves the proximity of nearby vectors in 1-dimensional space and facilitates balanced partitioning.

3.2 Vector Partitioner

Based on vectors encoded by the vector encoder, VStream provides a vector partitioner to achieve balanced vector partitioning. The partitioner initially segments the space filling curve based on the current distribution of vectors, creating a partitioning template.

Partitioning Templates. Through the aforementioned vector encoding process, vectors are projected onto a space filling curve and organized based on their encoded values. Consequently, leveraging the current count of incoming vectors N and the desired number of partitions p, the vector partitioner sequentially splits the z-order curve into p curve segments, with each curve segment containing $\lfloor \frac{N}{p} \rfloor$ vectors. The segmented z-order curve is referred to as partitioning templates $T = \{R_1, ...R_i, ..., R_p\}$, where $R_i \in R_z$ & $\bigcup_1^p R_i = R_z$. Taking Fig. 3(a) as an example, the left 9 vectors on the z-order curve are sequentially divided into 3 parts, forming the partitioning templates shown on the right. The original range R_z of z-values is divided into R_1, R_2, and R_3.

Vector Partitioning. Upon the arrival of a new vector v, it undergoes vector encoding $\tilde{v}$, and then it is assigned to the space filling curve segment R_i to which $\tilde{v}$ belongs, i.e., $\tilde{v} \in R_i$. Thus, each partition contains balanced vectors, and incoming query vectors only need to be assigned to the specified partition based on partitioning templates, effectively filtering out irrelevant partitions.

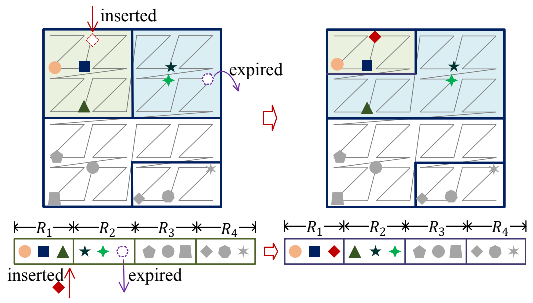


Figure 4: Dynamic Partitioning Templates

While in the skipped partitions, there may still exist a few vectors that are close to the query vector. Therefore, the vector partitioner utilize multiple hash families, resulting in multiple partitioning templates. Each vector can be assigned to multiple curve segments based on different templates. Then the segmented sets of vectors from all templates are sequentially unioned to form a partition. By using multiple hash families, most neighboring vectors are grouped into the same partition, thereby reducing the missing query results. Taking Fig. 3(b) as an example, the vector partitioner employs two hash families, HF_1 and HF_2 to create two partitioning templates, T_1 and T_2. The incoming vectors v_1 and v_2 enter different positions of the two templates. After sequentially merging templates T_1 and T_2, partition P_2 contains 4 vectors, which more comprehensively include neighboring vectors compared to R_{2,T_1} and R_{2,T_2}.

3.3 Dynamic Partitioning Templates

Although the vector partitioner enable balanced partitioning of vectors, the shift of streaming vectors distribution can rapidly cause the imbalance of partitions. To address this issue, VStream provides dynamic partitioning templates which adjust the boundaries of partitions based on the number of vectors in each partition. Specifically, for any subset R of the templates T, i.e., $R \in T$, if the current number of vectors exceeds the average number $\lfloor \frac{N}{p} \rfloor$, we decrease its right boundary until the number of vectors is reduced to $\lfloor \frac{N}{p} \rfloor$ or less. Conversely, if the number of vectors is below this average, we expand its right boundary. Taking Fig. 4 as an example, in the initially balanced templates, a red vector is inserted into R_1 while a purple vector in R_2 expires. Consequently, the templates shift from left to right: R_1's range shrinks, and R_2's expands. However, both R_1 and R_2 maintain an average number of vectors.

4 HIERARCHICAL STORAGE

Behind Idea. Existing vector search systems [10, 12, 15–17, 64, 83] often face a trade-off between system efficiency and scalability. Performing vector indexing and searching in memory ensures efficiency but cannot store large-scale data, whereas local/remote disk storage offers the opposite. Therefore, VStream is fortified by a hierarchical storage mechanism leveraging memory to store a small amount of more recent data. Simultaneously, it utilizes local and remote disks for the storage of older data, implementing an automatic flushing mechanism from the top tier down.

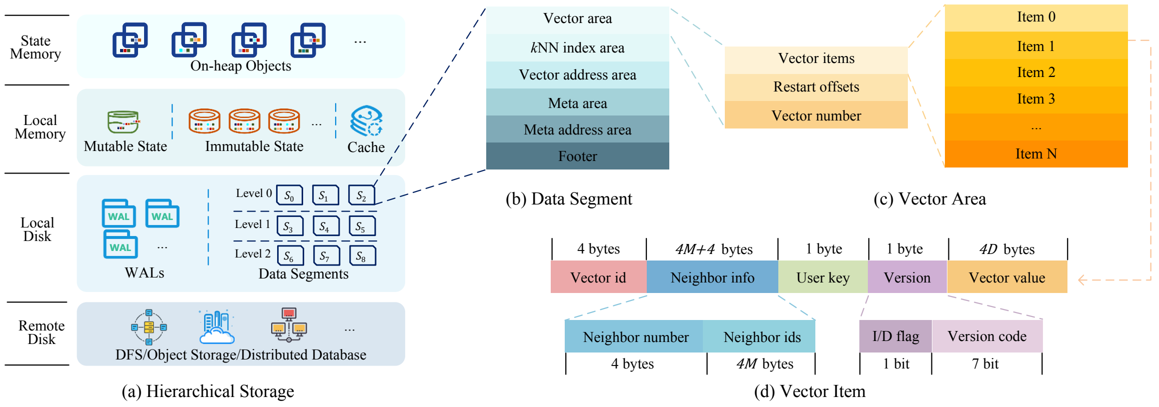


Figure 5: Hierarchical Storage

4.1 Data Organization

As depicted in Fig. 5 (a), the hierarchical storage comprises state memory, local memory, local disk, and remote disk.

State Memory. VStream establishes its first-level state storage, referred to as state memory, directly on the on-heap memory of the SPE. This eliminates the need for data serialization. Given the limited capacity of on-heap memory, VStream stores only a few recent vectors as on-heap objects.

Local Memory. VStream's second-level storage is built upon the off-heap memory located in the same node of state memory and is referred to as local memory. When the number of vectors in the state memory reaches its limit, they are indexed into a fixed-size vector index. These vector indexes, along with their corresponding vector data, are referred to as vector states. When a vector state reaches its size limit, a new state is created for storage. States that reach their capacity limit immediately become read-only and are known as immutable states, while those that have not reached their limit, and still accept new vector insertions, are referred to as mutable states. When the number of immutable states reaches its limit, they are flushed to the next layer of local disk. This ensures that a certain quantity of recent vectors remains in memory for rapid vector searches. Additionally, a cache is also implemented in local memory to accelerate local disk I/O.

Local Disk. VStream's third-level storage, known as local disk, primarily stores blocked vector data in the linear address disk space. Immutable states from the local memory are asynchronously flushed to various data blocks, referred to as data segments. The data segment is a disk structure specifically designed by VStream to facilitate quick access to vectors. Therefore, each data segment stores a specific quantity of vectors along with their corresponding index data and preserves metadata associated with these vectors. The structure of data segments is detailed in the subsequent subsection. Note that the data segments are organized in levels, which facilitates concurrent search of data segments at each level (see Sec. 4.4) and also support early stopping in the hot-cold separation optimizer (see Sec. 5.2). Additionally, the local disk also includes WALs to ensure the success of vector writes.

Remote Disk. VStream's bottommost-level storage comprises a scalable distributed file system, object storage, or distributed database. This level typically involves distributed disk nodes with substantial storage capacity but slower access speeds.

4.2 Data Segment

VStream offers the data segment as the fundamental storage structure in both the local disk and the remote disk. As shown in Fig. 5 (b), it encompasses the vector area, *k*NN index area, vector address area, meta area, meta address area, and footer. These areas are arranged in ascending order based on their storage addresses.

Vector area. It is located at the beginning of the segment and is responsible for housing all vector data within the current segment. It comprises a fixed number of *vector items* specified by the user, detailed in the subsequent subsection.

***k*NN index area.** It is located adjacent to the vector area and houses the vector index (e.g., HNSW [68], IVF_PQ [46], etc.). The range of the index data corresponds to all vector data within the current segment. Taking HNSW as an example, this area stores the serialized objects of the hierarchical neighbor graph.

Vector address area. After the *k*NN index area comes the vector address area, which serves as the addressing data for the vector area. This area is used to determine the starting address of each vector item. It consists of a sorted list of triplets, with each triplet corresponding to a vector item. Each triplet contains the largest internal ID in the vector area, the offset of the vector area in the data segment, and the length. When seeking the vector value corresponding to a specific internal ID, a binary search is conducted in the vector address area to locate the relevant vector item.

Meta area. The meta area houses the metadata of the vectors and indexes, encompassing details such as the total number of vectors in the current segment, index size, index type, index parameters, etc. The metadata is organized sequentially as key-value pairs to ensure the absence of duplicate keys.

Meta address area. In this area, the offsets of each key within the meta area are stored in the form of key-value pairs.

Footer. The footer stores the starting addresses of all areas above.

4.3 Vector Area

We proceed to introduce the vector area, which contains information such as the ID, version, and specific numerical values of all vectors within the current segment. The vectors are arranged in storage addresses in ascending order based on their IDs. At the end of the vector area, we record the current number of vectors and the starting offsets of each vector to facilitate quick random access. As illustrated in Fig. 5 (c), the vector area comprises vector items, restart offsets, and vector numbers arranged from top to bottom.

Vector items. As shown in Fig. 5 (d), each vector item is composed of five fields: vector ID, neighbor info (optional), user key, version, and vector value. The vector ID is a unique identifier assigned during disk storage, with a fixed length of 4 bytes, ensuring its progressive incrementation across the vector area. The neighbor info is an optional field that allows users to customize the storage of certain neighbor IDs for the current vector, thereby enhancing vector searches. The neighbor info field has a user defined length, $4M + 4$, where the first 4 bytes indicate the number M of stored neighbors, followed by $4M$ bytes representing the neighbor IDs. The user key is a 1-byte field specified by the user, used to store data IDs from the actual application scenario, such as product IDs. The version field is 1 byte long and the first bit indicates if the vector is deleted (0 for deleted, 1 for inserted), while the remaining 7 bits represent the vector's version. Lastly, the vector value, with a length of $4D$ bytes where D is the vector dimension, stores the data for each dimension of the current vector in sequential order.

Restart offsets. Restart offsets are a list of offsets, where each offset corresponds to a vector item within the vector area. The length of this list is equal to the total number of current vectors. Each offset, occupying one byte, indicates the offset of its respective vector item. When conducting random access of a vector based on its ID, it is essential to locate the starting address of that vector.

Vector number. It stores the total number of vectors in the current vector area, with a length of 4 bytes.

4.4 Search Procedure

Fig. 6 illustrates the vector search process on hierarchical storage.

STEP ①: Search in State Memory. VStream resides in on-heap memory and maintains a priority queue of length k to store and update the search results for the current partition. It also initiates the threads to sequentially search the vectors in the state memory and immediately update the result queue.

STEP ②: Search in Local Memory. VStream maintains an internal queue for query results in the local memory. Then VStream initiates several threads to search all states in memory and update the internal queue concurrently.

STEP ③: Search in Local Disk. VStream searches the data segments in the local disk level by level, such as level 0 and level 1 in Fig. 6. At each level, each segment is searched concurrently with multiple threads, such as S_0 to S_2 in Fig. 6. For each segment, its kNN index area is loaded into memory before searching. When accessing a vector with a certain ID, it is first accessed in the cache. If it is not hit, the vector area of the segment is accessed.

STEP ④: Search in Remote Disk. If there is data stored on the remote disk, it needs to be loaded into the local disk in the format of data segments. Then, the same search steps as STEP ③ are executed, and the results are added into the internal queue.

STEP ⑤: Queue Merge. Insert the results of internal queue into the result queue to complete the merging of the two queues.

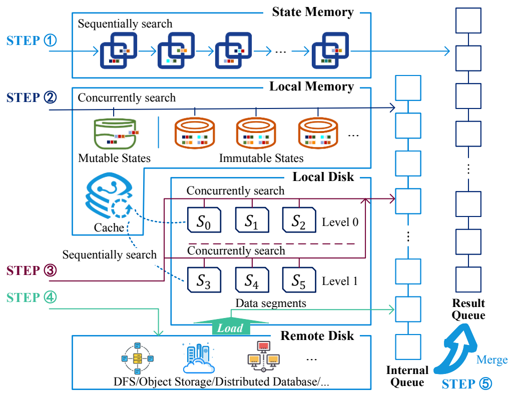


Figure 6: Search Procedure in the Hierarchical Storage

4.5 Consistency Mechanism

The consistency of a streaming processing engine (SPE) can be defined as the alignment between the internal state of the system and the external output data before and after recovery from a failover [19]. VStream adheres to this definition. However, ensuring consistency is more challenging in streaming systems compared to batch vector search systems, where the input is deterministic, and data consistency is guaranteed through the atomicity of computations. To avoid compromising the efficiency of vector search, VStream employs a periodic backup mechanism.

Specifically, VStream backs up data segments using a periodic checkpoint mechanism [29] common in streaming processing engines (SPEs). By periodically inserting checkpoint markers into the source, checkpoints can be triggered across all operator instances. Each operator instance then packages its state into a consistent snapshot, based on the Chandy-Lamport algorithm [31], upon receiving the marker. However, the changes in vector state between successive checkpoints are typically small, making it unnecessary to take snapshots of all vector states. Instead of copying existing segments to persistent storage, VStream allows the new checkpoint to reference previous data from the historical checkpoint.

5 STORAGE OPTIMIZER

Besides, VStream offers two types of storage optimizers, i.e., vector compression and hot-cold separation.

5.1 Vector Compression

Behind Idea. When handling large-scale vector and index data, it is natural to consider compression techniques. In scenarios where vectors exhibit proximity within a partition, the Gorilla algorithm [75] proves to be particularly effective. Although numerous vector compression methods are currently available, including families of vector quantization techniques (such as PQ [56], OPQ [46], RQ [92], LSQ [42]), scalar quantization [70], and more recent approaches like orthogonal projection [93], these are lossy compression methods.

5.1.1 Vector Compressor. The Gorilla algorithm efficiently compresses individual floating point numbers in a sequential manner, transforming initially fixed-length floating point formats into

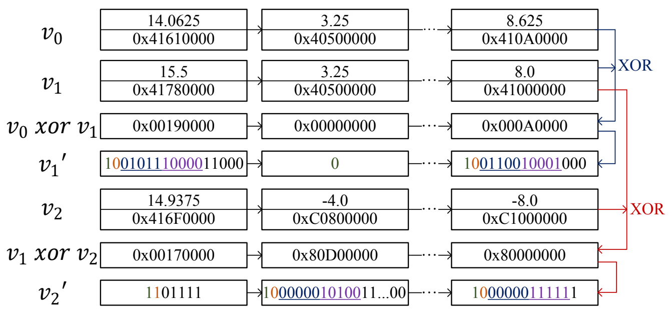


Figure 7: Vector Compression

shorter variable-length formats. However, it cannot be directly used for compressing high-dimensional vectors due to the lack of proximity between the floating point numbers within vectors, rendering the Gorilla algorithm ineffective for compression. Therefore, the vector compressor in VStream converts vectors into variable-length formats, where all floating point numbers across every dimension of a vector are stored sequentially in a continuous storage space.

During the compression, the original data of the first vector in the vector area is directly preserved for storage. Subsequently, an XOR operation is performed between each subsequent vector and the previous one. Fig. 7 shows an example of vector compression in the Float32 format under the IEEE 754 standard. The initial vector $v_0 = \langle 14.0625, 3.25, ..., 8.625\rangle$ is stored with its original value, and the subsequent vectors $v_1 = \langle 15.5, 3.25, ..., 8.0\rangle$ and $v_2 = \langle 14.9375, -4.0, ..., -8.0\rangle$ are compressed after being XORed with v_0 and v_1 respectively. The XOR result of 15.5 and 14.0625 is 0x00190000, which has 11 leading zeros and 16 trailing zeros, and the valid bits are 0b11000. The final storage result is 10010111000011000, compressing 32 bits to 17 bits. And, 8.0, -4.0, and -8.0 are also compressed similarly, with non-zero XOR results and no shared zeros. The XOR result of 3.25 is 0, so only 1 bit is stored. The XOR result of 14.9375 has the same number of leading and trailing zeros as 15.5, so the number of zeros does not need to be stored.

5.1.2 *Compressor Cache.*

Once the Gorilla algorithm compresses the vectors, it cannot support random access. Each time a vector is accessed, it must be decoded starting from the first vector, resulting in a significant decrease in the access speed of the original vector. Although data segments utilize memory caches to expedite access, they store the compressed values of the vectors, necessitating recursive access and decoding with previous vectors. As a result, we have opted to cache the decompressed values of the vectors instead. In this way, for cache-missed vectors, it is possible to recursively locate the nearest cached vector and start the decoding from it, bypassing the need to start from the first vector.

5.2 Hot-Cold Separation

Behind Idea. The design motivation behind this optimization is to run search operations in memory. Despite existing replacement algorithms [79] based on access frequency, like LRU, OPT, FIFO, and CLOCK used in buffer management, they encounter several challenges when applied to vector searches: (i) They primarily manage data based on access frequency, neglecting the proximity of vectors during the search process; (ii) They do not influence the search order of data segments during vector searches; (iii) They cannot avoid searching all data in extreme cases. To address these challenges, VStream's hot-cold separation integrates multiple metrics associated with vector proximity during search processes.

5.2.1 *Hot-Cold Metric.*

VStream provides hot-cold metrics for immutable states and data segments. That is, a higher hot-cold score indicates a greater likelihood of access. VStream establishes the following four hot-cold metrics.

Access Frequency. Assuming the access frequency of a data segment is *AF*, each time when this data segment is accessed, its *AF* increases by 1. With an initial setting of *AF = 0*, the calculation of the hot-cold score H_A based on access frequency is:

$$H_A(t) = (1 - d_t)(1 - e^{-\gamma AF(t)}) + d_t H_A(t') \tag{1}$$

Here, t represents the current system time, t' represents the previous update time, and $t, t' > t_0$, where t_0 is the initial time of the system. When $t = t_0$, $H_A(t) = 0$. $d_t = e^{-\eta(t-t')}$ represents the weight of *AF* with time decay, where η is the decay coefficient, and γ is the exponential smoothing coefficient used for *AF*.

Search Hits. Similarly, VStream can update the hot-cold score based on the number of hits in the data segment from queries. Assume that the search hits of a data segment are denoted as *SH*, each time a query result originates from this segment, *SH* increases by 1. With an initial setting of *SH* = 0, the calculation of the hot-cold score H_S closely resembles Equation 1.

$$H_S(t) = (1 - d_t)(1 - e^{-\gamma SH(t)}) + d_t H_S(t') \tag{2}$$

Search Contribution. The contribution from search hits also serves as a metric for assessing the hot-cold score of a data segment. We employ the distance between the query results from the segment and the query vector as a measure of search contribution and subsequently establish the hot-cold score H_C based on this contribution. The calculation of H_C is as follows.

$$H_C(t) = (1 - d_t)\frac{1}{|R|}\sum_{v \in R} e^{-\eta d(v,q)} + d_t H_C(t') \tag{3}$$

Here, R is the search result set, and q denotes the query vector.

Freshness. The freshness of a data segment is another factor influencing its hot-cold score. We randomly sample vectors within the segment to measure the segment's hot-cold score. The calculation of the hot-cold score H_F based on freshness is as follows.

$$H_F(t) = \frac{1}{|S|}\sum_{v \in S}(1 - \frac{t - v.t}{t - t_0}) \tag{4}$$

Here, S is the sampling subset of the data segment, and the timestamp of the vector $v \in S$ can be obtained from the version code, which is mentioned in Sec. 4.3.

Overall, via a linear combination manner [86], we define a hot-cold metric using four parameters ω_A, ω_S, ω_C, and ω_F.

$$H = \omega_A H_A + \omega_S H_S + \omega_C H_C + \omega_F H_F \tag{5}$$

5.2.2 *Segment Search Steps.*

VStream sorts the immutable states and data segments based on their hot-cold scores. In that case, segments are no longer being arranged by their generation time but rather by their hot-cold scores. The process of utilizing hot-cold scores for search is described as follows.

Table 1: Statistics of the Datasets Used

Attributes	SIFT1B	Deep1B	Tweets
# vectors	1×10^9	1×10^9	4.75×10^8
vector dimension	128	96	128
Value type	Int	Float	Float
Disk Usage	559.0 GB	1072.3 GB	700.8 GB

a) Search with Hot-cold Metric. First, a priority queue with a length of k is maintained to store the search results. Then, multiple threads are utilized to search through all immutable states, and the results found in each immutable state are asynchronously inserted into the priority queue. Once the search for immutable states is completed, the data segments at each level are searched in parallel using multiple threads, and the priority queue is updated accordingly. This approach ensures that segments with higher hot-cold scores are prioritized in the search process.

b) Early Stopping. As a result of query clustering, the search results consistently occur in data segments with higher hot-cold scores and occur less frequently in those with lower scores. Hence, it is helpful to stop the search early while traversing through each level of segments to filter out the ones with lower hot-cold scores. To facilitate this, VStream offers an early stopping strategy called the "threshold-stop" strategy. If the distance from the last element in the queue to the query vector is less than a certain threshold, the search is stopped early. The threshold is updated through a weighted average with the tail of the queue, where a higher weight θ at the tail indicates an earlier early stopping. Furthermore, VStream also allows users to customize their early stopping strategies.

6 EXPERIMENTS

6.1 Experimental Setup

6.1.1 Hardware and Software. The experimental environment is deployed on a cluster consisting of 10 physical nodes. Each node is equipped with 2 CPUs (Intel Xeon E5-2620 v3, 2.40 GHz), each containing 12 cores, 128 GB of memory, and a 1 TB hard drive. These nodes run CentOS 7.9 and are interconnected via a 2 Gbps/vcore network switch. Apache Flink 1.18.0 is employed as the computing environment for the cluster. HDFS 2.7.1 is used for storing the raw dataset. For vector storage, VStream uses RocksDB 8.9.1 as the base for the state backend, while the baseline methods utilize Milvus 2.3.x, Chroma DB 0.5.12, and Qdrant 1.12.1 for storage.

6.1.2 Configurations. We give configurations for Flink, RocksDB, HDFS, Milvus, Chroma DB, Qdrant, and VStream, respectively.

Flink. Following Flink's recommended configuration guidelines [7], we deploy Flink with 10 task managers in standalone mode. Each task manager runs with a 16 GB on-heap memory.

RocksDB. We rewrite the Java interface of the RocksDB state backend to support vector search functionality in RocksDB instances. In addition, the RocksDB instances utilize async I/O and direct I/O for flush operations to speed up disk access.

HDFS. The HDFS is deployed with default configurations, with a heap size of 1 GB allocated for each Hadoop daemon.

Milvus. We deploy Milvus on a Kubernetes cluster with all 10 physical nodes. The CPU and memory limit of each Milvus node is configured according to the official Milvus sizing tool [9]. To emulate the streaming processing environment, we start a Flink job on a separate Flink cluster that continuously reads vector data and sends requests to the Milvus proxy.

Chroma DB. Due to Chroma DB's lack of cluster deployment support, we start a standalone Chroma DB instance for each Flink subtask locally. Each Flink subtask receives partitioned vector streams, and forwards them to its dedicated Chroma DB instance.

Qdrant. We deploy Qdrant on the same Kubernetes cluster as Milvus with identical CPU and memory constraints. The Flink job connects to the Qdrant cluster via the official Qdrant Java API.

VStream. We deploy VStream on the cluster based on the Flink's standalone mode. Each node of VStream is allocated 32 GB of memory and 512 GB of disk storage. VStream accepts input vector streams from Flink and stores the vectors in hierarchical storage.

6.1.3 Datasets. We use two popular open-source vector datasets, SIFT [4] and Deep [2], as well as a Tweets dataset transformed from the Twitter dataset [14] using FastText [24]. Table 1 summarizes the statistics of the datasets used, where SIFT and Deep are benchmark datasets of vector search, both containing 1 billion vectors extracted from images, and Twitter dataset includes 475 million Twitter posts collected in 2009. Note that the disk usage of the dataset refers to the disk usage by the vectors after the dataset is decompressed and stored in HDFS as text, with each vector stored in lines.

6.1.4 Workload. We read static vector data from HDFS using Flink and use Flink's DataGenerator to construct simulated input streams at different data rates, i.e., $(0.5k, 3.5k, 6.5k, 9.5k)/s$, with a query/insert/deletion ratio of $1 : 3 : 1$. The data rates refers to the number of vectors flowing into these systems per second.

6.1.5 Baselines. We compare VStream with the commonly used vector systems, Milvus [49, 83], ChromaDB [3], and Qdrant [12] according to the DB Engins Ranking [5]. To evaluate the performance of Milvus, Chroma DB, and Qdrant, we utilized an existing Flink connector [8] or implemented it for the above systems. The commonly used IDPartitioner [69] was employed as the partitioner. Additionally, in the ablation experiment, we compared other variants of VStream: VStream without the dynamic partitioner (VStream w/o DP), VStream without vector compression (VStream w/o VC), and VStream without hot-cold separation (VStream w/o HCS). Finally, we also evaluate different partitioners, including VStream with k-Means, VStream with Odyssey, VStream with the Hilbert curve, and VStream w/o LSH.

6.1.6 Evaluation Metrics. We use six metrics. i) **Query Latency.** It denotes the time interval between a query's creation timestamp and the timestamp of the final result; ii) **Memory usage.** It denotes the memory usage of the cluster; iii) **Disk throughput.** It denotes the average disk throughput for read and write operations across all nodes; iv) **CPU usage.** It denotes the average CPU occupancy rate of cluster nodes; v) **Disk usage.** It represents the used disk capacity of the cluster; vi) **Recall.** It denotes the recall for each query. In the streaming vector search process, the size of vectors and indexes increases as the data flows in. Thus, we report the memory usage and disk usage at the end of the stream, as well as the other metrics based on the average of the last 1000 queries.

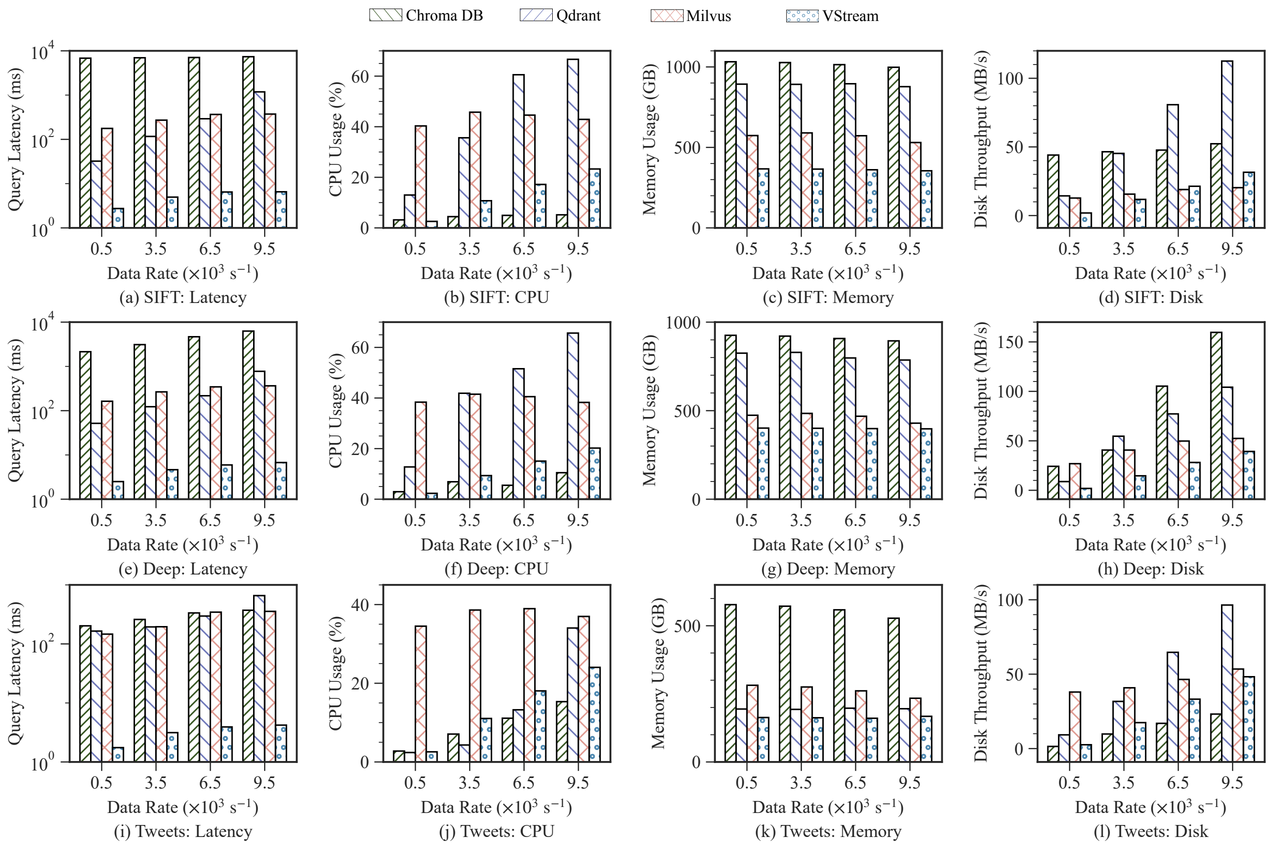


Figure 8: Efficiency and Scalability Evaluation vs. Data Rate

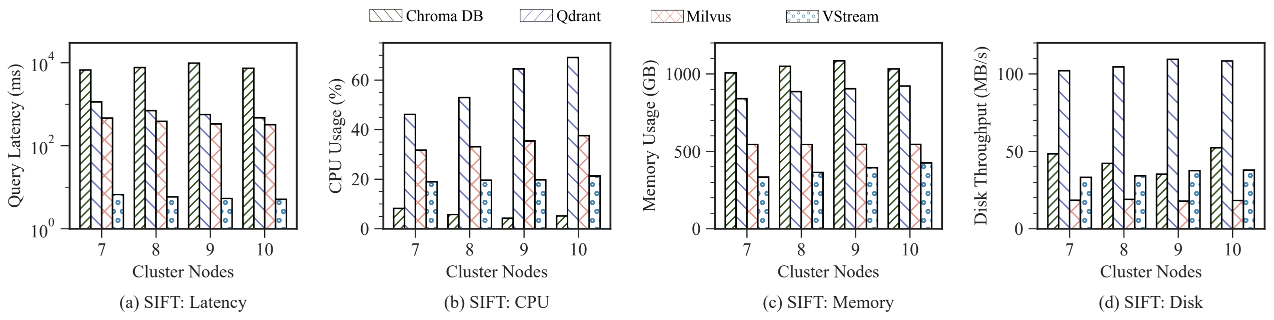


Figure 9: Efficiency and Scalability Evaluation vs. Cluster Nodes

6.2 System Efficiency and Scalability Study

To evaluate the efficiency and scalability of VStream, we conduct experiments at varying data rates, reporting their respective query latencies, and resource usage including memory, disk, and CPU. The experimental results on three datasets are shown in Fig. 8(a)–8(d), Fig. 8(e)–8(h), and Fig. 8(i)–8(l), respectively.

VStream performs better than Milvus, ChromaDB, and Qdrant, especially at higher data rates, e.g., $9.5 \times 10^3/s$, achieving 115–1127×, 33%–68%, and 9%–75% reduction in query latency, memory usage, and disk throughput, respectively. Although VStream's CPU usage is slightly higher than that of Chroma DB (which employs a single-threaded write approach on each instance), it is 29%—69% lower than that of Qdrant and Milvus. This is because, on one hand, VStream utilizes a dynamic partitioner to reduce the search space. On the other hand, the hierarchical storage and hot-cold separation optimization enable most searches in local memory of each node, thereby reducing disk I/O. In contrast, Milvus requires searching across all data partitions and involves frequent cross-node data transmission, requiring frequent disk accesses, which results in higher query latency, CPU usage, and disk throughput.

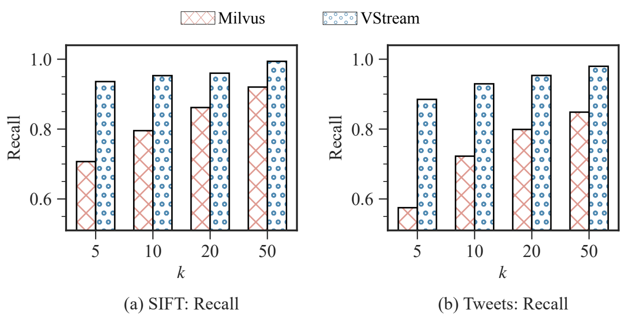


Figure 10: Effectiveness Evaluation vs. k

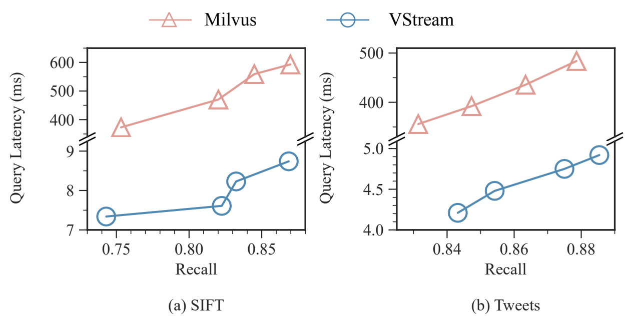


Figure 11: Latency-recall Curve

Chroma DB, being an embedded database, lacks an efficient data routing or partitioning mechanism for managing large-scale data in a distributed cluster and relies on frequent disk I/O operations when processing high-speed data updates.

We proceed to conduct experiments on Milvus, Chroma DB, Qdrant and VStream at various cluster nodes. The experimental results on the SIFT1B dataset are shown in Fig. 9. The results on the other two datasets are omitted due to similar observations and limited space. First, the increase in cluster nodes allows for more CPU cores to be utilized for data insertion and querying, resulting in reduced query latency and resource usage. Second, VStream consistently outperforms Chroma DB, Qdrant, and Milvus regarding query latency and memory usage across different cluster nodes. Additionally, VStream's superiority on query latency becomes even more pronounced at higher cluster nodes due to its ability to avoid accessing data across nodes. While increasing the cluster nodes implies that Milvus must access data from a larger number of data nodes. Finally, VStream exhibits higher disk throughput compared to Milvus. This is because, during small number of cluster nodes, VStream has idle nodes without data, whereas Milvus utilizes all available physical nodes.

6.3 System Effectiveness Evaluation

We perform vector recall experiments on Milvus and VStream at a variety of k to evaluate their effectiveness. The recall on the SIFT1B and Tweets datasets are presented in Fig. 10(a) and Fig. 10(b), respectively. First, Milvus requires caching and batch inserting data, which prevents it from searching for data in the uncompleted batch, resulting in a lower recall compared to VStream. As shown in Fig. 10, VStream achieves superior recall in streaming vector searches. This is because when Milvus receives high-speed incoming data streams from external sources, some real-time data gets stalled in external systems, resulting in delayed vector searches and reduced recall.

We also conducted evaluation experiments on VStream and Milvus at a data rate of $9.5k/s$, measuring the query latency under different recall conditions, and plotted the latency-recall curve. As shown in Fig. 11, both VStream and Milvus require longer query latencies to ensure higher recall. However, VStream's query latency is 50.8×–98.3× shorter than that of Milvus.

6.4 System Ablation Studies

Dynamic partitioner ablation. We compared VStream's performance with VStream with k-Means, VStream with Odyssey, VStream with Hilbert Curve, VStream w/o LSH, and VStream w/o DP at different cluster nodes. Fig. 12(a)–Fig. 12(d) presents the ablation results. VStream achieves a significant reduction in query latency (44%—46%) and disk throughput (10%—22%) compared to VStream with k-Means and VStream with Odyssey. While memory usage is slightly higher than that of k-Means and Odyssey, the CPU usage is nearly identical to k-Means and 12%—16% lower than Odyssey. Experiments using the Hilbert curve show similar results to VStream, indicating that the choice of space-filling curve has minimal impact on the performance of the partitioner. Due to the stronger proximity of vectors on the Hilbert curve, its performance is slightly better. After removing LSH, VStream's query latency, CPU usage, memory usage, and disk throughput increased by 1.40×—1.59×, 41%–64%, 1%—9%, and 63%–79%, respectively.

Storage optimization ablation. To evaluate the effectiveness of VStream's two storage optimizers, we conduct experiments where we removed vector compression and hot-cold separation individually, i.e., VStream w/o VC and VStream w/o HCS. Fig. 13(a)–Fig. 13(d) presents the ablation results. First, VStream has a 55%/39% lower disk throughput/usage compared to VStream w/o VC especially at higher data rates, e.g., $9.5 \times 10^3/s$. And as Fig. 13(d) shows, VStream's vector compression optimizer achieves a 61%–65% of compression ratio at varing data rates. This is achieved through the Gorilla-based vector compression optimizer's disk resource savings. However, enabling the vector compression optimizer slightly increases the CPU usage of VStream by 2%–12%. This optimizer mainly helps in scenarios where disk capacity or disk throughput is limited. VStream also allows users to selectively enable the vector compression optimizer. Secondly, VStream w/o HCS exhibits higher query latency, CPU usage, and disk throughput compared to VStream due to the hot-cold separation optimizer reducing the search space. Lastly, the faster the data flows, the more noticeable the effects of the above two optimizers. This is because higher data rates result in increased pressure on the disk and CPU. Fig. 14 shows the memory and disk usage for three datasets in VStream, with a data rate of $9.5k/s$. As the figure shows, VStream's memory usage accounts for only 6.93%–9.85% of the total storage. And the majority of the data is stored on disk. We have also evaluate the time of VStream for a vector to go from state to local memory and then to

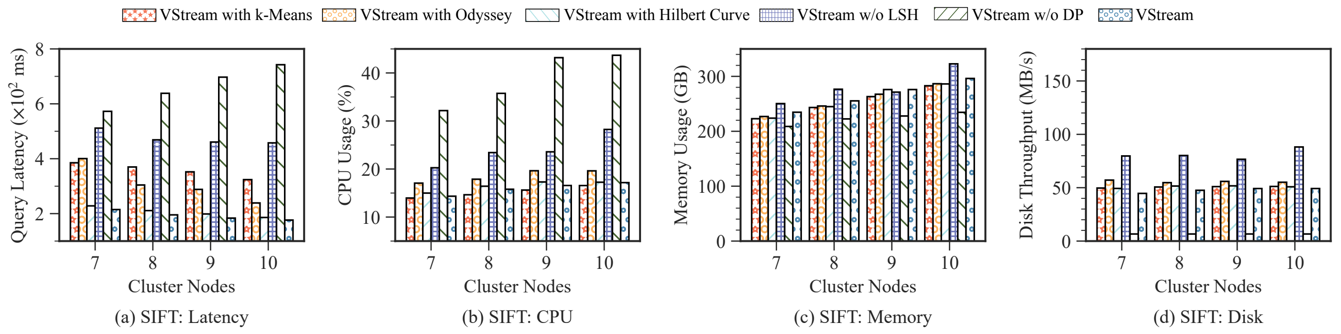


Figure 12: Dynamic Partitioner Ablation vs. Cluster Nodes

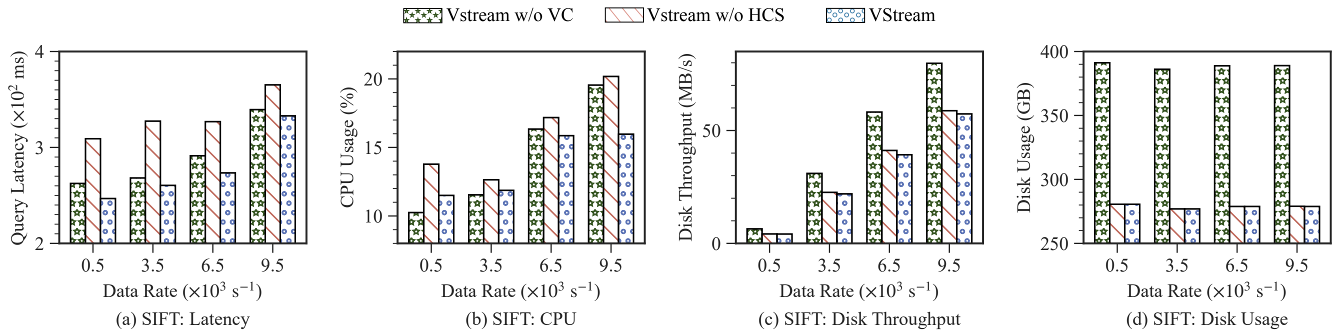


Figure 13: Storage Optimization Ablation vs. Data Rate

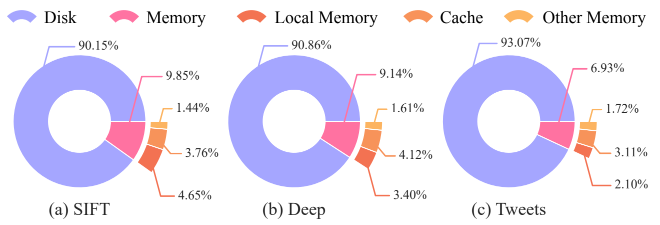


Figure 14: Memory/Disk Uasge of VStream

disk under different data rates. The results for the three datasets are presented in Table 2. As observed, VStream efficiently utilizes both memory and disk to store vectors shortly after receiving the data, rather than storing all the data in memory.

6.5 System Knobs Tuning

To explore the influence of system knobs on system performance and offer some guidance for knob configuration tuning, our experiments emphasize the examination of VStream's performance under varying segment sizes s and hot-cold separation parameter θ.

Tuning of s. Fig. 15(a)–Fig. 15(d) shows the results of tuning segment sizes s. To present the system's performance across various segment sizes, we initially lowered the hot-cold separation threshold θ to 0.5 to ensure that all queries necessitated disk access. So the query latency, CPU usage, and disk throughput are relatively higher. As the segment size s increases, query latency, disk throughput, and CPU usage initially decrease and then rise again. And as the segment sizes s increases, memory usage increases because the mutable and immutable states in local memory become larger. Therefore, it is essential to determine the segment size based on the available memory resources in the user's cluster.

Tuning of θ. As shown in Fig. 15(e)–Fig. 15(h), the query latency of VStream decreases with the increase of hot-cold separation parameter θ. The results show that increasing the value of θ can significantly improve search efficiency. This is because a larger θ value means stopping the search earlier, skipping more cold data, and thus greatly reducing query latency, CPU usage, and memory usage. In practical scenarios using VStream, the determination of the amount of cold data to be skipped should align with specific application requirements and business needs. For instance, in social media platforms characterized by frequent updates, a larger θ value may be preferable. On the other hand, in applications like email or log analysis with lower data update frequencies, a smaller θ value might be more suitable. Moreover, the disk throughput in the figure is less impacted by θ, mainly attributed to the influence of disk cache, which helps maintain relatively stable disk access volumes.

7 RELATED WORK

(A) Vector Similarity Search is widely studied in communities [39, 40]. It can be classified into exact algorithms and approximate algorithms. Specifically, exact similarity search mainly relies on tree-based structures [34, 66, 77, 84, 89, 95] to construct

Table 2: Time from State to Local Memory/Local Disk

Data Rates ($\times 10^3/s$)			0.5	3.5	6.5	9.5
Time (h)	SIFT	Local Memory	3.71	0.53	0.28	0.19
		Local Disk	22.27	3.17	1.71	1.17
	Deep	Local Memory	3.69	0.53	0.29	0.19
		Local Disk	22.37	3.18	1.71	1.17
	Tweets	Local Memory	2.22	0.32	0.17	0.12
		Local Disk	13.35	1.91	1.03	0.70

similarity indexes, prioritizing result accuracy over search performance. Recent researchers have proposed approximate algorithms [38, 44, 45, 54, 65, 67, 68] to balance efficiency and accuracy, which can be further categorized as δ-ϵ-Approximate algorithms and ng-Approximate algorithms based on guarantees. The former refers to probabilistic algorithms [38, 45, 54] with δ and ϵ acting as control parameters for probability guarantees. The latter refers to non-guaranteed approximate algorithms that typically utilize the concept of "neighbors of neighbors are also neighbors" by constructing neighbor graphs such as RNNG [44], NSG [67], and HNSW [68]. While these approximate indexes may not ensure precision, they offer high efficiency and have proven effective in practical validations. However, all of these methods have limitations. First, they lack scalability for large-scale vector searches as they primarily run in memory. Second, they are restricted to single-machine vector searches and don't support the addition of computational and storage resources. Third, they are ill-suited for streaming vector search scenarios with dynamic data updates due to their reliance on static indexes. While there are incremental indexes for single-machine vector search [18, 20, 23, 26, 27, 32, 47, 78, 80, 87, 88], they struggle to maintain fast search speeds in a multi-machine cluster under high data flow rates. In contrast, VStream, built upon the SPE model [19], addresses these challenges by implementing a new distributed streaming vector search engine.

(B) Vector Search Engines typically employ similarity search techniques to retrieve high-dimensional vectors. These engines can be broadly classified into two categories. The first category includes libraries such as Facebook Faiss [57], Microsoft SPTAG [13], HNSWlib [68], and Annoy [1], which extend vector search algorithms for one or multiple types of vector indexes. Additionally, they introduce optimizations like multi-threading. However, they are not complete systems and thus lack capabilities such as distributed computation, fault recovery, and consistency guarantees. The second category includes distributed vector search systems, such as Vearch [64], Vespa [16], Weaviate [17], Vald [15], Qdrant [12], Pinecone [10], Milvus [49, 83], and Chroma DB [3]. Note that, general-purpose databases like ElasticSearch [6], Postgres [11], AnalyticDB-V [85], and PASE [90] are gradually being equipped with vector search capabilities. However, they do not offer specific optimizations for streaming vector search scenarios. They still need batch data transfers between nodes and lack capabilities for handling online data and indexes, resulting in higher response latency. In contrast, VStream is designed for streaming vector search, offering record-at-a-time capability, supporting four-layer storage.

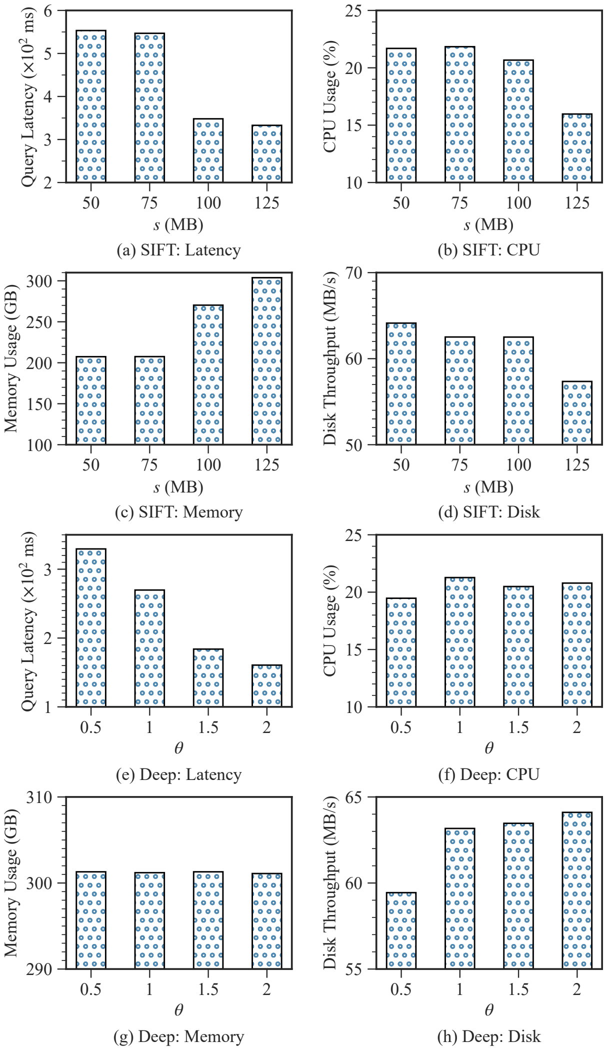


Figure 15: Configuration Knobs Tuning

8 CONCLUSIONS

In this work, we introduce VStream, a streaming vector search system. VStream provides a dynamic partitioner for adaptive data partitioning. Data is stored in a hierarchical storage mechanism with support for multiple types of storage. It utilizes efficient vector compression and hot-cold separation strategies for optimization. In the future, we plan to extend our hierarchical storage mechanism to GPUs, leveraging the powerful vector computing capabilities to further enhance system efficiency. Additionally, multi-vector search is also one of the functionalities we plan to support.

ACKNOWLEDGMENTS

This work was supported by the NSFC under Grants No. (62025206, U23A20296, 62402422, and 62472377), and Zhejiang Province's "Lingyan" R&D Project under Grant No. 2024C01259. Yunjun Gao is the corresponding author of the work.

REFERENCES

[1] 2024. Annoy: Approximate Nearest Neighbors Oh Yeah. https://github.com/spotify/annoy

[2] 2024. Benchmarks for Billion-Scale Similarity Search. https://research.yandex.com/datasets/biganns

[3] 2024. ChromaDB. https://www.trychroma.com/

[4] 2024. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/

[5] 2024. DB Engines Ranking. https://db-engines.com/en/ranking/vector+dbms

[6] 2024. ElasticSearch: Open Source, Distributed, RESTful Search Engine. https://github.com/elastic/elasticsearch

[7] 2024. Flink Guide. https://nightlies.apache.org/flink/flink-docs-release-1.18/

[8] 2024. Flink Milvus Connector. https://github.com/CuitingChen/flink-connector-milvus

[9] 2024. Milvus Sizing Tool. https://milvus.io/tools/sizing/

[10] 2024. Pinecone. https://www.pinecone.io/

[11] 2024. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/

[12] 2024. Qdrant. https://qdrant.tech/

[13] 2024. SPTAG: A library for fast approximate nearest neighbor search. https://github.com/microsoft/SPTAG

[14] 2024. Twitter Dataset. https://snap.stanford.edu/data/twitter7.html

[15] 2024. Vald. https://github.com/vdaas/vald

[16] 2024. Vespa. https://vespa.ai/

[17] 2024. Weaviate. https://github.com/semi-technologies/weaviate

[18] Cecilia Aguerrebere, Mark Hildebrand, Ishwar Singh Bhati, Theodore L. Willke, and Mariano Tepper. 2024. Locally-Adaptive Quantization for Streaming Vector Search. *CoRR* abs/2402.02044 (2024). https://doi.org/10.48550/ARXIV.2402.02044 arXiv:2402.02044

[19] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803.

[20] Guilherme Andrade, Willian Barreiros, Leonardo Rocha, Renato Ferreira, and George Teodoro. 2024. Large-scale response-aware online ANN search in dynamic datasets. *Clust. Comput.* 27, 3 (2024), 3499–3519.

[21] Sunil Arya and David M. Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *ACM/SIGACT-SIAM*. 271–280.

[22] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. In *ECCV*. 209–224.

[23] Guy E. Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *ALENEX*, Cynthia A. Phillips and Bettina Speckmann (Eds.). 195–208.

[24] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguistics* 5 (2017), 135–146.

[25] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Networks* 30, 1-7 (1998), 107–117.

[26] Sebastian Bruch, Franco Maria Nardini, Amir Ingber, and Edo Liberty. 2023. An Approximate Algorithm for Maximum Inner Product Search over Streaming Sparse Vectors. *CoRR* abs/2301.10622 (2023).

[27] Sebastian Bruch, Franco Maria Nardini, Amir Ingber, and Edo Liberty. 2024. An Approximate Algorithm for Maximum Inner Product Search over Streaming Sparse Vectors. *ACM Trans. Inf. Syst.* 42, 2 (2024), 1–43.

[28] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *CoRR* abs/2303.12712 (2023).

[29] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10 (2017), 1718–1729.

[30] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[31] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (1985), 63–75.

[32] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *NeurIPS*. 5199–5212.

[33] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *DLRS@RecSys*. 7–10.

[34] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB*. 426–435.

[35] Open Geospatial Consortium et al. 2017. Discrete Global Grid Systems Abstract Specification.

[36] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *ACM Conference on Recommender Systems*. 191–198.

[37] Emmanuel Gbenga Dada, Joseph Stephen Bassi, Haruna Chiroma, Shafi'i Muhammad Abdulhamid, Adebayo Olusola Adetunmbi, and Opeyemi Emmanuel Ajibuwa. 2019. Machine learning for email spam filtering: review, approaches and open research problems. *Heliyon* 5 (2019).

[38] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *ACM Symposium on Computational Geometry*. 253–262.

[39] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2018. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *Proc. VLDB Endow.* 12, 2 (2018), 112–127.

[40] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2019. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *Proc. VLDB Endow.* 13, 3 (2019), 403–420.

[41] Yury Elkin and Vitaliy Kurlin. 2022. Paired compressed cover trees guarantee a near linear parametrized complexity for all k-nearest neighbors search in an arbitrary metric space. *CoRR* abs/2201.06553 (2022).

[42] Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. 2020. Learned Step Size quantization. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

[43] Absalom E. Ezugwu, Abiodun M. Ikotun, Olaide Nathaniel Oyelade, Laith Mohammad Abualigah, Jeffrey O. Agushaka, Christopher I. Eke, and Andronicus Ayobami Akinyelu. 2022. A comprehensive survey of clustering algorithms: State-of-the-art machine learning applications, taxonomy, challenges, and future research prospects. *Eng. Appl. Artif. Intell.* 110 (2022), 104743.

[44] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.

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

[46] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 4 (2014), 744–755.

[47] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *WWW*. 3406–3416.

[48] Mihajlo Grbovic and Haibin Cheng. 2018. Real-time Personalization using Embeddings for Search Ranking at Airbnb. In *SIGKDD*. 311–320.

[49] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: A Cloud Native Vector Database Management System. *Proc. VLDB Endow.* 15, 12 (2022), 3548–3561.

[50] Helia Hashemi, Aasish Pappu, Mi Tian, Praveen Chandar, Mounia Lalmas, and Benjamin A. Carterette. 2021. Neural Instant Search for Music and Podcast. In *KDD*. 2984–2992.

[51] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2018. Identifying impactful service system problems via log analysis. In *ESEC/SIGSOFT*. 60–70.

[52] David Hilbert and David Hilbert. 1935. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes: Nebst Einer Lebensgeschichte* (1935), 1–2.

[53] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry P. Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *CIKM*. 2333–2338.

[54] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 9, 1 (2015), 1–12.

[55] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *ACM Symposium on the Theory of Computing*. 604–613.

[56] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128.

[57] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Trans. Big Data* 7, 3 (2021), 535–547.

[58] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *SIGIR*. 39–48.

[59] 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.

[60] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *NetDB*, Vol. 11. 1–7.

[61] Pushpendra Kumar and Ramjeevan Singh Thakur. 2018. Recommendation system techniques and related issues: a survey. *International Journal of Information Technology* 10 (2018), 495–501.

[62] Sudarshan Lamkhede and Sudeep Das. 2019. Challenges in Search on Streaming Services: Netflix Case Study. In *SIGIR, France, July 21-25, 2019*, Benjamin Piwowarski, Max Chevalier, Éric Gaussier, Yoelle Maarek, Jian-Yun Nie, and Falk Scholer (Eds.). 1371–1374.

[63] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. 2014. *Mining of Massive Datasets, 2nd Ed.*

[64] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, and Ning Wang. 2019. The Design and Implementation of a Real Time Visual Search System on JD E-commerce Platform. *CoRR* abs/1908.07389 (2019).

[65] Linhao Li and Qinghua Hu. 2020. Optimized high order product quantization for approximate nearest neighbors search. *Frontiers Comput. Sci.* 14, 2 (2020), 259–272.

[66] Xingxin Li, Youwen Zhu, Rui Xu, Jian Wang, and Yushu Zhang. 2024. Indexing dynamic encrypted database in cloud for efficient secure k-nearest neighbor query. *Frontiers Comput. Sci.* 18, 1 (2024), 181803.

[67] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.* 45 (2014), 61–68.

[68] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.

[69] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1 (2010), 330–339.

[70] Fabian Mentzer, David Minnen, Eirikur Agustsson, and Michael Tschannen. 2024. Finite Scalar Quantization: VQ-VAE Made Simple. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*.

[71] Erxue Min, Xifeng Guo, Qiang Liu, Gen Zhang, Jianjing Cui, and Jun Long. 2018. A Survey of Clustering With Deep Learning: From the Perspective of Network Architecture. *IEEE Access* 6 (2018), 39501–39514.

[72] Jay Nanavati and Unnati Patel. 2023. Hybrid Model for Analysis of Social Media Posts for Identification of Depression and Measuring Its Severity. *ICDSNS* (2023), 1–5.

[73] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank Citation Ranking : Bringing Order to the Web. In *The Web Conference*.

[74] Giuseppe Peano and G Peano. 1990. *Sur une courbe, qui remplit toute une aire plane.*

[75] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (2015), 1816–1827.

[76] Loren K Platzman and John J Bartholdi III. 1989. Spacefilling curves and the planar travelling salesman problem. *Journal of the ACM (JACM)* 36, 4 (1989), 719–737.

[77] Jin Shieh and Eamonn J. Keogh. 2008. *i*SAX: indexing and mining terabyte sized time series. In *SIGKDD*. 623–631.

[78] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *ArXiv* abs/2105.09613 (2021).

[79] Alan Jay Smith. 1987. *Design of CPU cache memories.*

[80] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. 2013. Streaming Similarity Search over one Billion Tweets using Parallel Locality-Sensitive Hashing. *Proc. VLDB Endow.* 6, 14 (2013), 1930–1941.

[81] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023).

[82] Soroush Vosoughi, Deb K. Roy, and Sinan Aral. 2018. The spread of true and false news online. *Science* 359 (2018), 1146–1151.

[83] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. *Proceedings of the 2021 International Conference on Management of Data* (2021).

[84] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB*. 194–205.

[85] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proc. VLDB Endow.* 13, 12 (2020), 3152–3165.

[86] Shengli Wu. 2012. Linear combination of component results in information retrieval. *Data Knowl. Eng.* 71, 1 (2012), 114–126.

[87] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *SOSP*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). 545–561.

[88] Zhaozhuo Xu, Weijie Zhao, Shulong Tan, Zhixin Zhou, and Ping Li. 2022. Proximity Graph Maintenance for Fast Online Nearest Neighbor Search. *CoRR* abs/2206.10839 (2022).

[89] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2017. DPiSAX: Massively Distributed Partitioned iSAX. In *2017 IEEE International Conference on Data Mining, ICDM 2017, New Orleans, LA, USA, November 18-21, 2017*. 1135–1140.

[90] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *SIGMOD*. 2241–2253.

[91] Matei A. Zaharia, Reynold Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph E. Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark. *Commun. ACM* 59 (2016), 56–65.

[92] Neil Zeghidour, Alejandro Luebs, Ahmed Omran, Jan Skoglund, and Marco Tagliasacchi. 2022. SoundStream: An End-to-End Neural Audio Codec. *IEEE ACM Trans. Audio Speech Lang. Process.* 30 (2022), 495–507.

[93] Xu Zhang, Felix X. Yu, Ruiqi Guo, Sanjiv Kumar, Shengjin Wang, and Shih-Fu Chang. 2015. Fast Orthogonal Projection Based on Kronecker Product. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. 2929–2937.

[94] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. 2019. Tools and benchmarks for automated log parsing. In *ICSE*, Helen Sharp and Mike Whalen (Eds.). 121–130.

[95] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2016. ADS: the adaptive data series index. *VLDB J.* 25, 6 (2016), 843–866.