Vexless: A Serverless Vector Data Management System Using Cloud Functions

YONGYE SU, Purdue University, USA
YINQI SUN, Purdue University, USA
MINJIA ZHANG, University of Illinois Urbana-Champaign, USA
JIANGUO WANG, Purdue University, USA

Cloud functions, exemplified by AWS Lambda and Azure Functions, are emerging as a new computing paradigm in the cloud. They provide elastic, serverless, and low-cost cloud computing, making them highly suitable for bursty and sparse workloads, which are quite common in practice. Thus, there is a new trend in designing data systems that leverage cloud functions. In this paper, we focus on vector databases, which have recently gained significant attention partly due to large language models. In particular, we investigate how to use cloud functions to build high-performance and cost-efficient vector databases. This presents significant challenges in terms of how to perform sharding, how to reduce communication overhead, and how to minimize cold-start times.

In this paper, we introduce Vexless, the first vector database system optimized for cloud functions. We present three optimizations to address the challenges. To perform sharding, we propose a global coordinator (orchestrator) that assigns workloads to Cloud function instances based on their available hardware resources. To overcome communication overhead, we propose the use of stateful cloud functions, eliminating the need for costly communications during synchronization. To minimize cold-start overhead, we introduce a workload-aware Cloud function lifetime management strategy. Vexless has been implemented using Azure Functions. Experimental results demonstrate that Vexless can significantly reduce costs, especially on bursty and sparse workloads, compared to cloud VM instances, while achieving similar or higher query performance and accuracy.

CCS Concepts: • **Information systems** → **Data management systems**; **Semi-structured data**.

Additional Key Words and Phrases: Vector Databases, Cloud Functions, Serverless Databases, Serverless Computing

ACM Reference Format:
Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. 2024. Vexless: A Serverless Vector Data Management System Using Cloud Functions. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 187 (June 2024), 26 pages. https://doi.org/10.1145/3654990

1 INTRODUCTION

Cloud functions have gained great popularity in recent years as a new computing paradigm in the cloud. Major cloud service providers have offered cloud function services such as AWS Lambda [4], Azure Functions [6], and Google Cloud Functions [9]. Compared to traditional cloud

Authors' addresses: Yongye Su, su311@purdue.edu, Purdue University, West Lafayette, Indiana, USA, 47907; Yinqi Sun, sun1226@purdue.edu, Purdue University, West Lafayette, Indiana, USA, 47907; Minjia Zhang, minjiaz@illinois.edu, University of Illinois Urbana-Champaign, Urbana, Illinois, USA, 61801; Jianguo Wang, csjgwang@purdue.edu, Purdue University, West Lafayette, Indiana, USA, 47907.

ACM 2836-6573/2024/6-ART187
https://doi.org/10.1145/3654990

computing where users have to manage virtual machines (VMs), cloud functions are designed to hide operational concerns. Users only need to submit the payload functions as code, and cloud service providers are responsible for automatically executing them upon triggering events. The instance will terminate immediately after the payload function has finished executing, and users will only pay for the total uptime of function invocations. Thus, cloud functions are attractive among performance and cost-sensitive applications with bursty and sparse workloads.

As a result, there is a new trend of optimizing data systems for cloud functions to be elastic and cost-efficient [28, 55, 59]. For example, Starling [59] and Lambada [55] are designed to improve analytical databases by dynamically controlling the resources allocated for the cloud function invocation on a query-by-query basis. This avoids over-provisioning of unnecessary resources while ensuring competitive performance. Moreover, Bian et al. propose to use a hybrid of virtual machines and cloud functions to improve cost-efficiency in data analytics [28].

In this paper, we propose the use of cloud functions to support efficient vector similarity search in vector databases, as vector databases have recently gained significant attention [57, 68, 74]. The rise of vector databases is attributed to (1) vector embedding, where unstructured data can be embedded into high-dimensional vectors for advanced data analytics [26, 44, 53], and (2) large language models (LLMs), where vector databases can address many limitations in LLMs such as hallucinations [21] and the inability to incorporate up-to-date information [20].

A straightforward solution is to treat each cloud function as a cloud VM and apply the existing distributed vector search approach. However, cloud functions bring unique challenges due to their restrictive and stateless characteristics.

Challenge 1. First, unlike virtual machines, the resources for cloud functions can only be provisioned automatically by the service provider and are adjusted at runtime, meaning the resources available for each invocation will vary, and each invocation also has an upper limit on resources. This necessitates the proper partition of workloads to fit the available resources in each function instance.

Challenge 2. Another challenge is the communication overhead. Because the workloads are partitioned with finer granularity due to limited resources on each instance, there will be more frequent synchronizations between peer functions working on the same query. Although synchronization overhead can be alleviated by optimizing communication protocols, it still remains significant because communications between cloud functions rely on high-latency internet connections.

Challenge 3. Finally, because cloud functions are activated on demand, each function invocation would have a cold-start delay to start up and initialize the environment and context, before the payload function can be executed. This can cause significantly delayed responses, usually seconds, for a cold-start invocation. A straightforward solution could be reusing each invocation by keeping it active after its payload has finished while waiting for the next payload. However, because of the service model of cloud functions, for the same hardware configuration, the cost for cloud functions to stay active is much higher than virtual machines [28]. Increased active time translates to significantly increased cost, which would defeat the purpose of using cloud functions.

Contributions. The main contribution and overall novelty of this work is the design and implementation of Vexless, the first vector database management system optimized for cloud functions that supports vector search for both bursty and non-bursty workloads. It develops a series of new techniques:

- A new sharding strategy and dispatch engine (Orchestrator) for vector similarity search workloads that employs the Constrained K-Means algorithm. The algorithm maximizes the

utilization of each cloud function instance by evenly distributing the workloads across each instance according to the resources they have respectively. See Section 3.2.
- A new communication mechanism between different function invocations that carry out different parts of the task. We use Azure Durable Functions [13] as a centralized communication hub that coordinates the distribution of messages across worker functions. Each worker function will also have a bilateral message queue using Azure Queue Storage [14] to ensure non-blocking, asynchronous communication. The communications are handled completely by Azure's internal services where low-latency internal channels are used. See Section 3.3.
- A novel adaptive scheduling algorithm within the Orchestrator that reduces the cold-start time. First, each cloud function will stay alive in case future similar tasks can reuse this function without invoking a new one. With the intuition that adjacent queries are usually related, the algorithm prolongs the lifetimes for frequently invoked functions to maximize the opportunity for function reuse while maintaining reasonable active time. Finally, we utilize idle invocations that are still within their lifetimes to perform optional exhaustive search tasks which will increase overall search accuracy. See Section 3.4.

Open-source. The code is open-sourced at https://github.com/Vexless/Vexless.

2 BACKGROUND AND TARGET WORKLOADS

2.1 Vector Similarity Search

Vector similarity search, as the foundation of vector databases, aims to find similar vectors in the dataset to the vector representing the query (Query Vector). The measure for vector similarity is usually L2 (Euclidean) or Cosine distance. This essentially involves finding k Nearest Neighbors to the query vector, abbreviated as k-NN. However, calculating k-NN can be expensive because it requires distance computation to every point in the dataset and maintaining the top-k results. This process can take a long time for a single query on a billion-scale dataset. Although indexes like kd-trees are able to reduce the search time using their hierarchical structure, the search performance degrades significantly when dimensionality increases, a phenomenon often referred to as the "curse of dimensionality" [37, 51]. However, many interactive applications that involve k-NN search require millisecond-level query time on billion-scale datasets with over a thousand dimensions. In this case, where 100% accuracy is usually not required, we can use algorithms that find Approximate Nearest Neighbors (ANN) orders of magnitude faster with only slightly lower recall. The recall rate, or "recall" for short, is a popular measure for the quality of an ANN result. It is defined as the percentage of real top-k results within the k results the algorithm has returned. Note that while in certain applications, such as mass spectrometry [46], it is possible to use exact vector search, this work focuses on approximate vector search.

Indexes for ANN include tree-based [48, 63], hash-based [24, 39], Product Quantization (PQ) [40], and graph-based methods [38, 50, 65]. Tree-based methods include randomized kd-tree forest and K-Means trees, which were popular for large-scale datasets with low dimensions due to their stable results. Unfortunately, their performance degrades significantly with higher data dimensionality [68]. Probabilistic hashing such as LSH [24, 39] has high query speed at low recall, but it typically loses its speed advantage at slightly higher recall. Product Quantization methods [40] are able to achieve high query speed with respectable recall, but they are also known to be less performant for high recall settings. Graph-based methods [38, 50, 65] have recently emerged as the current state-of-the-art algorithm for ANN [52], thanks to their fastest query speed with high recall in large-scale, high-dimensional datasets. In particular, HNSW (Hierarchical Navigable Small

World) [50] is a popular graph-based ANN method. It uses a hierarchical graph index to progressively find better entry points for each layer that are closer to the query vector.

2.2 Cloud Functions

Traditional cloud computing services require users to manage every aspect of the server. This includes selecting hardware configurations, choosing operating systems, as well as installing and managing required software. If users decide to scale up the hardware capabilities, they must either manually launch and set up new parallel instances or migrate to a new instance with more powerful hardware configurations. Users are charged for a cloud instance from when it is created to the time it is deleted, regardless of its operational status, meaning that even if the server is powered off, users will still be charged the full price.

On the other hand, cloud functions, as a form of serverless computing, allow users to only submit tasks in the form of a payload function, i.e., a piece of code, and the cloud service provider will take care of the underlying infrastructure management such that the actual servers are transparent to users [4, 6, 9, 43]. This includes automatic provisioning and environment setup; starting up the server and executing the payload functions upon triggering events; scaling up during runtime by launching more instances; shutting down the server when the payload is finished. Also, users are only charged for the total uptime, i.e., the time when the functions are running, which is referred to as the pay-as-you-go cost model. Compared to traditional cloud computing, cloud functions offer the advantage of no server management, a fine-grained billing model, and faster start-up time (within seconds vs. minutes for VMs).

The benefit comes at a price. First, the unit prices for similar hardware configurations are higher for cloud functions than for VMs, making cloud functions unsuitable for dense and continuous workloads that require long uptime. The second problem is the limited maximum hardware resources, which require resource-intensive applications to use horizontal scaling, i.e., increase parallelism by launching more instances. The resources provisioned for each invocation are also different and controlled by the service provider; therefore, it will require delicate workload allocation to ensure that different tasks finish at around the same time. Another problem stems from the stateless nature of cloud functions, meaning they do not have local storage. To maintain runtime states, such as intermediate results and labels, applications often need a separate storage service. This not only incurs additional cost but also introduces more communication overhead to the storage service. Finally, cloud functions terminate immediately after finishing their payloads, so new function invocations are required for every task, and the start-up time will add to the total execution time. In contrast, VMs usually stay online 24/7 since users will be charged regardless of the operational status.

Fortunately, cloud service providers have managed to alleviate some of the problems with specialized services. Stateful Functions [1, 5, 13, 23] like Azure Durable Functions [13], AWS Step Functions [5], and Stateful Function Invocations for Alibaba Cloud [1] allow a cloud function instance to have persistent storage. To reduce communication overhead between peer functions running in parallel, some cloud providers offer message queues, such as Azure Queue Storage [14], Amazon Simple Queue Service [3], Message Service for Alibaba Cloud [2], and Google Cloud Pub/Sub [12], as a buffer for non-blocking communications. In this work, we chose Microsoft Azure as the cloud provider, but our techniques are also generalizable to other cloud providers that offer stateful serverless computing services and message queues mentioned above.

However, there are still challenges in building a performant and cost-effective vector database with cloud functions. These include even distribution of workloads, minimizing communication overhead, and reducing cold-start time. Vexless addresses these challenges with architectural optimizations that will be elaborated in Sec. 3.

2.3 Target Workloads

This paper targets two types of workloads that are more suitable for cloud functions following [28, 55, 59, 67]:

- Type 1: Sparse workload. The sparse workload is characterized by a low volume of queries, such that the number of submitted queries is insufficient to fully utilize the system at all times. This does not necessarily mean that the number of submitted queries is zero. For instance, if each vector search takes 1ms (i.e., 1000 qps), and the number of issued queries is less than say 500, then we consider it a sparse workload. It is important to note that even if the workload is continuous, as long as it is sparse, cloud functions can be more suitable.
- Type 2: Bursty workload. The key characteristic of the bursty workload is that it has peak hours and spikes. This workload can be continuous or intermittent, dense or sparse. As long as there are spikes, cloud functions can reduce costs since they are billed per query. In contrast, VMs are provisioned and charged based on peak workload, often wasting a lot of resources.

In summary, cloud functions are suitable for sporadic and unpredictable workloads, i.e., sparse and bursty workloads. Indeed, many real-world applications have such characteristics. For example, Figure 16 of [70] shows that the vehicle peccancy detection system, which requires vector search, exhibits a significant variation in the number of queries per second. Also, the work [72] states that "online users tend to exhibit bursty behavior" in web queries requiring vector search. Similarly, [75] says that question answering on Twitter, which also involves vector search, is bursty, as shown in Figure 4 of [75]. Furthermore, workloads in IoT applications and many data processing platforms are sporadic and bursty [55, 59, 60].

In contrast, VMs are more suitable for continuous workloads with stable, predictable, and extensive queries, such that the resource utilization in VMs will always be high. Considering that VMs are cheaper than cloud functions per resource unit, the overall cost for VMs will also be lower.

3 SYSTEM DESIGN

3.1 Overview

Main Idea. The main idea of our system is to follow a distributed vector search approach, treating each cloud function as a computation unit, partitioning vector data into distinct shards, building a unique index for each shard, and then conducting searches within those shards. In particular, there are two primary phases:

- **Sharding Phase:** The data vectors are horizontally partitioned using an unsupervised balanced clustering method, creating distinct data cluster shards with centroid vectors, and building indexes for further vector searches. Each index shard is then stored within an individual cloud function, essentially converting a cloud function into a self-contained individual vector data processor.
- **Search Phase:** When the system receives a query vector (q), the system will identify the appropriate index shards to perform vector search. Using an optimized communication mechanism, Vexless activates cloud functions containing the shards, and each of these functions will return a partial vector search result. Then the aggregator receives and re-ranks these search outputs to compute the global top-k results as the final output. However, implementations using serverless functions would inherently face the frequent cold-start problem. To overcome this, we adopted stateful functions and a heuristic keep-alive strategy. This significantly reduces the cold-start overhead in sporadic and unpredictable real-world search scenarios.

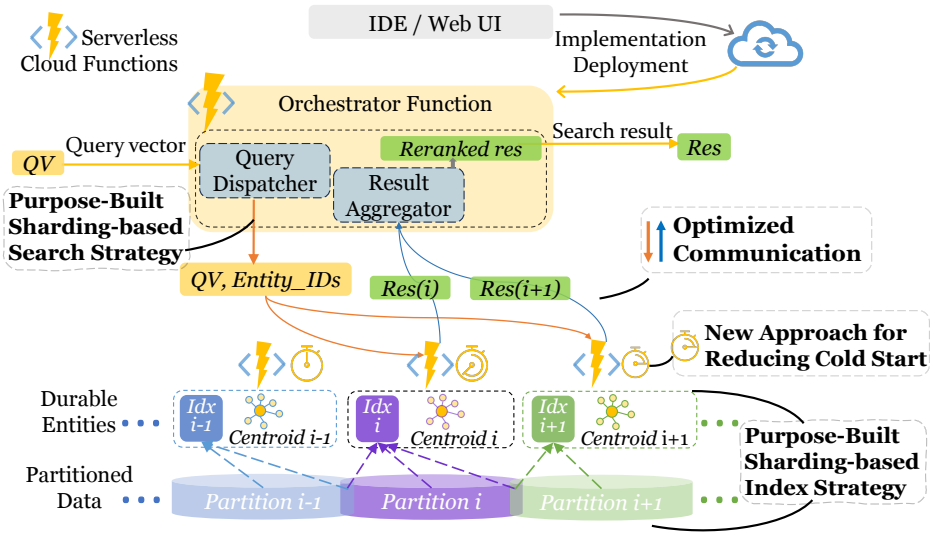
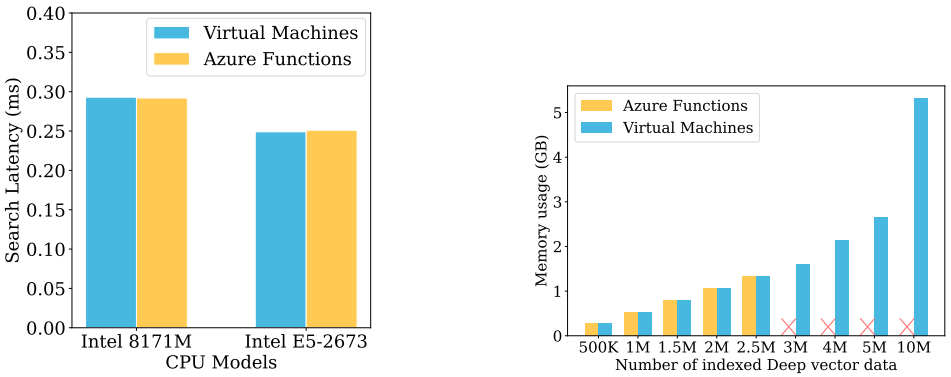


Fig. 1. System Architecture Overview of Vexless

As shown in Figure 1, Vexless adopted a variety of techniques for managing and searching large amounts of vector data. The yellow lightning symbols represent cloud functions we used in Vexless such as durable entity functions and the orchestrator function, with the latter working as a serverless resource coordinator, task dispatcher, function lifetime scheduler, and result aggregator, the users can have the serverless system control handed to the orchestrator. Such a distributed search architecture solves data distribution via sharding, leveraging cloud functions as compute units, and accelerating the distributed search process with our refined communication protocol.

Challenges. However, vector search systems face several non-trivial challenges in leveraging the cloud functions: the constraint on memory size, communication latency, and frequent cold-start overhead.

- **Sharding and Managing Data:** To handle a large amount of vector data, it is not possible to simply use one cloud function to handle the entire dataset (as shown in Figure 2b) due to the memory limit for one cloud function. Consequently, a naive implementation is to deploy multiple functions to accommodate the large scale of data. Notably, traditional uniform sharding will result in high cost and increased query time due to the buckets effect caused by "straggler tasks". Furthermore, clustering-based sharding can also result in uneven clusters, necessitating more function invocations [65]. We explore this further in Section 3.2.
- **Reducing Data Communication Overhead:** Traditional HTTP-based and storage-centric communication protocols impose a considerable communication overhead in vector search, which could be problematic for such latency-sensitive applications. Thus, a novel approach to streamline communication in this scenario becomes critical. A detailed discussion of our strategies are shown in Section 3.3.
- **Minimizing Cold-start Time:** Last but not least, real-world vector search on cloud functions could take very long time when all the functions are invoked at cold state. Even so, existing works for reducing serverless function cold-start times are insufficient, they either do not fit the Azure platform [47] and introduce additional service cost [73], or do not solve



(a) Vector Search Latency

(b) Memory Capacity

Fig. 2. Vector Search Computing and Memory Characterization of Azure Cloud Functions

the cold-start problem with just prebaking for nearly frequent activation of our vector search with low latency [64]. This challenge is studied in depth in Section 3.4.

3.2 Purpose-Built Sharding-based Index and Search Strategy

Cloud functions are built for lightweight tasks and often have very limited resources (CPU, memory, disk storage, etc.). For example, in the serverless plan, Azure Functions have a maximum memory limit of 1.5 GB per instance [7, 69]. As shown in Figure 2b, each bar represents the memory usage for the specific number of indexed vectors. An "X" on a bar indicates that the corresponding solution is not able to accommodate the data index for the specified number of vectors. However, in Figure 2a, we observed that within the Azure CF memory limit, its vector search computing latency remains the same as the virtual machine when using the same CPU configuration.

This phenomenon brings up an interesting question: can we have the same level of computing power as virtual machines while paying in a pay-as-you-go solution with great elasticity? As such, one natural idea is to split the vector database into multiple smaller partitions and use the aggregated resources of multiple serverless cloud functions to serve user requests. Notably, the primary objective of utilizing multiple serverless cloud functions is to ensure cost efficiency while maintaining high search accuracy and low latency.

However, a naive implementation using many serverless cloud functions cannot achieve the above goal because it either cannot meet the latency requirement, or it uses more serverless cloud functions than necessary, leading to increased costs. To address these challenges, we rearchitect the vector search index and propose new search algorithms specifically designed for serverless cloud functions to achieve the goals.

While both traditional and vector databases require balanced sharding for efficiency, the unique characteristics of vector data – such as high dimensionality, particular distance metrics, and the need for efficient ANN search algorithms present unique challenges and considerations for partitioning in vector databases. Traditional databases often rely on random, ranged-based, or hash-based sharding, whereas vector databases often require more complex partitioning methods that account for the data's spatial distribution and the nature of vector space, as traditional sharding methods may not effectively preserve the proximity relationships essential for accurate and efficient vector search. To split a vector database into multiple partitions, prior works often resort to uniform sharding, hash-based sharding [24, 49], and K-Means clustering-based sharding such as

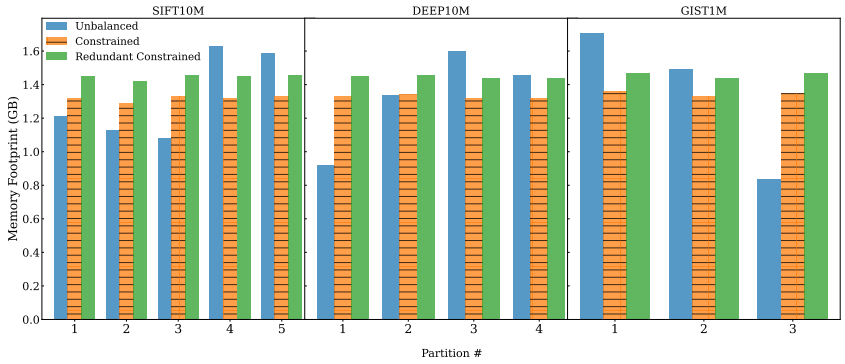


Fig. 3. Comparison of Vector Data Clustering Results using Different Unsupervised Clustering Algorithms

Faiss [42], SPTAG [31], and SPANN [32]. While the aforementioned techniques are common, they may not be suitable for constructing indexes in serverless cloud functions. For example, uniform sharding evenly distributes a vector database across multiple partitions. Yet, due to the lack of locality within each partition, a query would need to search all partitions to identify the nearest neighbors. Both hash-based and clustering-based sharding group similar data points together, reducing the number of groups a query needs to search to retrieve near neighbors. However, these methods have been criticized for various drawbacks, one of which is the issue of partition imbalance. Though SPANN appears to achieve balanced partitioning, it relies on disk-based solutions without setting a strict partition limit on the index partition memory usage. Instead, it emphasizes overcoming latency issues through its balanced posting list. Our work focuses on memory constraints, and our solution trade-off redundancy with only a 12% memory increase, nearly half of the 20% expansion SPANN incurs [32], thereby optimizing memory costs.

3.2.1 Index strategy. Figure 3 provides a concrete example of how different clustering algorithms partition three common vector search datasets. The current K-Means implemented in Faiss library [42] and employed in Inverted File Index (IVF) [40] does not guarantee that the clusters are balanced, even pre-setting *min_points_per_centroid* and *max_points_per_centroid* parameters, as it only defines the ratio of the number of training points to the number of centroids. As shown, the memory footprint of different partitions can reach up to 113% of the memory capacity limit. This imbalance may not pose a significant issue when deploying ANN search in data center servers, which have substantial memory capacity. However, serverless cloud functions have limited memory resources that vector search heavily relies on, and such imbalance could easily lead to out-of-memory problems. Alternatively, it might require the use of additional serverless cloud functions to handle a dataset, thereby increasing unnecessary deployment costs.

To address this imbalanced partition issue, we utilize constrained K-Means [29] as shown in the first half of Algorithm 1, as it produces the best and well-balanced result.

While balanced K-Means is not new, applying it in Vexless poses unique challenges in the index design. For example, balanced K-Means addresses the issue of partition imbalance, enabling each partition to satisfy the memory constraint of serverless cloud functions. However, this approach results in worse search efficiency when compared to standard K-Means. One reason is that balanced clustering may force more boundary points between two clusters to achieve a better balance, rather than assigning them to their nearest cluster centroids. As such, it increases the likelihood of searching in incorrect clusters during the retrieval phase. For a random query vector q encountered in the boundary of multiple cluster partitions, if we simply consider the partition having the closest distance to q, we may miss its true nearest neighbor vector. This is because

Algorithm 1: Vexless's Indexing with Constrained K-Means Clustering and Redundancy Option

Input: Database D of m base vector in $\mathbb{R}^n$. Initial cluster centroids $C_{1,t}, C_{2,t}, \ldots, C_{k,t}$ at iteration t.
Analyze the single vector dimension of D, linearly infer the maximum number of vectors N_{max} that each cluster can host under the current memory limit, $k \leftarrow \lceil \frac{m}{N_{max}} \rceil$, $N_{min} \leftarrow \frac{m}{k}$.
Stage 1: Clustering-based Sharding
Constrained_Clustering_Algorithm(N_{min}, N_{max}, k) [29]
Get clusters $c_1, c_2, \ldots, c_k$ with centroids $C_1, C_2, \ldots, C_k$.
Stage 2: Index Building
Initialization: For cluster c_i, initialize an empty index partition I_i. Set distance threshold T for redundancy indexing.
for each base vector v in D **do**
  Compute v's distances $\mathrm{d}(v, C_i)$ to centroids $C_1, C_2, \ldots, C_k$.
  Rank $\mathrm{d}(v, C_i)$.
  Include v in the index I_h corresponding to its centroid C_h.
  for each centroid C_i where $i \neq h$ **do**
    if $\mathrm{d}(v, C_i) < T$ **then**
      Add v to I_i.
    end if
  end for
end for

while the q's distance to a partition X's centroid is farther than that of another partition Y, the true nearest neighbor may be in partition X.

This raises a question: can balanced K-Means achieve a similar search efficiency as standard K-Means? To address this issue, we introduce **radius-threshold-based optimization** that significantly improves the search efficiency of balanced K-Means by carefully adding a small set of boundary nodes to each cluster. After the balanced K-Means clustering process with a set of cluster centroids' coordinates with the desired count, to decide which cluster partition each base vector belongs to, we first calculate the distances between q and centroids.

We used HNSW from the HNSWLib library [50] in our implementation as the index algorithm because of its great scalability with higher dimensional vectors and great balance between accuracy and query speed. However, our techniques do not limit the choice of vector index algorithms; other methods like IVF or LSH can also be used.

3.2.2 Search strategy. In Vexless, our serverless vector search starts by receiving a query vector (q) from the user's end. The queries are directly accepted by an external query handler designed for low latency and cost-efficient messaging that forwards the queries to the orchestrator, in order to prevent the function from being overwhelmed with excessive queries.

Given a query q, the orchestrator determines and activates the q-relevant partitions for vector search based on the qualifying distance scores. It calculates the q's distance to different partition centroids and then determines its search partition DEF IDs by comparing the distance score. Once the score is lower than the threshold we use in the indexing phase, the q is dispatched to the corresponding partition. At the same time, it dispatches the q to previously activated Durable Entity Functions (DEFs) for them to voluntarily conduct the vector search. For those q-related partitions, once their DEFs are ignited, they are dedicated to their partition's vector search tasks.

By only activating q-related partitions instead of activating all the partitions, we achieve cost efficiency.

Once a local search on a DEF partition is finished, Vexless channels results via our *Optimized Communication Mechanism* introduced in Section 3.3, ensuring that the combined results from multiple DEFs are delivered and then aggregated. For aggregating results from shards, the returned results include a list of double-value tuples (original vector IDs and their corresponding distances to the query vector). Then we merge and re-rank all the top-k results based on distances returned, as they are globally comparable, and return the global top-k. During irrelevant query periods for activated DEFs, our *new approach for reducing cold-start latency* presented in Section 3.4 plays a crucial role in managing keep-alive times, ensuring a balance between responsiveness and resource conservation cost.

3.3 Optimized Communication Mechanism

Vexless tailors the serverless architecture to the needs of real-time, low-latency vector search. In particular, we identify the primary bottleneck for achieving low-latency vector search is the communication latency across serverless cloud functions. Next, we present a serverless architecture that substantially minimizes communication overhead.

State-of-the-art vector search systems deliver single-digit millisecond search latency on datasets with multi-million records on a single machine. When multiple serverless cloud functions are employed in parallel, additional synchronization is necessary to gather near neighbors from these parallel serverless cloud functions. As such, it raises the question: would this synchronization overhead become a major bottleneck or is it small enough for low-latency vector search?

Before introducing our method, we first examine two common approaches for communication in serverless functions: the global-storage-based method, and the HTTP-based method:

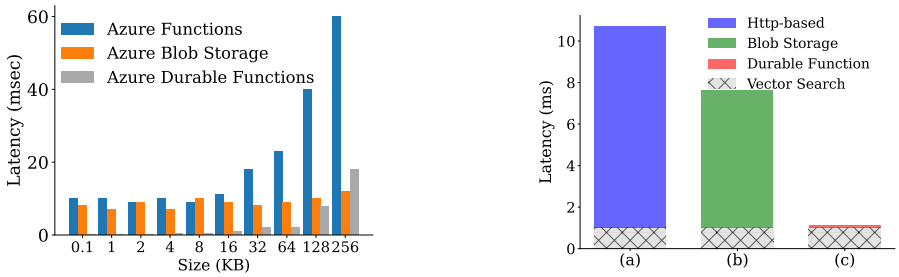
Global-storage-based: Prior works, such as Starling [59], have employed Amazon S3 storage to transfer intermediate results between multiple serverless cloud functions. Since S3 is global storage, multiple serverless cloud functions can utilize it as a medium for data and state exchange for its affordability and universality.

HTTP-based: An alternative approach is to use the conventional HTTP-API of cloud functions as a direct communication channel between serverless functions. This method assigns a unique URL to each function, making it publicly accessible including other serverless cloud functions.

We conduct experiments to measure the actual search time and the communication time spent on synchronization, using the above two methods as well as ours. Figure 4b illustrates the time breakdown. As shown, while the pure search time for high-precision is around 1 millisecond, the time spent on communication usually takes less than 0.1 milliseconds. Therefore, the latency is dominated by network communication. Following Amdahl's law, the search latency cannot be significantly decreased with more serverless cloud functions unless the communication overhead can be reduced.

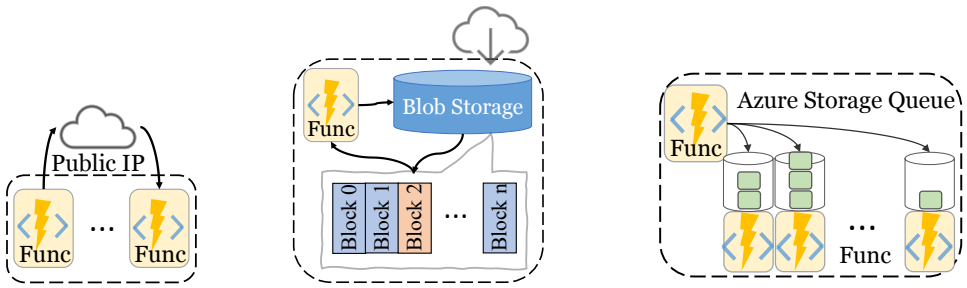
As shown in Figure 5a and Figure 5b, HTTP-based and storage-based methods often seem ineffective in intra-cloud communications simply because the overhead is too high compared with vector search (see Figure 4b) which directly compares the single query vector's transmission overhead of different communication solutions combined with the latency of conducting costly single query search on SIFT1M index for 99.95% recall.

Furthermore, according to Figure 4a as the size of the communication message increases, the communication latency increases from 0.06 ms on 0.1 KB to 7.90 ms on 128 KB for the HTTP-based mechanisms. This increase in communication overhead is not scalable, because the latency



(a) Communication Overhead **(b)** Overhead with Vector Search

Fig. 4. Cloud Communication Overhead Comparison



(a) HTTP-Based **(b)** Blob Storage **(c)** Durable Function

Fig. 5. Comparison Between Various Communication Methods for Vector Search on Azure Serverless Platform

is completely dominated by network transmission, resulting in an end-to-end latency that is significantly worse than server-based solutions. The high overhead of HTTP-based communication is attributed to the costly process of establishing HTTP connections. Given that generally, serverless cloud functions are stateless by default, such a connection must be established each time before Vexless performs research results synchronization in vector searches, leading to substantial overhead. On the other hand, the latency for the global storage-based mechanism remains relatively stable. However, the absolute latency (~10 ms) is still a barrier to achieving low-latency vector search. The reason for the slow yet relatively constant speed of global storage-based communication is that global storage is typically optimized for throughput-oriented workloads with limited bandwidth (for instance, the network throughput for a single page blob of Azure Blob storage is 60 MiB/s [17], which is significantly lower than the mainstream DDR4 main memory bandwidth). Without an efficient communication mechanism, the potential of serverless cloud functions for low-latency vector search remains quite limited.

To tackle the above challenge, Vexless introduces a hybrid serverless architecture for vector search. This approach significantly minimizes communication overhead while maintaining the cost advantage of serverless cloud functions. **Our key idea is the incorporation of a stateful function** (e.g., Azure Durable Function) as the orchestrator function in Vexless. A stateful function has the ability to preserve states across multiple function invocations, which helps in faster recovery after deactivation and minimizes the impact of cold-starts mentioned above. It eliminates the need for costly HTTP-based and storage-based communication during synchronization and allows for more efficient communication channels, such as message passing via Azure Queue Storage [14].

Our experiments in Sec. 4 show that this hybrid approach reduces the communication overhead by an order of magnitude, supporting serverless cloud functions for low-latency vector search while still leading to high search quality with low cost.

3.4 Cold Start Reduction

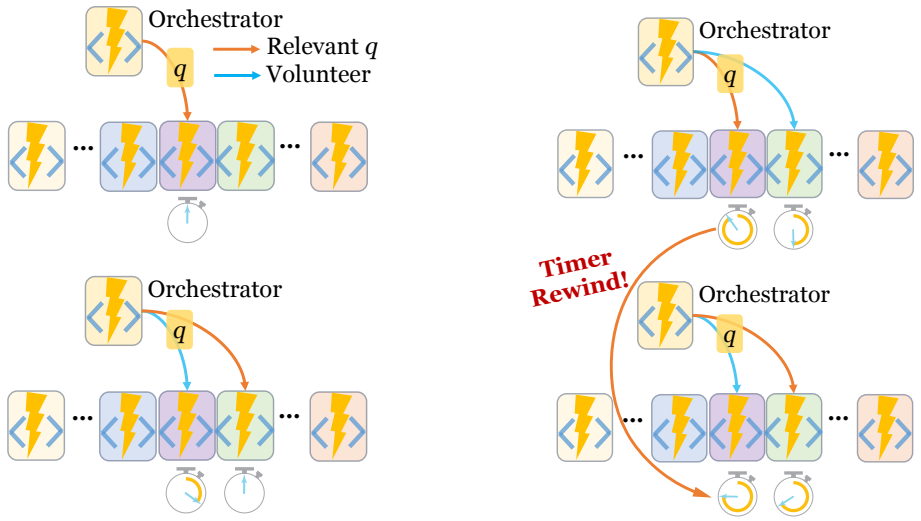
In pay-as-you-go serverless platforms, the term *cold-start* represents the necessary initialization phase of a serverless cloud function. Serverless cloud functions are likely to encounter cold-starts when they are called upon after a period of inactivity or when no warm instances are available to handle an external request. During such a cold-start process, the serverless cloud function must first allocate compute and memory resources to create a new environment for running the serverless cloud function and load the function code before starting the actual execution process. Additionally, a cold-start also triggers the overhead of loading the Vexless vector index, often about 1 GB in size, to DRAM via the high-latency cloud storage, further prolonging the critical path length. Therefore, cold-starts can significantly restrict the performance of Vexless without tailored optimizations. It is also worth noting that among all the runtime stacks and environments in Azure Functions, Python functions running in Linux stand out from peers [10], therefore, our system was built based on the setting.

When invoking serverless cloud functions, the users often need to specify a timeout for the allocated compute and memory resources. During this timeout period, the serverless cloud function instance remains warm, meaning that it can process requests without incurring all the overhead associated with a cold-start. Once this timeout period is over, the serverless cloud function releases its hold of the resources and the subsequent invocation will trigger a cold-start. It is non-trivial to set this timeout duration. A long duration tends to mitigate the cold-start issue but also increases the overall deployment cost. However, a shorter duration gives the opposite tradeoff. More importantly, it is suboptimal to place a fixed timeout duration for all serverless cloud functions at all times, given that actual query distribution may vary (e.g., queries may exhibit diurnal distribution) and different serverless cloud functions may become activated during vector search. Therefore, one optimization opportunity is to adaptively set timeout duration for each serverless cloud function in Vexless.

To tackle the above challenge, Vexless introduces a novel approach to mitigate the overall cold-start issue while maximizing the cost savings with two strategies as follows (shown in Figure 6 and Algorithm 2). The key of such a mechanism is that orchestrator functions in Azure Durable Functions can signal and control Durable Entities to reset or adjust their state through administrative messages. With the timer functionality of the orchestrator function combined with signaling, we can maintain the stateful entities in an active, ready-to-execute state, optimizing the performance and responsiveness of the serverless application.

Rewind Mechanism. The "rewind" operates as an initializing and rewarding mechanism. Upon receiving a random query vector q, this method first winds up task-relevant stateful functions (DEFs) with a full lifecycle, then observes recent query workloads, alters the lifecycle of serverless functions dynamically, and refreshes their "warm" states when necessary. This heuristic approach dynamically adjusts the keep-alive time of each cloud function thread based on the query arrival rate observed in the previous time window. Then the keep-alive time is prolonged if the query workload is high, reducing frequent cold-starts. As described in Algorithm 2, when a q comes with top relevant partitions to search, the corresponding related partitions' running time less than the threshold will be rewound to the time threshold.

Diligent Search Before Laziness Mechanism. Once receiving a query vector from the external handler, our system adopts a diligent search strategy, which emphasizes searching within the most



(a) DEF Starts with a Relevant q and Then Volunteers Search **(b)** Insufficient DEF Timer Being Rewound by Relevant q

Fig. 6. The Two Mechanisms in Section 3.4 to Maintain the Warmness for Recently Used Computing DEFs

probable serverless cloud function target partitions according to our search strategy, but also gives tasks to those running DEFs, letting them work as volunteer DEFs to avoid instance idling in their lifecycles. Only before or after the diligent search phase (the active lifecycle) does the system shift to an opportunistic "lazy" search strategy for inactive DEFs, making sure a DEF will only be invoked by its top relevant search task. This ensures high search relevance in our defined search strategy while maximizing the use of available computing resources.

Figure 6 illustrates the combination of the two aforementioned mechanisms. The volunteer job is a non-relevant query task for the corresponding query, which is the idea that "diligent searching starts with laziness". The rewind operation happens if and only if a relevant query task hits the corresponding partition function with the remaining lifetime being less than the threshold time, and it will rewind again to the threshold. While the rewind mechanism maintains the "warmness" of computing functions that ensure quick response times, the diligent searching mechanism optimizes the search process by using available resources diligently. They work in pairs to enable Vexless to deliver rapid search results while also managing resources effectively.

3.5 Generalizability

Note that although we built our system using the Azure cloud platform, our design ideas can be applicable to other cloud providers such as AWS and GCP. This is because our system mainly relies on services such as cloud functions, stateful functions, and message queue services that are also available in AWS and GCP. For example, we used Azure Durable Functions, which are similar to Google Cloud Workflows services hosted on cloud functions and AWS Step Functions. For Azure Queue Storage, the equivalent services are Google Cloud Pub/Sub and Amazon Simple Queue Service (SQS).

4 EXPERIMENTS

In this section, we perform a comprehensive evaluation of our vector search system that systematically compares our solution with contemporary alternative solutions on the same platform

Algorithm 2: Vexless Rewind Mechanism for Cold Start Reduction

Input: Query vector q, list of serverless function instances F each with function timers T_f and corresponding states S. A function f with $T_f = 0$ is considered as a "cold function".
Output: List of selected function instances $F_{selected}$ to execute the query
Init: $F_{diligent}$, F_{lazy}, **and** $F_{volunteer}$**:**
$F_{diligent}$: Top probable functions (active) related to q in F.
F_{lazy}: Top probable functions (inactive) related to q in F.
$F_{volunteer}$: Active functions not directly related to q in F.
for each function instance f in F **do**
  if f is related to q **then**
    if $T_f \neq 0$ **then**
      $T_f \leftarrow \max(T_{threshold}, T_f)$ /* *Timer Rewind* */
      $F_{diligent} \leftarrow F_{diligent} \cup \{f\}$
    else if $T_f == 0$ **then**
      $F_{lazy} \leftarrow F_{lazy} \cup \{f\}$
    end if
  else if $T_f \neq 0$ **then**
    $F_{volunteer} \leftarrow F_{volunteer} \cup \{f\}$
  end if
end for
$F_{selected} \leftarrow F_{diligent} \cup F_{lazy} \cup F_{volunteer}$
return $F_{selected}$

using the HNSW index (HNSWLib). As a popular method for ANN vector similarity search, HNSW boasts exceptional performance on vector search speeds at high recall rates. We evaluated the monthly cost and end-to-end performance across different vector search solutions to demonstrate the overall competence of Vexless as a vector database. Then we designed experiments in controlled settings to prove the effectiveness of our individual design choices.

4.1 Experiment Setup

Datasets (Table 1): We used three datasets that are commonly used in previous ANN research [32, 58, 68]. The first vector dataset we used for our evaluation was SIFT10M [40]. The Scale-Invariant Feature Transform (SIFT) is a well-known computer vision algorithm that identifies and outlines local features in images. The second dataset, DEEP10M, is extracted from DEEP1B [25]. The DEEP10M dataset also consists of 10 million 96-dimensional L2-normalized [8] floating point vectors, known as deep descriptors. These vectors are taken from the last fully-connected layer (FCL) of the GoogLeNet [66] model, which was pre-trained on the Imagenet [33] classification task. The third dataset, ANN_GIST1M [35], contains 1 million GIST feature vectors that encapsulate the global attributes of images, including color, texture, and spatial structure. Despite its smaller size compared to the SIFT10M and DEEP10M datasets, GIST1M remains a considerable challenge for vector search algorithms due to its high dimensionality.

Query Pattern and Workloads: The query vectors in three datasets are provided with their ground truth. The dimensionality (i.e., width) of the query vectors remains the same as their base vectors in Table 1, thus enabling us to find the approximate nearest neighbor via indices.

Table 1. Datasets

	SIFT10M	DEEP10M	GIST1M
# of Base Vectors	10,000,000	10,000,000	1,000,000
# of Query Vectors	10,000	10,000	1,000
Dimensionality	128	96	960
Data type	int32	float32	int32

To ensure a comprehensive evaluation, we test our system on various query workloads, including **sparse**, **bursty**, **continuous**, and **real-world** workloads. For the sparse workloads, the query workload generation proceeds as follows: In the first 5 minutes, we issue either 1000 qps or 4000 qps, then enter an idle period for the next t minutes (e.g., t equals 2 minutes), followed by another 5 minutes of issuing queries, and then another idle period of t minutes. This cycle repeats periodically. By default, we set the idle time t to 2 minutes for this workload. This is also the **default workload** used in many experiments such as Figure 7 and 14. But we also explore the impact of different idle times in Sec. 4.5.3. For the bursty workloads, we generate three synthetic workloads with low, medium, and high levels of burstiness as described in Sec. 4.5.1. The data generator is explained in Sec. 4.2. Additionally, we assess our system's performance on real-world workloads, as explained in Sec. 4.5.2.

Evaluation Metrics: To assess search efficiency and performance, both latency and accuracy are critical parameters. Query latency is evaluated by calculating the average execution time for sequentially executed queries in milliseconds across different minute-wise query workloads. As for accuracy, we conducted experiments with varying values of k. By default, k is 10. This presents a great challenge as it measures the accuracy of identifying the top 10 nearest neighbors among the base vectors, denoted as recall@10.

Experimental Environments: The configuration we use for the static VM baseline solution is Azure compute-optimized VM F4 [11]. It uses the exact same Intel(R) Xeon(R) E5-2673 and 8171M CPU [54] as Azure Function, and it also has a suitable 8GB memory to support the current vector search workload with appropriate disk storage. We have verified that the underlying system of Azure Function is able to produce highly optimized machine code utilizing SIMD and intrinsic instructions native to the hardware. In our experiments, we utilized the Numpy library which is well-optimized for SIMD, leveraging the AVX2 instruction set that enables data parallelism, meaning that a single instruction will operate on multiple data points at a time. This optimization significantly accelerates our compute-intensive workloads, which is critical to the efficiency and speed of our vector search process. The startup cost of a cloud function is approximately 3×10^{-6} USD each time, with the overhead of starting up being around 1-2 seconds, while the startup overhead of a VM is around 1-2 minutes. Azure Functions follows a pay-as-you-go pricing model and does not incur a fixed cost per instance per day.

Experimental Methodology: We compare our optimized strategies in Vexless with a complete HNSW index (built using the entire dataset) running vector searches on a cloud VM as the baseline. Specifically, due to the non-disjoint property of sharding in the baseline, every data shard directly returns a list of its top-k search results: global vector IDs and their distances to the query vector.

4.2 Workload Generator

To create synthetic workloads of various properties and simulate real-world workloads, we present our workload generator by controlling the query arrival time. It assigns dispatch times

to n jobs (i.e., queries) within a given time interval following the principle of signal decomposition [36], where signals can be synthesized by components of different frequencies. Specifically, our approach first divides the interval into g bins and then assigns the n jobs to each bin using an outer distribution; subsequently, each job is assigned its precise timestamp within the sub-interval (bin) using another set of inner probability distributions. The binning process allows for the flexibility of using different distributions for the lower frequency components to better model macro-level features [34, 36, 62]. Both the outer and inner distributions can be a combination of the following parametric distributions: Uniform, Gaussian, Zipf, and Poisson, which are used to model the specific features of the target workload. For example, the Zipf distribution is used to simulate bursts [30], the Gaussian distribution is widely used to model stochastic events such as network traffic [41, 62], and the Poisson distribution is effective for modeling the arrival times of jobs [22, 27]. We can adjust the composition of the distributions and parameters of each distribution, as well as the granularity of binning, based on the characteristics of the target workload, such as skew, smoothness, number of spikes, etc.

For a comprehensive evaluation of Vexless under a variety of workloads, we first generated workloads that resemble real-world scenarios, such as those found in an analytical database system [70] (Figure 11d) and a social network platform [75] (Figure 11e). Then, we generated workloads under three different bursty levels: high (Figure 11a), medium (Figure 11b), and low (Figure 11c).

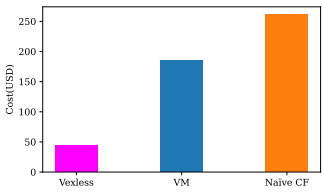
4.3 Overall Results

Our first experiment demonstrates the general search performance with cost analysis across different query arrival rate scenarios. As mentioned in Section 3, Vexless builds on Azure Functions, which adopts a pay-as-you-go billing model. This means users pay only for the computational resources their functions actually use. The cost is related not just to the function activation count, but also to the duration of time the function runs after activation. In contrast, when opting for a virtual machine, users are committed to a more rigid pricing scheme, bearing costs for predefined resource allocations charging 24/7, and irrespective of actual usage. We also compared our solution to another serverless approach: a naive cloud function-based method. This alternative offers a straightforward approach to implementing vector search on serverless cloud functions using a uniform-sized partition. However, it lacks the clustering and redundancy enhancements we discuss in Section 3.2.

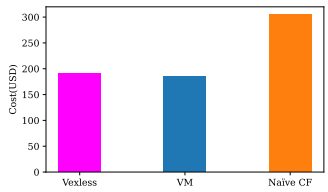
Our results, as shown in Figure 7 that follows the styles of [32, 45, 70], highlight Vexless's significant advantage in both the performance-to-cost ratio and latency across different datasets. For example, Figure 7(a, d, g) shows that under sparse workloads, Vexless offers monthly cost savings of up to **5.3x** compared to a statically provisioned VM-based solution and as much as **6.5x** when compared to naive cloud function implementations. In Figure 7(b, e, h) we see the Vexless continues to have significantly lower monthly costs across the board even under dense workloads. Furthermore, Figure 7(c, f, i) also shows that Vexless achieved the best search performance across three different datasets.

This demonstrates one of Vexless's key features – cost-efficiency. It optimally allocates serverless resources to guarantee the best search performance with cost-effective results.

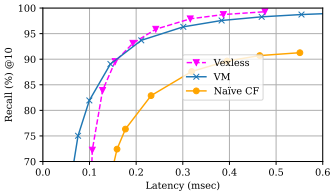
Moreover, we conducted tail latency analysis on vector search using different datasets to ensure consistent search performance. Our percentile latency analysis shows Vexless's reliability and consistency in vector searches on the DEEP10M dataset. Through a comparative figure, we observed that Vexless consistently outperforms a VM-based benchmark across average, 95th, and 99th percentiles shown in Figure 8.



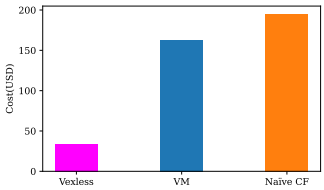
(a) SIFT10M: Monthly Cost, Sparse Queries



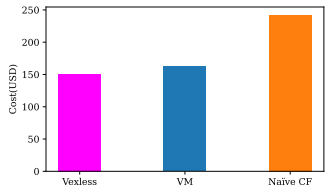
(b) SIFT10M: Monthly Cost, Dense Queries



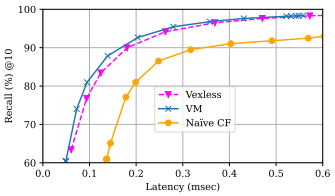
(c) SIFT10M: Search Recall@10 Performance



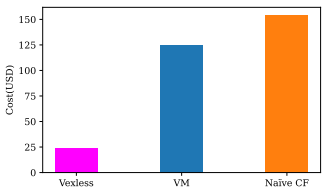
(d) DEEP10M: Monthly Cost, Sparse Queries



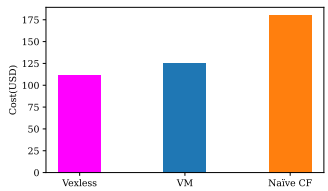
(e) DEEP10M: Monthly Cost, Dense Queries



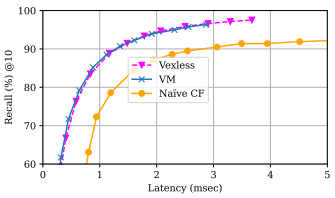
(f) DEEP10M: Search Recall@10 Performance



(g) GIST1M: Monthly Cost, Sparse Queries



(h) GIST1M: Monthly Cost, Dense Queries



(i) GIST1M: Search Recall@10 Performance

Fig. 7. The Left Two Columns Show the Monthly Cost of Vector Search with the Same Parameters under Sparse (a, d, g) and Dense (b, e, h) Workloads using Different Implementations: Vexless, Traditional Cloud Virtual Machines (the Cost Was Reduced by Memory Usage, with 100% CPU Cores Used for Intra-query Parallelism), and Naive Implementation with Cloud Functions. The Right Column Shows the Recall-Latency Graph of the Three Implementations. Each Row Shows Results from Different Datasets (SIFT10M, DEEP10M, and GIST1M)

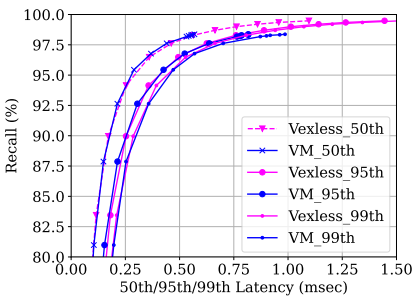
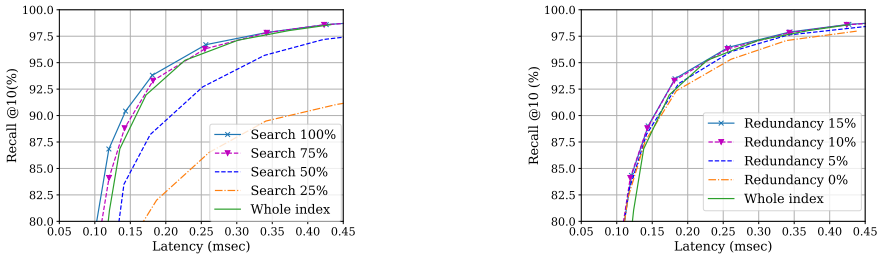


Fig. 8. The Percentile Search Latency Comparison of Vexless and Multi-threaded VM on Deep10M



(a) Partition Search Strategy Analysis on DEEP10M **(b)** Indexing Redundancy Analysis on DEEP10M

Fig. 9. The Search Strategy and Redundancy Analysis in Vexless

4.4 Ablation Study

Next, we will justify each part of our design by conducting related performance evaluations on the DEEP10M dataset. We also analyze the cost breakdown of different components of Vexless, including the techniques in indexing, vector search, communication, and cold-start optimization respecting different query distributions and diverse datasets.

Our search strategy is observed with search performance in distinct function activation levels illustrated in Figure 9a. We distributed the DEEP10M data index on four partitions, then conducted distributed vector searches by dispatching the search task to the top-p (p = 100%, 75%, 50%, 25%) closest index partitions and activating them for one query search. The "Whole Index" represents the baseline where the HNSW Index is built as a whole. By searching more index partitions, the performance gets better due to larger search scopes. Moreover, we discovered that compared with searching all the partitions, searching the top-75% closest index partitions not only saves a quarter of the search cost but also has a comparable search performance with the former top-100% search strategy. This observation brings us more reference for setting distance thresholds when selecting search partitions.

On the other hand, in our vector data sharding strategy, we observed that some vectors, as mentioned in Section 3.2 which we term "boundary vectors" or "boundary points", were at nearly equal distances from multiple centroids. After managing to solve such a problem, with variable memory redundancy ratios, Vexless showcased search efficiencies on datasets such as DEEP10M Figure 9b. The percentage in the figure legend represents the overall memory consumption increased by introducing redundant boundary vectors when indexing the same data range.

With indexing that trades off redundancy for search performance, we looked into the indexing and activation threshold in the sharding and searching phase, our tests show that sharding with diverse redundant options that have different distance thresholds would also have great effects on the search efficiency, partly because of affecting the activation ratio of serverless cloud functions and partly because of the boundary points was more included in the indexing and further search. Thus, establishing the right activation threshold is important. Our observations of Vexless offer a flexible performance range with diverse threshold settings in Figure 10, the figure displays how efficiency changes as the distance to cluster centroids varies. Clusters are selected for search only if the distance between the query vector and centroid is below the specified threshold. In the real practice of vector search on DEEP10M, we choose the threshold as T=0.95, with the memory footprint only increasing about ten percent more than the whole HNSW index.

4.5 Additional Experiments

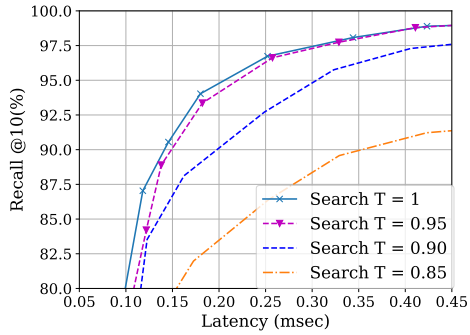


Fig. 10. Search Efficiency Evaluation Under Varying Distance Thresholds

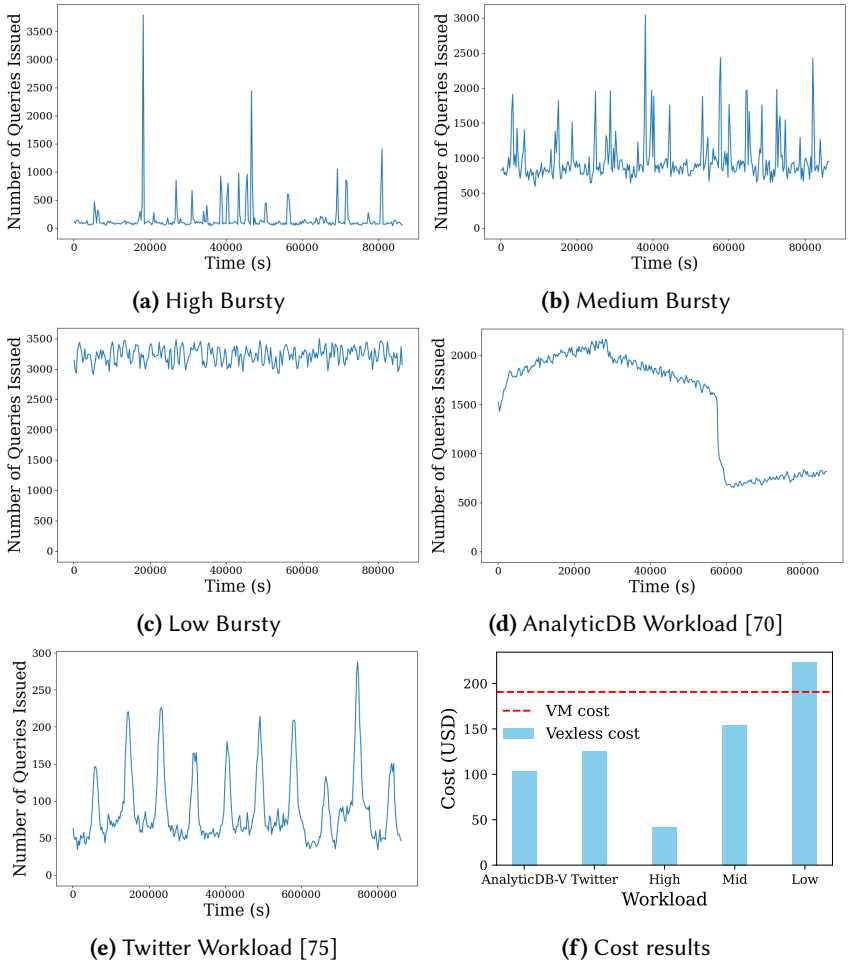


Fig. 11. Results on Bursty and Real Workloads

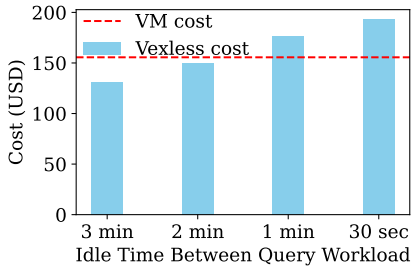


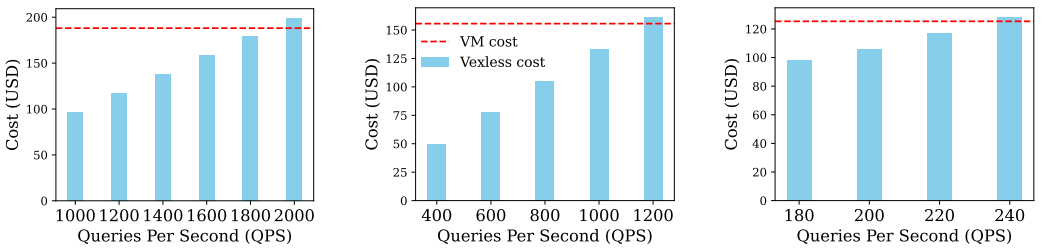
Fig. 12. Evaluating the Impact of Sparseness (by Controlling Idle Time)

4.5.1 Experiments on Bursty Workloads. In this experiment, we evaluate the impact of burstiness. We generated three synthetic workloads with low, medium, and high levels of burstiness in Figure 11a, 11b, 11c. We explained the data generator in Sec. 4.2. The experimental results can be found in Figure 11f. It shows that cloud functions achieve lower costs on medium- and high-burst workloads (given similar recall and performance), as expected.

4.5.2 Experiments on Real Workloads. In this experiment, we added two real-world workloads presented in [70] and [75]. As those workloads are not publicly available, we generated two workloads using our workload generator (Sec. 4.2), see Figure 11d and 11e. We have verified that the query patterns of our generated workloads are consistent with those of the two real-world workloads. The experimental results can be found in Figure 11f. **It shows that on real workloads, cloud functions win because the workloads are indeed bursty and sparse**.

4.5.3 Evaluating the Impact of Sparseness. Recall that in Figure 7, we issued queries in a periodic pattern of 5 minutes active followed by 2 minutes idle. In this experiment, we varied the idle time. Figure 12 shows the results on DEEP10M dataset (assuming the query rate is 4000qps). It shows that when the idle time exceeds 2 minutes, cloud functions are more cost-effective.

4.5.4 Experiments on Continuous Workloads. We conducted a new experiment to evaluate the cost of continuously issuing queries at different qps rates, see the results in Figure 13. It shows that for the Deep10M dataset, as long as the continuous query rate is below 1000 qps (the query rate of a moderate website [19]), cloud functions win.



(a) SIFT10M **(b)** Deep10M **(c)** GIST1M

Fig. 13. Results on Continuous Workloads

Table 2. Re-ranking Latency for Varying k

k	1	10	50	100	1000
Re-ranking (µs)	2.24	2.32	4.77	9.53	85.83

4.5.5 Varying k. Next, we evaluate the impact of k in top-k vector search. In Figure 14, which shows different k values corresponding to diverse search scenarios, the search performance and cost comparison results indicate that Vexless has a better cost and performance advantage in all aspects. As k grows, the search accuracy leads to an even larger performance gap and a significant cost advantage compared to the baseline.

Furthermore, we also added a new experiment to show the re-ranking time for different k values, as shown in Table 2. It shows that the re-ranking cost is in the order of microseconds (while vector search is in the order of milliseconds).

4.5.6 Case Study. In this experiment, we tested Vexless in an end-to-end machine learning application. We chose Image Search as it is a popular application for vector search [68].

Image Search consists of two steps in online query processing. First, we convert the query image S into a feature vector V_S using a pre-trained neural network. Second, we search this vector V_S in a vector database $\mathbf{R}$. This database stores feature vectors corresponding to images in a web-scale repository, potentially containing billions of images. We obtain the vector database $\mathbf{R}$ offline. Figure 15 shows this entire workflow.

For the embedding model, we chose MobileNetV3 [61], known for its high-quality embeddings and high throughput. We run this model on a server equipped with Tensorflow to encode the dataset in batches. The server includes an Intel Xeon Platinum 8364 CPU and two NVIDIA A40 GPUs. We selected a pre-processed subset of ImageNet21 [33] with 1 million images for the dataset. The images are embedded into 1000-dimensional vectors, and we use HNSW as the vector index.

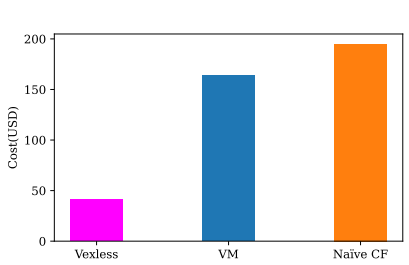
The result shows that the image embedding time (step 1 in Figure 15) is around 0.18ms per image on average, while the vector search (step 2 in Figure 15) takes 2ms. The vector search task costs 10 times more than image embedding and is clearly the bottleneck in this application. This is because the embedding task uses a static model and is highly parallelizable since each embedding process is independent of the others, and therefore, it is often accelerated with specialized hardware such as GPUs, TPUs, and FPGAs. This further confirms the necessity of optimizing vector search performance in large-scale vector search applications.

5 RELATED WORK

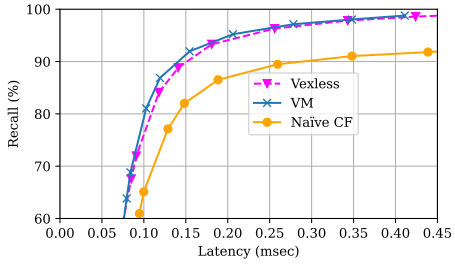
While we have covered many related techniques in Sec. 2 and Sec. 3, in this section, we briefly emphasize that there are two lines of work relevant to this paper. First, there are many vector databases, such as Milvus [68], Pinecone [16], Vespa [18], and pgvector [15]. For a survey, refer to [56, 57]. However, these databases are not built using cloud functions. Although Pinecone recently released a serverless version of a vector database, it does not utilize cloud functions. Another line of work involves leveraging cloud functions for building data systems, such as data analytics [28, 55, 59], and machine learning [67, 71]. However, they do not focus on building vector databases, which is the main focus of this work.

6 CONCLUSION

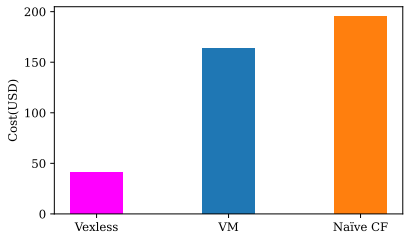
In this paper, we presented Vexless, the first vector database built for cloud functions with the benefits of high elasticity, low operational cost, and fine-grained billing model. Experiments show



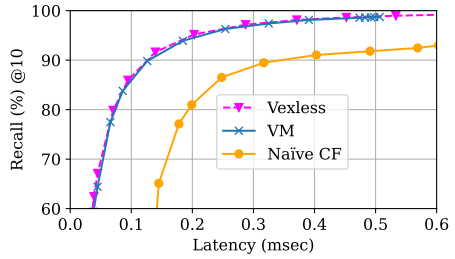
(a) Search Cost at k = 1



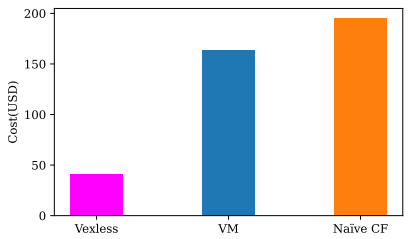
(b) Recall@1



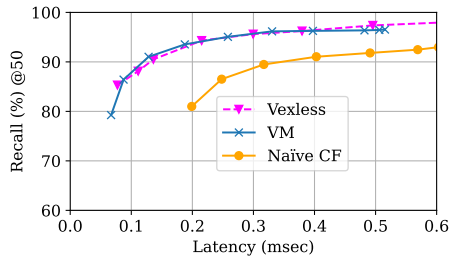
(c) Search Cost at k = 5



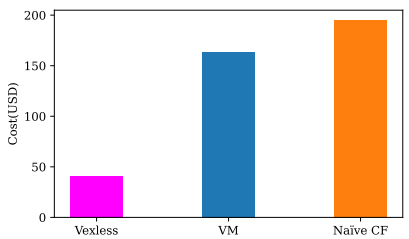
(d) Recall@5



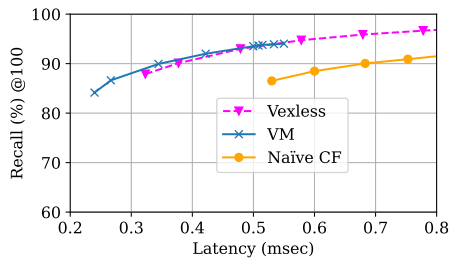
(e) Search Cost at k = 50



(f) Recall@50



(g) Search Cost at k = 100



(h) Recall@100

Fig. 14. The Monthly Cost and Search Performance of Three Implementations under Varying k

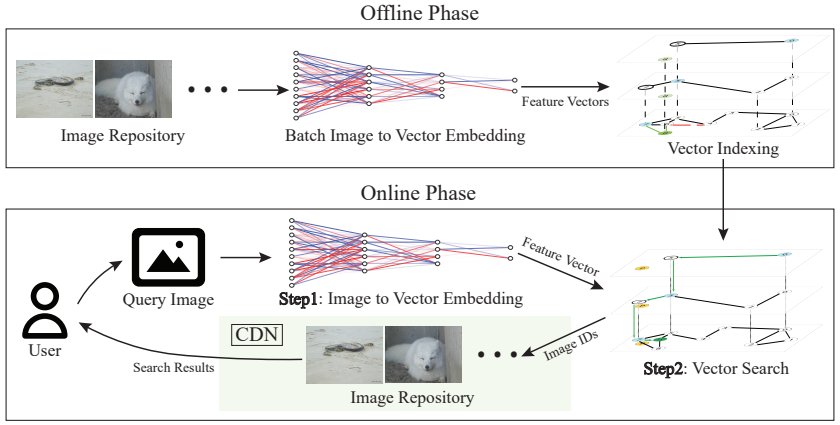


Fig. 15. Workflow of an Image Search System

that Vexless achieves better cost-efficiency on sparse and bursty workloads than traditional VM-based implementation. Vexless is also flexible such that different index methods including IVF and LSH can be used, and our design can also be generalized to other cloud platforms such as AWS and GCP.

ACKNOWLEDGEMENTS

Jianguo Wang acknowledges the support of the National Science Foundation under Grant Number 2337806. We are also grateful for the generous support from Microsoft on Azure credits.

REFERENCES

[1] [n. d.]. Alibaba Cloud: Manage Stateful Asynchronous Invocations. https://www.alibabacloud.com/help/en/fc/developer-reference/manage-stateful-asynchronous-invocations.
[2] [n. d.]. Alibaba Cloud: Message Service (MNS). https://www.alibabacloud.com/product/message-service.
[3] [n. d.]. Amazon Simple Queue Service. https://aws.amazon.com/sqs.
[4] [n. d.]. AWS Lambda - Serverless Compute - Amazon Web Services. https://aws.amazon.com/lambda.
[5] [n. d.]. AWS Step Functions. https://aws.amazon.com/step-functions.
[6] [n. d.]. Azure Functions - Serverless Code. https://azure.microsoft.com/services/functions.
[7] [n. d.]. Azure Functions Scale and Hosting. https://learn.microsoft.com/azure/azure-functions/functions-scale.
[8] [n. d.]. Benchmarks for Billion-Scale Similarity Search. https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search.
[9] [n. d.]. Cloud Functions: Serverless Computing, Google Cloud. https://cloud.google.com/functions.
[10] [n. d.]. Cold Starts in Azure Functions. https://mikhail.io/serverless/coldstarts/azure.
[11] [n. d.]. Compute Optimized F Series - Azure Virtual Machines. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-compute.
[12] [n. d.]. Google Cloud Pub/Sub. https://cloud.google.com/pubsub.
[13] [n. d.]. Microsoft Azure Durable Functions. https://learn.microsoft.com/azure/azure-functions/durable/durable-functions-overview.
[14] [n. d.]. Microsoft Azure Queue Storage. https://learn.microsoft.com/azure/storage/queues/storage-queues-introduction.
[15] [n. d.]. pgvector. https://github.com/pgvector/pgvector.
[16] [n. d.]. Pinecone: Vector Database for Vector Search. https://www.pinecone.io.
[17] [n. d.]. Scalability and Performance Targets for Blob storage. https://learn.microsoft.com/azure/storage/blobs/scalability-targets.
[18] [n. d.]. Vespa (https://vespa.ai/).
[19] [n. d.]. What's the "Average" Requests Per Second for a Production Web Application? https://stackoverflow.com/questions/373098/whats-the-average-requests-per-second-for-a-production-web-application.

[20] 2023. LLM Limitations. (https://zilliz.com/use-cases/llm-retrieval-augmented-generation).
[21] 2023. Solving ChatGPT Hallucinations With Vector Embeddings https://www.youtube.com/watch?v=FUgp4oaxj-M.
[22] Ryan Prescott Adams, Iain Murray, and David JC MacKay. 2009. Tractable Nonparametric Bayesian Inference in Poisson Processes with Gaussian Process Intensities. In *International Conference on Machine Learning (ICML)*. 9–16.
[23] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful Functions as a Service in Action. *Proceedings of the VLDB Endowment (PVLDB)* 12, 12 (2019), 1890–1893.
[24] Alexandr Andoni and Piotr Indyk. 2008. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. *Commun. ACM* 51, 1 (2008), 117–122.
[25] Artem Babenko and Victor Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2055–2063.
[26] Oren Barkan and Noam Koenigstein. 2016. Item2Vec: Neural Item Embedding for Collaborative Filtering. In *International Workshop on Machine Learning for Signal Processing (MLSP)*. 1–6.
[27] Vandy Berten, Joël Goossens, and Emmanuel Jeannot. 2006. On the Distribution of Sequential Jobs in Random Brokering for Heterogeneous Computational Grids. *IEEE Transactions on Parallel and Distributed Systems* 17, 2 (2006), 113–124.
[28] Haoqiong Bian, Tiannan Sha, and Anastasia Ailamaki. 2023. Using Cloud Functions as Accelerator for Elastic Data Analytics. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 2 (2023), 161:1–161:27.
[29] Paul S Bradley, Kristin P Bennett, and Ayhan Demiriz. 2000. Constrained K-means Clustering. *Microsoft Research, Redmond* 20, 0 (2000), 0.
[30] Mudashiru Busari and Carey Williamson. 2002. ProWGen: a Synthetic Workload Generation Tool for Simulation Evaluation of Web Proxy Caches. *Computer Networks* 38, 6 (2002), 779–794.
[31] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. *SPTAG: A Library for Fast Approximate Nearest Neighbor Search. https://github.com/Microsoft/SPTAG.*
[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 *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*. 5199–5212.
[33] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A Large-Scale Hierarchical Image Database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 248–255.
[34] Zhaoxu Ding, Guoqiang Zhong, Xianping Qin, Qingyang Li, Zhenlin Fan, Zhaoyang Deng, Xiao Ling, and Wei Xiang. 2024. MF-Net: Multi-frequency Intrusion Detection Network for Internet traffic Data. *Pattern Recognition* 146 (2024), 109999.
[35] Matthijs Douze, Hervé Jégou, Harsimrat Sandhawalia, Laurent Amsaleg, and Cordelia Schmid. 2009. Evaluation of GIST Descriptors for Web-Scale Image Search. In *Proceedings of the ACM International Conference on Image and Video Retrieval (CIVR)*. 1–8.
[36] Thomas Eriksen and Naveed ur Rehman. 2023. Data-driven Nonstationary Signal Decomposition Approaches: a Comparative Analysis. *Scientific Reports* 13, 1 (2023), 1798.
[37] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. 2006. High Dimensional Nearest Neighbor Searching. *Information Systems* 31, 6 (2006), 512–540.
[38] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proceedings of the VLDB Endowment (PVLDB)* 12, 5 (2019), 461–474.
[39] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *International Conference on Very Large Data Bases (VLDB)*. 518–529.
[40] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 33, 1 (2011), 117–128.
[41] Dingde Jiang, Zuyao Zhao, Zhengzheng Xu, Chunping Yao, and Hongwei Xu. 2014. How to Reconstruct End-to-end Traffic Based on Time-frequency Analysis and Artificial Neural Network. *AEU-International Journal of Electronics and Communications* 68, 10 (2014), 915–925.
[42] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.
[43] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019).
[44] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *International Conference on Machine Learning (ICML)*. 1188–1196.

[45] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. 2020. Improving Approximate Nearest Neighbor Search Through Learned Adaptive Early Termination. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2539–2554.

[46] Yuliang Li, Jianguo Wang, Benjamin S. Pullman, Nuno Bandeira, and Yannis Papakonstantinou. 2019. Index-Based, High-Dimensional, Cosine Threshold Querying with Optimality Guarantees. In *International Conference on Database Theory (ICDT)*, Vol. 127. 11:1–11:20.

[47] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. 2023. FaaSLight: General Application-Level Cold-Start Latency Optimization for Function-as-a-Service in Serverless Computing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 32, 5 (2023).

[48] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. 2020. VHP: Approximate Nearest Neighbor Search via Virtual Hypersphere Partitioning. *Proceedings of the VLDB Endowment (PVLDB)* 13, 9 (2020), 1443–1455.

[49] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *Proceedings of the VLDB Endowment (PVLDB)*. 950–961.

[50] Yury A. Malkov and Dmitry A. Yashunin. 2018. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)* 42, 4 (2018), 824–836.

[51] R. B. MARIMONT and M. B. SHAPIRO. 1979. Nearest Neighbour Searches and the Curse of Dimensionality. *IMA Journal of Applied Mathematics* 24, 1 (1979), 59–70.

[52] Erik Bernhardsson Martin Aumueller. 2023. ANN-Benchmarks. https://ann-benchmarks.com.

[53] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *International Conference on Learning Representations (ICLR)*.

[54] Diego Montes, Juan A Añel, David CH Wallom, Peter Uhe, Pablo V Caderno, and Tomás F Pena. 2020. Cloud Computing for Climate Modelling: Evaluation, Challenges and Benefits. *MDPI Computers* 9, 2 (2020), 52.

[55] Ingo Müller, Renato Marroquin, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 115–130.

[56] James Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Companion of the International Conference on Management of Data (SIGMOD)*.

[57] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of Vector Database Management Systems. *CoRR* abs/2310.14021 (2023).

[58] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. 2023. iQAN: Fast and Accurate Vector Search with Efficient Intra-Query Parallelism on Multi-Core Architectures. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 313–328.

[59] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 131–141.

[60] Florin Pop, Ciprian Dobre, Valentin Cristea, and Nik Bessis. 2013. Scheduling of Sporadic Tasks with Deadline Constrains in Cloud Environments. In *International Conference on Advanced Information Networking and Applications (AINA)*. 764–771.

[61] Siying Qian, Chenran Ning, and Yuepeng Hu. 2021. MobileNetV3 for Image Classification. In *International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*. 490–497.

[62] Rudolf H Riedi, Matthew S Crouse, Vinay J Ribeiro, and Richard G Baraniuk. 1999. A Multifractal Wavelet Model with Application to Network Traffic. *IEEE Transactions on Information Theory* 45, 3 (1999), 992–1018.

[63] Chanop Silpa-Anan and Richard I. Hartley. 2008. Optimised KD-trees for Fast Image Descriptor Matching. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–8.

[64] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the International Middleware Conference*. 1–13.

[65] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 13748–13758.

[66] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9.

[67] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 495–514.

[68] 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. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2614–2627.

[69] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *USENIX Annual Technical Conference (USENIX ATC)*. 133–146.

[70] 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. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (2020), 3152–3165.

[71] Yuncheng Wu, Tien Tuan Anh Dinh, Guoyu Hu, Meihui Zhang, Yeow Meng Chee, and Beng Chin Ooi. 2022. Serverless Data Science - Are We There Yet? A Case Study of Model Serving. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1866–1875.

[72] Huafeng Xi, Jianfeng Zhan, Zhen Jia, Xuehai Hong, Lei Wang, Lixin Zhang, Ninghui Sun, and Gang Lu. 2011. Characterization of Real Workloads of Web Search Engines. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. 15–25.

[73] Zhengjun Xu, Haitao Zhang, Xin Geng, Qiong Wu, and Huadong Ma. 2019. Adaptive Function Launching Acceleration in Serverless Computing Platforms. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 9–16.

[74] Yunan Zhang, Shige Liu, and Jianguo Wang. 2024. Are There Fundamental Limitations in Supporting Vector Data Management in Relational Databases? A Case Study of PostgreSQL. In *International Conference on Data Engineering (ICDE)*.

[75] Zhe Zhao and Qiaozhu Mei. 2013. Questions about Questions: An Empirical Analysis of Information Needs on Twitter. In *International World Wide Web Conference (WWW)*. 1545–1556.

Received October 2023; revised January 2024; accepted March 2024