# Building Efficient Serverless Vector DBs via Block-based Data Partitioning: [Experiments & Analysis]

Anonymous Author(s)

## Abstract

Retrieval-Augmented Generation (RAG) and other AI/ML workloads rely heavily on vector databases (vector DBs) for efficient analysis of unstructured data. However, cluster vector DB architectures, such as Milvus, lack the elasticity properties required to handle high workload fluctuations, sparsity, and burstiness. Serverless vector DBs have recently emerged as a promising alternative architecture, but they are still in their infancy. In this paper, we provide the first characterization of serverless vector DBs. Moreover, we identify that the state-of-the-art serverless vector DB —*i.e.*, Vexless [28]— relies on clustering-based data partitioning (*i.e.*, balanced K-means with vector redundancy). Via extensive experiments, we evaluate the limitations of such a data partitioning scheme and propose a block-based data partitioning alternative. We empirically demonstrate that a serverless vector DB using block-based data partitioning achieves comparable performance to a cluster vector DB (Milvus) —indexing time, query recall— at a fraction of the cost for sparse workloads.

## CCS Concepts

• **Information systems** → **Data management systems**; **Search engine architectures and scalability**; • **Computer systems organization** → **Cloud computing**.

## Keywords

Vector Databases, Cloud Functions, Serverless Computing, Data Partitioning, Indexing

**ACM Reference Format:**

Anonymous Author(s). 2025. Building Efficient Serverless Vector DBs via Block-based Data Partitioning: [Experiments & Analysis]. In *Proceedings of 2026 International Conference on Management of Data (SIGMOD '26).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

The exponential growth of unstructured data has created a pressing need to leverage AI and machine learning (ML) techniques to unlock its potential. Modern AI workloads, such as those using Retrieval-Augmented Generation (RAG) models, rely heavily on generating *vector embeddings* [11] from unstructured data (*e.g.*, text, images, and audio) to enable efficient and meaningful analysis. These embeddings convert complex, high-dimensional data into lower-dimensional vectors that preserve semantic relationships, facilitating advanced tasks like natural language processing [23], image recognition [14], and recommendation systems [34]. Consequently, the ability to manage vector embeddings is key for extracting value from unstructured data in AI/ML applications.

The need for managing vector embeddings has led to the rising popularity of *vector databases* (vector DBs) [12, 24, 31, 32]. Vector DBs are specialized systems designed to store and retrieve high-dimensional vectors efficiently [27]. They enable fast and accurate similarity search, making them essential for AI and machine learning applications. In this sense, Pinecone [24], Weaviate [32], and Milvus [12, 31] are, among others, examples of popular vector DBs used at scale in production analytics pipelines.

### 1.1 Motivation

Despite the specialization of their search and indexing algorithms, most distributed vector DBs rely on a traditional *cluster architecture* [21, 27]. This requires that administrators carefully manage the resources of the vector DB deployment. Under fluctuating or sparse query workloads, this architecture can result in periods of over-provisioning —leading to higher costs— and under-provisioning —causing saturation. Likewise, adapting such an architecture to handle high workload burstiness is challenging.

In response to such architectural limitations, there is an emerging trend for porting vector DBs to the *serverless paradigm*. In the industry, the main objective seems to be simplifying the provisioning of vector DBs and offering them "as-a-service" —and, in some cases, applying a pay-per-query model. Instantiations of this trend include Weaviate Serverless Cloud [32], Upstash [30], and Amazon OpenSearch Service as a Vector DB [1], to name a few.

However, the true realization of a serverless vector DB architecture goes far beyond simplified provisioning: it lies in distributing the vector DB engine across *cloud functions*. Function-as-a-Service (FaaS) providers, such as AWS Lambda [2], Azure Functions [4], or Google Cloud Functions [10], deliver a powerful substrate for executing embarrassingly parallel workloads. Crucially, such a design has the potential to overcome the elasticity and burstiness challenges that cluster vector DB architectures typically face.

Serverless vector DBs must distribute tasks across parallel cloud functions that are normally executed on compute nodes in cluster vector DBs. These tasks include: i) *data ingestion*, which consists on storing new vector embeddings in the system; ii) *data partitioning*, which involves distributing a collection of vector embeddings into smaller pieces (*e.g.*, data objects in AWS S3); iii) *data indexing*, which generates indexes for efficient vector lookups within dataset partitions; and iv) *querying*, which performs similarity searches on vectors across dataset partitions. While promising, building a serverless vector DB presents unique challenges from a design perspective compared to cluster vector DBs (see §3).



*SIGMOD '26, May 31 - June 05, 2026, Bengaluru, India*



ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

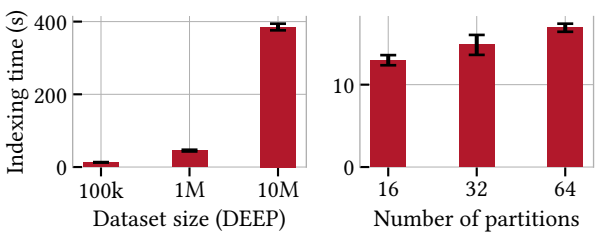


**Figure 1: Indexing time of DEEP dataset for different volumes (in 16 partitions) and number of partitions (100k vectors) via K-means based data partitioning in AWS Lambda.**

## 1.2 Dynamic Data in Serverless Vector DBs

The use of cloud functions as the core engine for a vector DB, as pioneered by Vexless [28], has shown promise in optimizing query performance. However, the approach of Vexless is limited to *static datasets* that are partitioned and indexed in advance. Concretely, it relies on a *clustering-based data partitioning* scheme —a bounded K-means with vector redundancy— to build a *global centroid-based index* to trade off data filtering and query recall. Unfortunately, Vexless's static approach to data partitioning and indexing may not be well-suited for real-world applications with dynamic datasets.

A key insight of this work is to empirically show that clustering-based data partitioning in serverless vector DBs induces significant costs that may hinder their feasibility. While full details are provided in §6, Fig. 1 depicts the processing time of partitioning and indexing a vector dataset via K-means in AWS.[1] Visibly, data partitioning and indexing exhibit a growing processing time depending on both the number of centroids and the dataset size. This means that the stateful and global properties of clustering-based data partitioning cannot fully exploit the parallelism of cloud functions. Even worse, due to the nature of K-means, inserts to the dataset may imply re-partitioning and re-indexing, hence involving high processing costs. This calls for exploring alternative data partitioning schemes for serverless vector DBs that efficiently handle dynamic data.

## 1.3 Contributions

Serverless vector DBs must handle dynamic datasets. To this end, a core contribution of this paper is to identify the key limitations of clustering-based data partitioning for this new family of systems and evaluate a block-based data partitioning alternative. Moreover, we empirically show that block-based data partitioning in serverless vector DBs enable competitive system designs compared to a cluster vector DB. In summary, our contributions in this paper are:

- We present an overview of serverless vector DBs, highlighting their key components and design trade-offs. Given their recent emergence, this work is the first to offer a timely overview of this new family of systems (see §3).
- We identify the main limitations of using a clustering-based data partitioning scheme for serverless vector DBs, as it is the approach used in the state-of-the-art (see §4).
- We provide an experimental comparison of clustering-based data partitioning with a simple, yet practical, block-based data partitioning scheme for serverless vector DBs (see §6).
- We show through experimentation that a serverless vector DB using block-based data partitioning is competitive with a cluster vector DB (Milvus) in terms of indexing time, query latency, recall, and economic costs (see §7).

Our experimental analysis of a serverless vector DB prototype on AWS Lambda shows that our block-based data partitioning outperforms clustering-based schemes in terms of partitioning performance ($x$% to $y$%), query times ($x$% to $y$%), and cost ($x$% to $y$%). Also, we find that techniques proposed to enhance clustering-based data partitioning [28], such as balancing data partitions and vector redundancy, may not offer clear benefits overall. Finally, when comparing our prototype to Milvus, we achieve better partitioning times ($x$% to $y$%), similar query recalls, and an acceptable overhead in query times that translate into cost reduction ($x$% to $y$%).

The rest of the paper is as follows. We provide key background in §2. We present a timely overview of serverless vector DBs in §3. In §4, we analyze the data partitioning trade-offs in serverless vector DBs and, after introducing our experimental framework (see §5), we show an empirical evaluation of clustering-based data partitioning versus block-based data partitioning (§6). In §7, we compare our block-based serverless vector DB implementation against Milvus. We revisit the related work in §8 and conclude the paper in §9.

# 2 Background

In this section, we provide the necessary background to understand the remainder of the paper: FaaS and vector DBs.

## 2.1 Function-as-a-Service (FaaS)

The *serverless computing paradigm* is a cloud computing execution model where developers build and deploy applications without managing the underlying infrastructure [26]. Instead, cloud providers handle the provisioning, scaling, and maintenance of servers dynamically under the hood. This approach enhances agility, reduces operational complexity, and optimizes resource utilization by charging users only for the actual execution of their code.

The most prominent instantiation of serverless computing is *Function-as-a-Service* (FaaS) [2, 4], which allows developers to write modular functions that are executed in response to specific events, such as direct HTTP requests, database changes, messages in a queue, or other triggers. FaaS follows an *event-driven programming model*, where function invocations are ephemeral, stateless, and designed to perform a single task independently. This allows providers to offer them in a *pay-per-use billing* model, where charges are based on the number of function invocations and the execution duration, typically measured in milliseconds. This model differs from traditional server-based pricing, in which instances are provisioned and billed at a coarser granularity and incurs costs for idle resources. Moreover, developers focus solely on writing business code (functions) rather than managing infrastructure. This is key for improving developer productivity and accessibility to cloud environments for non-expert users.

However, the FaaS model also comes with limitations, such as function cold start latency (*i.e.*, start-up delay after a period of

[1] We set up a favorable scenario in which K-means is executed on a `c7i.xlarge` EC2 VM whereas data partitions are indexed by parallel cloud functions.

inactivity) [16, 19] and execution time limits. Furthermore, the ephemeral nature of FaaS models hinders inter-function communication [5]. Therefore, building a serverless vector DB on top of a FaaS service involves unique design challenges.

## 2.2 Vector DBs

The operation of any database consists of two phases: (i) *Ingestion and storage of data*, which is stored with different schemas depending on the database; (ii) *Querying*, where the database performs efficient lookups on the stored data. The architecture of vector DBs is similar to general-purpose ones. However, vector DBs have schemas specialized for storing high-dimensional vector data from unstructured data like text, sound, or images. The query phase also differs, as vector DBs store large volumes of data, making exact search computationally expensive. Instead, search algorithms based on similarity metrics (*e.g.*, cosine similarity, euclidean product) are used to trade-off query recall and latency [33].

When the collection of stored vector embeddings grows, vector DBs face two main scalability-related challenges: (i) the index(es) must be recomputed when the stored data changes, and (ii) as the number of vectors grows, searching becomes slow. For this reason, in recent years, a new generation of *cluster vector DBs* have emerged to address these challenges via a distributed architecture and parallel data management. In a cluster vector DB, the different tasks are performed by individual services that can be scaled independently. Moreover, a vector dataset is split into chunks, so vector DB tasks can be executed in parallel. Interestingly, this architecture shift introduces a new dimension to the vector DB operation that is the primary focus of this paper: *data partitioning*.

One instance of cluster vector DB is Milvus [12, 31]. It scales each process independently as services, with nodes (servers or VMs) assigned to specific roles: Data, Index, or Query. Data nodes are responsible for ingestion, partitioning, and storage management. Index nodes create indexes for data chunks, while Query nodes load these indexes and data into memory to respond to queries. Additionally, specialized nodes such as Query Coordinators and Data Coordinators aggregate results and manage data flows, including load balancing. Although services are scaled independently, the stateful nature of this cluster architecture can be rigid and difficult to adapt to rapid workload changes. It also requires an administrator to carefully right-size the deployment. Solving such elasticity and provisioning issues is a key goal of serverless vector DBs.

# 3 Serverless Vector DBs: An Overview

Building a vector DB on top a FaaS service is an emerging trend. In this section, we contribute a general description of the architecture and design trade-offs of this new family of systems (see Fig. 2).

## 3.1 Architecture

A serverless vector DB retains the core components of a cluster vector DB but leverages serverless compute services for elasticity and cost efficiency. A serverless vector DB consists of two service types: *data services* for ingestion and storage, and *compute services* for partitioning, indexing, and querying. The *data ingestion component* handles vector inserts, supporting both batch and streaming data through event brokers (*e.g.*, Kafka, AWS Kinesis) or event-driven

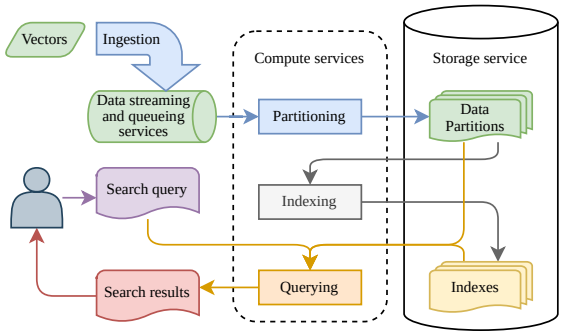


**Figure 2: Key processes of a serverless vector DB.**

pipelines (*e.g.*, AWS Lambda with S3 triggers). Once vectors are received, they are *partitioned and preprocessed* (*e.g.*, k-means clustering) before being stored in long-term storage, which typically consists of object storage for raw vector data partitions (*e.g.*, S3).

Data partition *indexing* is a crucial process performed by index generator functions, which transform vector partitions into efficient approximate nearest neighbor (ANN) search structures. Common indexing techniques include tree-based (KD-Tree), graph-based (HNSW), and quantization-based (IVF-PQ) approaches, enabling fast vector retrieval. These indexes, along with the raw vector data, are stored in scalable object storage systems, ensuring durability and high availability. Since indexing workloads are intermittent, FaaS services are well-suited for dynamically executing index generation without maintaining persistent compute resources.

Query execution in a serverless vector DB is distributed and parallelized across multiple partitions. Typically, a *query coordinator* orchestrates the execution of serverless functions responsible for retrieving candidate vectors from different partitions. These functions compute similarity scores (*e.g.*, cosine similarity, Euclidean distance) and return partial results that are then aggregated using a *reduce function*. Given the stateless nature of cloud functions, query coordination mechanisms (*e.g.*, Step Functions) are essential for synchronizing results. The final ranked vectors are returned to the user and functions terminate their execution.

By combining cloud functions with scalable storage, a serverless vector DB eliminates the need for persistent infrastructure while maintaining high-performance search capabilities. Compared to cluster architectures, this approach automatically scales with workload demands, optimizes cost through pay-as-you-go billing, and reduces operational overhead. This makes serverless vector databases a compelling choice for applications requiring efficient and scalable similarity search, such as recommendation systems, image retrieval, and AI-powered analytics.

## 3.2 Vector DB Design: Cluster vs Serverless

Next, we discuss the architectural differences between a popular cluster vector DB (Milvus/Manu) and existing serverless counterparts (Vexless, this paper). In Table 1, the systems compared represent different architectural approaches to vector DB management.

Milvus/Manu is a popular example of a cluster vector DB architecture designed to manage vector embeddings as a service. One

**Table 1: Comparison of Vector DB architectures.**

| Dimension | Milvus [31]/Manu [12] | Vexless [28] | This work |
|---|---|---|---|
| *Architecture* | Cluster | Serverless (*Stateful*, Azure Durable Functions) | Serverless (*Stateless*, AWS Lambda) |
| *Communication* | Distributed coordination (gRPC) | Stateful functions with message passing | Object storage-based communication |
| *Elasticity* | Horizontal node scaling | Automatic function scaling | Automatic function scaling |
| *Operation* | Node management/maintenance | Automatic function provisioning | Automatic function provisioning |
| *Billing* | Node/hour based | Pay per function invocation | Pay per function invocation |
| *Data Ingestion* | Data nodes (streaming-based) | Not supported (static datasets) | Supported (data object ingestion) |
| *Data Partitioning* | Shard-based (dynamic sharding with load balancing) | Clustering-based (balanced K-means with redundancy) | Block-based (fixed-size blocks with parallel indexing) |
| *Data Indexing* | Per shard segment on Index nodes (multi-algorithm) | Per cluster on cloud functions* (HNSW) | Per block on cloud functions (IVF) |
| *Querying* | Query nodes (similarity/multi-vector search, attribute filtering) | Cloud functions on partition subsets (similarity search) | Cloud functions on entire dataset (similarity search) |

*Vexless actually performs indexing in a centralized VM, but as we show in §6 this phase can be parallelized.

of the primary advantages of its design is its ability to scale both horizontally and vertically, allowing the addition of more resources per node to handle increasing workloads. This makes it particularly well-suited for handling continuous query workloads with moderate fluctuations, as it provides immediate replies. In a typical deployment, a Milvus/Manu cluster uses event streaming systems (*e.g.*, Kafka, Pulsar) to achieve high-performance data ingestion. The low latency and high performance of service nodes are key advantages of the cluster architecture over serverless counterparts, making it an attractive option for applications that require low-latency guarantees on vector search.

However, a cluster vector DB deployment can become cumbersome to manage in the presence of high workload fluctuations and burstiness. This is because this architecture introduces additional complexity and operational management to adapt the service to the workload needs. Furthermore, workload sparsity can lead to low cost-effectiveness for a cluster vector DB, as resources may be underutilized during periods of low activity. In such scenarios, serverless vector DBs have emerged as an interesting alternative, building on top of FaaS offerings (*e.g.*, AWS Lambda, Azure Functions). By automatically provisioning cloud functions and billing them on a per-execution basis, serverless vector DBs can minimize operational overhead and reduce costs. As a result, cluster and serverless vector DB architectures provide trade-offs that make them ideal for different scenarios, thus requiring careful consideration of the specific use case at hand.

When focusing on serverless vector DBs, there are significant differences between Vexless and this paper. Vexless utilizes a *stateful* type of FaaS offering (*e.g.*, Azure Durable Functions) to maintain low query times by keeping functions in memory. This approach reduces function invocation and data loading times. However, Azure Durable Functions represents a unique FaaS approach that is not widely adopted across vendors. This work, instead, focuses on *stateless* FaaS offerings. Despite potential trade-offs due to their stateless nature, they are the most prevalent FaaS models today (*e.g.*, AWS Lambda, Azure Functions, Google Cloud Functions).

Crucially, the design of Vexless cannot handle continuous data ingestion as it assumes static datasets. Vexless performs a clustering-based data partitioning and indexing approach that must be executed in advance, limiting its ability to handle dynamic datasets. In contrast, we propose a block-based data partitioning approach that enables continuous data ingestion to the system, providing greater flexibility and scalability. In the next section, we analyze in depth the trade-offs of data partitioning between Vexless and this work.

## 4 Trade-offs in Data Partitioning

We have overviewed the architecture of serverless vector DBs. Next, we focus on a critical aspect when distributing a vector DB engine across cloud functions: *data partitioning*. Concretely, we compare the state-of-the-art clustering-based data partitioning [28] with our block-based data partitioning scheme (see Fig. 3) regarding three dimensions relevant in a serverless scenario: i) *partitioning complexity*, ii) *load balancing*, and iii) *query performance*.

### 4.1 Clustering-based Data Partitioning

*Partitioning complexity*: This partitioning method creates clusters of nearby vectors aiming to accelerate queries by filtering out unrelated data partitions. To this end, it uses a three-step indexing pipeline. First, an unsupervised clustering algorithm, such as K-means, is implemented to cluster nearby vectors. It is important to note that clustering algorithms like K-means are *computationally complex* and require the *entire dataset* to operate. In practice, while distributed versions of K-means have been investigated [9], it seems inefficient to implement it on top of cloud functions. For this reason, we found that resorting to a virtual machine for executing the clustering algorithm is a feasible alternative. Once the clustering algorithm completes, the dataset is partitioned and distributed among multiple cloud functions, along with the centroids, to classify the vectors according to their centroids. Finally, multiple parallel cloud functions index the dataset partitions containing nearby vectors.

*Load balancing*: Clustering algorithms like K-means can lead to imbalanced dataset partitions, where some clusters may contain significantly more vectors than others. To address this issue, balanced versions of K-means have been developed, which ensure equal partitioning of datasets across clusters. As pointed out in Vexless [28], producing balanced data partitions is crucial in a serverless setting to avoid straggler cloud functions. Stragglers can delay query responses due to uneven data distribution. However, the balanced version of K-means incurs a higher computational cost

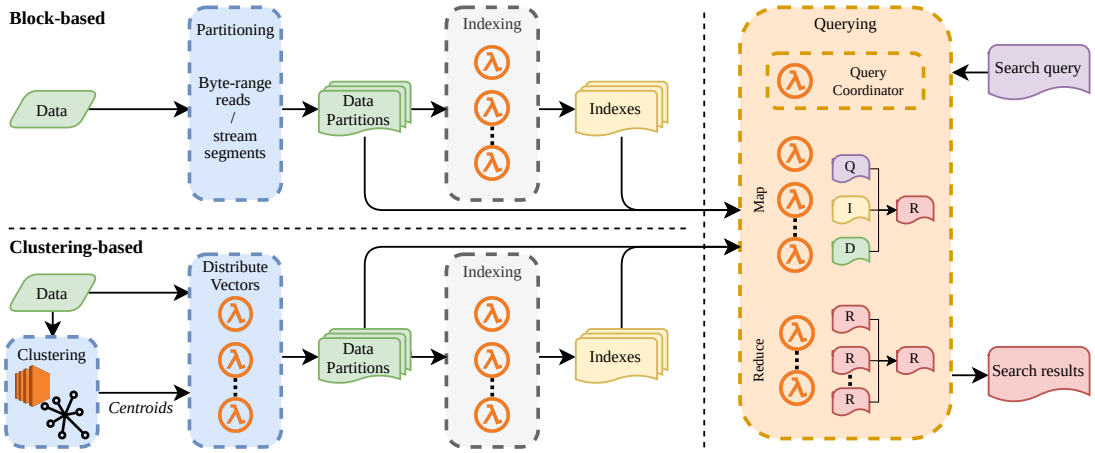


**Figure 3: Life cycle of a serverless vector DB, comparing the block-based and clustering-based approaches. Querying is equal architecture-wise, but while block-based must query on all partitions, the clustering-based allows the query coordinator to filter partitions based on the distance between the query vector and a partition's centroid.**

compared to the standard algorithm, making it impractical for large datasets [6, 18] under a continuous ingestion workload.

*Query performance*: This method groups data partition indexes based on their distances, enabling users to discard partitions with vectors far from the query input. Upon receiving a query, a clustering-based data partitioning system identifies the centroids closest to the query vector. Based on a *data discard* parameter, only the closest centroids are returned. The similarity search is then performed exclusively on data partition indexes associated with these nearby centroids. However, querying fewer centroids reduces *query recall*, as it increases the likelihood of missing the actual closest vectors. Users must understand the trade-off between data filtering and query recall, which can be challenging since many applications require high recall (*e.g.*, > 90%). To address this, Vexless [28] incorporates a data redundancy mechanism that replicates vectors near partition boundaries into adjacent partitions. In §6, we evaluate the behavior of vector redundancy for clustering-based partitioning.

## 4.2 Block-based Data Partitioning

*Partitioning complexity*: In a block-based scheme, the complexity of data partitioning is significantly reduced by not requiring a view of the entire dataset. This approach creates equal-sized data partitions, which simplifies the process and ensures uniform distribution of data. Additionally, the ability to index these chunks in parallel improves scalability, as multiple chunks can be processed simultaneously without dependencies on other parts of the dataset.

*Load balancing*: By creating equal-sized chunks, the scheme ensures that each cloud function handles a uniform amount of data, thus preventing straggler functions. Moreover, the ability to index chunks in parallel allows multiple cloud functions to operate simultaneously, maximizing resource utilization.

*Query performance*: A block-based data partitioning scheme does not consider vector relationships, requiring all partitions to be queried for similarity searches. This results in fast and simple partitioning and indexing phases, avoiding the complexity of analyzing vector relationships. However, the querying phase becomes more computationally expensive, as it must query all partitions without discarding initially dissimilar vectors. Consequently, all vectors are downloaded and queried across indexes, increasing computational requirements and potentially lengthening query times. This trade-off highlights the balance between efficient indexing and comprehensive querying for accurate results.

# 5 Experimental Methodology

In this section, we provide details on our serverless vector DB implementation, as well as the experimentation setup and methodology.

## 5.1 Prototype Implementation

We have developed a serverless vector DB prototype that leverages cloud functions to parallelize the execution of vector DB compute services.[2] Our prototype orchestrates the invocation of cloud functions and transfers data through object storage. The implementation uses Lithops [25], a serverless framework for running cloud functions in parallel in map-reduce fashion.

The main goal of our prototype is to allow us to evaluate different data partitioning schemes in serverless vector DBs. Irrespective of the scheme, the data partitioning process reads a dataset from and stores $N$ partitions to object storage. To generate indexes, the data partitions are distributed across a set of functions using a Lithops `map` operation. Each function creates the corresponding index (or indexes) using Faiss [8] and uploads it to object storage. We use an Inverted File (IVF) configuration of $k = 512$ and multi-probe at 32. After extensive testing, this proved to give the best results overall for the datasets explored in this evaluation.

When querying the data, our prototype acts as a query coordinator and uses two Lithops `maps` to perform a map-reduce operation. Queries are processed in batches, with each Lithops operation (a set of cloud functions) handling a full batch. The `map` phase distributes the data partitions among parallel functions, each of which runs

[2] Source code available at: Anonymized-for-review

**Algorithm 1** Vector distribution for clustering-based partitioning.

**Input:** Database $D$ of $m$ vectors in $\mathbb{R}^n$. Clusters $c_1, c_2, \ldots, c_N$ with centroids $C_1, C_2, \ldots, C_N$, and labels for each vector in $D$. Percentage of redundancy threshold $r$.
**Initialize:** For each cluster $c_i$, initialize indexing partition $I_i$.
**for all** vector $v$ in $D$ **do**
  $c_h \leftarrow$ label from clustering assignment of $v$.
  Add $v$ to $I_h$.
  **for** $i = 1, k$ where $i \neq h$ **do**
    **if** $d(v, C_i) \leq (1 + r)d(v, C_h)$ **then**
      Add $v$ to $I_i$.
    **end if**
  **end for**
**end for**

all queries on the corresponding index (or indexes) and generates partial similarity responses. If a function processes multiple partitions, it combines the partial results before passing them to the `reduce` phase. The `reduce` phase aggregates the map responses and produces the overall top-$k$ similarity results. Finally, our prototype collects the results and computes their accuracy. This process enables efficient and scalable similarity search, leveraging the power of distributed computing and object storage.

*Block-based partitioning.* Data partitioning is applied directly on the indexing functions. Since this approach simply splits the data into chunks, the functions read the dataset from object storage with byte-range requests to get a specific partition. Indexing is then applied to each partition. For querying, the full batch of queries is sent to all parallel functions and applied to all $N$ partitions.

*Clustering-based partitioning.* Clustering-based partitioning requires three additional steps: (1) dataset clustering, (2) vector distribution with redundancy, and (3) partition filtering in querying.

First, clustering is executed on a VM and produces a set of $N$ clusters $c_1, c_2, \ldots, c_N$ with centroids $C_1, C_2, \ldots, C_N$, and a label for each vector in the database that assigns it to a cluster. The baseline clustering algorithm is the Faiss K-means implementation. We also explore the balanced version [15] proposed in Vexless [28]. Note that using a VM for computing the clustering is a favorable configuration, as it is faster and more practical than implementing this phase in a distributed approach on top of cloud functions.

After clustering, we must distribute the vectors to their corresponding partition. While Vexless does this on the same VM where clustering happens, we parallelize this step with a Lithops `map` operation on cloud functions to speed up the process. Each function reads a part of the input dataset from storage using byte ranges (like the blocks implementation) and applies the distribution logic layout in Algorithm 1. This logic includes vector redundancy, which adds boundary vectors to multiple partitions to improve search accuracy. We develop our own redundancy logic for vector redundancy because Vexless does not offer specific guidelines or code for this purpose. Our solution is based on a redundancy percentage[3] $r$ that acts as threshold on how close a second centroid must be compared to the closest for that vector to be also included in the second partition. Initially, vectors are added to the partition that K-means assigns them to.[4] Then the vector is added to other partitions if the distance to their centroid is lower than to the first one extended by $r$. For instance, if the clustering assigns a vector $v$ to a cluster $c_h$ with its centroid $C_h$ at distance $d(v, C_h) = 1$, with $r = 5\%$ we will add $v$ to any partition whose centroid $C_i$ is at $d(v, C_i) \leq 1.05$.

To optimize queries, we employ a data partition filtering technique that selectively discards partitions. We can configure the system to search only on $N_{\text{search}}$ partitions, given as a number or as a percentage of the total number of partitions $N$. During query coordination, we rank partitions based on the proximity of their centroid to the query vector and select the top $N_{\text{search}}$ for searching. This process generates a mapping that determines which partitions to search for each vector in the query batch, which is then sent to the query functions for execution. Although all functions are typically spawned for each batch, the search computation is reduced by $100 - N_{\text{search}}\%$, resulting in improved efficiency.

## 5.2 Setup

*Deployment.* All experiments are run on Amazon Web Services (AWS) using a combination of AWS Lambda, EC2, and S3. In the case of our serverless prototype, the coordination runs on a `c7i.xlarge` EC2 instance. All data is stored in S3, including the datasets and the intermediate results of the processes. AWS Lambda functions are configured with 10 GiB of memory for indexing and 8 GiB for querying, which gives them 6 and 4 vCPUs, respectively, according to service documentation.[5] The indexing process is parallelized on 16 functions for all configurations, splitting partitions evenly among them. Querying uses either 4 or 8 functions in the map phase (also splitting partitions evenly) and a single one for the reduce phase. The block-based approach does not need any additional resources. For the clustering-based approach, K-means clustering runs on a `c7i.12xlarge` instance, which also acts as coordinator in these executions. Vector distribution uses 16 parallel functions (10 GiB). Datasets must be stored in an S3 bucket, while query files must be uploaded to the client EC2 instance.

We deploy Milvus on `c7i.4xlarge` and `c7i.8xlarge` VMs. The goal is to compare Milvus against our prototype with the same amount of resources.[6] This setup matches the CPU and memory resources of our serverless vector DB prototype, ensuring a fair and comparable evaluation. Milvus is set to use the same IVF configuration for indexing as our prototype. These experiments also use a `c7i.xlarge` instance as a client running the `vectordbbench` tool [36], developed by the same team as Milvus (Zilliztech). Specifically, we use a custom docker container image that includes the `vectordbbench` tool, all its requirements, and both the datasets and vectors in the specific format.

*Methodology.* Datasets containing collections of vectors are initially stored in object storage as a single object each. All processes of a vector DB are evaluated on different number of partitions $N = \{16, 32, 64, 128\}$, and use equivalent resources in all systems,

[3] As opposed to Vexless, which uses an arbitrary distance value.

[4] This is the closest centroid for the baseline K-means, but it could be another one in the balanced version of the algorithm.

[5] Functions have one vCPU per 1769 MiB and scale both resources proportionally [3].

[6] We performed experiments with larger VM flavors for Milvus (*e.g.*, `c7i.24xlarge` to match serverless indexing resources) and we observed that it does not seem to use all the resources during indexing phase.

either with cloud functions or VMs. Data ingestion, partitioning, and indexing are reported aggregated as "data partitioning time" because it is the most significant part and the focus of our evaluation. Ingestion is always from object storage and equivalent for all systems. Indexing is an independent, stateless process that only depends on the size of the partitions, not the partitioning scheme, so it does not show significant variation between the evaluated approaches. Reported results are averaged across $\geq 5$ executions, and whiskers in plots denote standard deviation.

We evaluate search accuracy using the recall metric, which measures the proximity of the response to the true neighbors of the query vector, with 100% recall indicating a perfect score. To calculate recall, we first identify the true neighbors of all query vectors using a Flat Index on the entire dataset and store them in object storage. During evaluation, each query execution is compared to these pre-computed true results to assess accuracy. For querying evaluation, we utilize a batch of 1000 queries extracted from the original query file accompanying each dataset, with each query requesting the top-10 nearest vectors.

We calculate cost based on the processing time of the experiments and the resources being utilized at each moment. We use current AWS pricing as of March 2025. For the serverless implementation, cost is the sum of the running time of all cloud functions plus invocation fees. We account the cost of VMs as the fraction corresponding to the seconds they are actively used in the experiments. Later, we also show the cost of having a cluster running for longer periods of time, even if idle, as in a real scenario.

*Datasets.* We use three publicly available datasets that are commonly employed to evaluate the quality of approximate nearest neighbors search algorithms in vector DBs: (i) DEEP [35], which we use in subsets of 100k, 1M, 10M, and 100M embeddings of the Deep1B with 96 dimensions normalized with L2 distance extracted from the last fully-connected layer of a GoogLeNet [29] model trained with the ImageNet [7] dataset; (ii) SIFT [17], which consists of 10M embeddings with 128 dimensions, the Scale Invariant Feature Transform (SIFT) transforms the image data into a large number of features (scale invariant coordinates) that densely cover the image features; and (iii) GIST [20], which consists of 1M embeddings with 960 dimensions, a gist represents an image scene as a low-dimensional vector.

## 6 Clustering vs Block-based Data Partitioning

In the first experimental section, we aim to answer the following questions regarding data partitioning in serverless vector DBs:

(1) What is the cost of achieving balanced data partitions in the clustering-based approach? (§6.1)
(2) How do query recall and storage overhead trade off with vector redundancy in clustering-based partitioning? (§6.2)
(3) How do clustering-based and block-based partitioning compare in terms of data partitioning and indexing performance? (§6.3)
(4) What are the query performance differences between clustering-based and block-based data partitioning? (§6.4)
(5) Does clustering-based partitioning amortize partitioning costs via query filtering, compared to a block-based scheme? (§6.5)

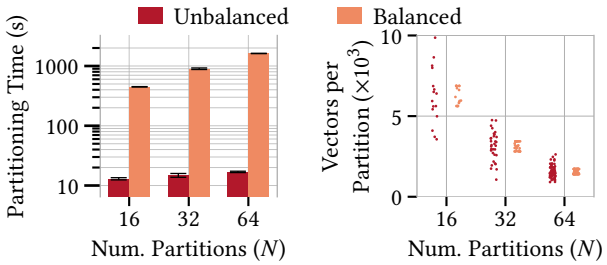


**Figure 4: Partitioning time and vector dispersion across partitions for the two K-means versions (DEEP100k dataset).**

**Table 2: Query performance comparison or the two K-means on DEEP100k dataset (single batch of 1000 queries).**

| Params | | Balanced | | | Unbalanced | | |
|---|---|---|---|---|---|---|---|
| $N$ | $N_{search}$ | Time (s) | $\pm\sigma$ | Recall | Time (s) | $\pm\sigma$ | Recall |
| 16 | 1 | 7.103 | 0.924 | 71.30 | 7.491 | 1.791 | 70.68 |
| 16 | 4 | 6.716 | 0.031 | 92.55 | 6.742 | 0.026 | 92.97 |
| 16 | 8 | 6.712 | 0.026 | 95.01 | 6.789 | 0.048 | 95.37 |
| 16 | 12 | 6.771 | 0.026 | 95.49 | 6.779 | 0.066 | 95.64 |
| 16 | 16 | 6.761 | 0.048 | 95.52 | 6.825 | 0.077 | 95.65 |
| 32 | 1 | 7.095 | 0.874 | 63.54 | 7.118 | 0.923 | 65.59 |
| 32 | 8 | 6.766 | 0.049 | 94.35 | 6.750 | 0.031 | 94.75 |
| 32 | 16 | 6.772 | 0.036 | 95.64 | 6.823 | 0.036 | 95.90 |
| 32 | 24 | 6.856 | 0.027 | 95.81 | 6.860 | 0.062 | 96.05 |
| 32 | 32 | 6.934 | 0.048 | 95.82 | 7.128 | 0.343 | 96.05 |
| 64 | 1 | 7.127 | 0.904 | 60.54 | 6.708 | 0.029 | 60.43 |
| 64 | 16 | 6.815 | 0.065 | 96.44 | 7.247 | 0.888 | 96.34 |
| 64 | 32 | 7.699 | 0.993 | 97.03 | 8.315 | 0.976 | 96.97 |
| 64 | 48 | 8.811 | 0.480 | 97.12 | 9.371 | 1.706 | 97.03 |
| 64 | 64 | 9.061 | 0.425 | 97.13 | 10.287 | 2.169 | 97.06 |

### 6.1 The Cost of Balanced Data Partitions

As a representative of clustering-based data partitioning for serverless vector DBs, Vexless emphasizes the need for balanced data partitions to prevent straggler cloud functions. To this end, it proposes a balanced K-means algorithm that maintains near-equally-sized vector partitions. We analyze the cost of keeping balanced data partitions by deploying two clustering algorithms in our prototype: unbalanced K-means (Faiss default implementation) and balanced K-means [15]. The latter formulates the cluster assignment step as a Minimum Cost Flow linear network optimization problem.[7]

First, we focus on the data partitioning and indexing cost for both K-means flavors. Fig. 4 (right) shows the number of vectors per partition in both cases. Naturally, the balanced K-means exhibits a significantly lower dispersion of vectors per partition (*e.g.*, $\sigma = 0.541 \times 10^3$ for 16 partitions) compared to unbalanced K-means (*e.g.*, $\sigma = 1.778 \times 10^3$ for 16 partitions). However, Fig. 4 (left) also shows the differences in computational complexity related to partitioning and indexing the DEEP100k dataset for unbalanced ($O(nc)$) and balanced K-means ($O((n^3c + n^2c^2 + nc^3)\log(n + c))$) algorithms.

[7] https://pypi.org/project/k-means-constrained/

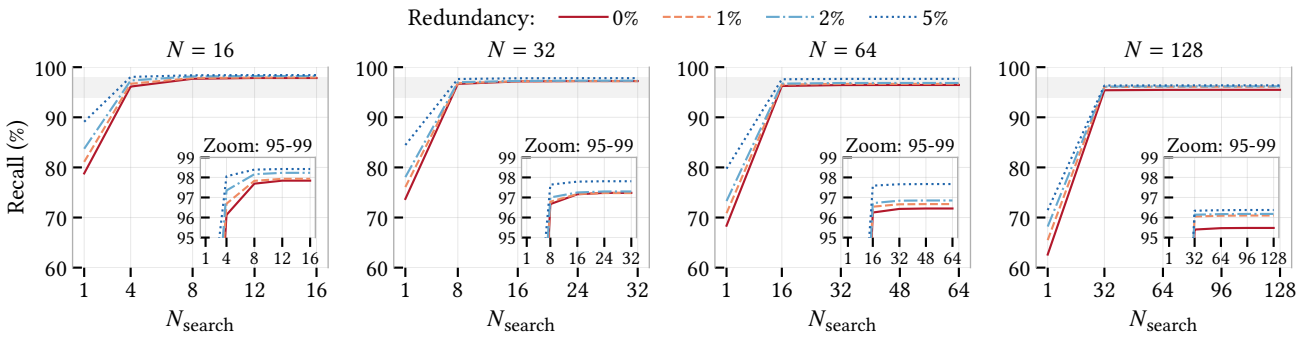


**Figure 5: Effect of redundancy and $N_{\text{search}}$ on recall depending on the total number of partitions ($N$).**

**Table 3: Impact of vector redundancy on the vectors per data partition standard deviation ($\sigma$) for DEEP100k dataset.**

| Vector redundancy ($r$) | Standard deviation ($\sigma$) | | |
|---|---|---|---|
| | $N = 16$ | $N = 32$ | $N = 64$ |
| $r = 0\%$ (Unbalanced K-means) | 1778 | 903 | 378 |
| $r = 0\%$ (Balanced K-means) | 541 | 264 | 138 |
| $r = 5\%$ (Balanced K-means) | 1491 | 1792 | 660 |

Visibly, using the balanced version of K-means incurs from 35× up to 96× higher partitioning time, mostly due to the computation of centroids. We could not experiment with larger datasets using the balanced K-means version due to the growing clustering times.

One could think that higher partitioning and indexing cost may be linked to significant advantages in query performance. In Table 2, we observe that the balanced K-means algorithm outperforms the unbalanced version in terms of query time, with an average reduction of 2.6% across all experiments, and a maximum reduction of 13.5% for $N = 64$ and $N_{search} = 64$. In terms of recall, the balanced K-means algorithm achieves an average increase of 0.3% across all experiments, with a maximum increase of 3.2% for $N = 32$ and $N_{search} = 1$. The standard deviation of query times is also reduced by an average of 34.6% for the balanced K-means algorithm, indicating more consistent performance. However, such advantages seem modest compared to the cost related to data partitioning and indexing. This is especially true when considering large and/or dynamic datasets that require continuous indexing activity.

*Conclusion.* Using a balanced version of K-means is not practical for delivering efficient data partitioning in serverless vector DBs, especially considering large and/or dynamic data.

## 6.2 Effects of Vector Redundancy

A contribution of Vexless is to propose a vector redundancy mechanism to mitigate loss in query recall when filtering data partitions according to the input vector distance to centroids. Next, we provide an evaluation of the effects of incorporating vector redundancy. The vector redundancy implementation details can be found in §5.1.

Retaking the discussion on partition load balancing, Table 3 shows the impact of vector redundancy ($r$) on the dispersion of vectors per data partition ($\sigma$). Visibly, adding vector redundancy to the balanced K-means version re-introduces load imbalance across partitions. For example, in the DEEP dataset, setting $r = 5\%$ makes the dispersion of vectors per data partition ($\sigma$) to be up to 1.98× worse than the unbalanced K-means version. This insight is important, as Vexless proposes using two techniques that seem to have conflicting outcomes. Note that given the previous results, the remainder of our analysis uses the unbalanced version of K-means.

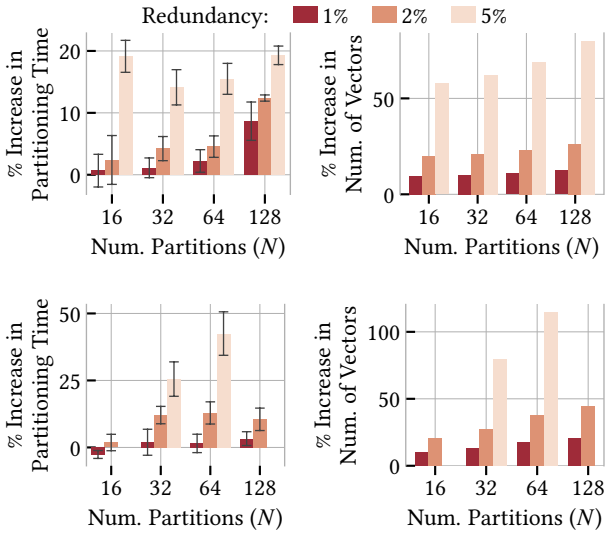


**Figure 6: Percentage of increase in partitioning time and stored vectors for different $r$ values with respect to no redundancy. DEEP10M dataset on top, GIST1M dataset on bottom.**

Fig. 5 shows the query recall of query search based on the number of partitions available ($N$) and searched ($N_{search}$). Visibly, vector redundancy exhibits a 3% to 16% improvement in query recall compared to the baseline (*i.e.*, no redundancy) when searching in the closest data partition ($N_{search} = 1$). However, for $N_{search} = 1$, we also observe that query recall is relatively low (< 90%), which may not be precise enough for many applications. At the same time, the recall improvements of vector redundancy become less evident (< 2%) as $N_{search}$ increases.

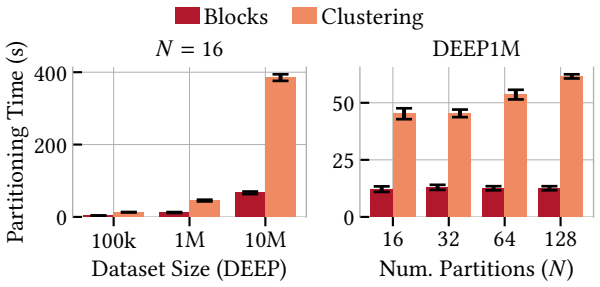


**Figure 7: Partitioning time comparison for different data volume and number of partitions.**

Interestingly, vector redundancy has additional costs in terms of data partitioning and indexing time, as well as storage overhead. Fig. 6 (left) shows the relative increment in data partitioning and indexing time depending on the vector redundancy level for DEEP10M and GIST1M datasets. In case of DEEP, redundancy seems affordable up to $r = 2\%$ if $N \leq 64$ ($< 5\%$). However, higher $r$ or $N$ configurations increase processing time between 9 to 19% on average. Storage overhead also grows from 9 up to 80% depending on $r$ and $N$. GIST, which has a higher dimensionality, suffers even more from this cost growth, reaching a 40% overhead in processing time and more than doubling stored vectors for $r = 5\%$ and $N = 64$.

*Conclusion.* The benefits of vector redundancy ($r \geq 5\%$) are limited to searching very few partitions (*e.g.*, $N_{search} = 1$), resulting in low query recall levels ($< 90\%$) that may not meet many applications' needs. Considering the load balancing, indexing, and storage costs, the general applicability of vector redundancy is unclear.

## 6.3 Clustering vs Blocks: Data Partitioning

Next, we compare the proposed block-based data partitioning with the clustering-based counterpart (imbalanced, no redundancy) in terms of data partitioning and indexing time (see Fig. 7). As described in §5.2, we use 16 cloud functions for indexing data partitions in both cases, whereas the clustering-based data partitioning uses an additional `c7i.12xlarge` VM for computing the centroids.

As expected, for a fixed dataset size (DEEP1M), the block-based partitioning is not sensitive to the number of partitions (see Fig. 7, right). The main reason is that block-based partitioning can be executed in an embarrassingly parallel fashion and can be performed with parallel byte-range reads from storage. On the other hand, the stateful nature of clustering-based data partitioning requires a complete view of the dataset and its complexity increases with the number of data partitions (*i.e.*, centroids).

But the real limitation of clustering-based data partitioning is rendered when scaling the dataset size for the same amount of resources (Fig. 7, left). Visibly, increasing the dataset size from 100k to 1M and from 1M to 10M leads to data partitioning times 3.6× and 8.6× higher, respectively. The main reason is that the K-means clustering stage needs to load the full dataset in a single VM —not in parallel— and perform an increasingly expensive computation with dataset size. Conversely, for block-based data partitioning, partitioning time scales linearly with data volume for the same amount of resources. This is just because each cloud function has more data to index in each block.

*Conclusion.* Block-based data partitioning is more efficient and scalable than clustering-based partitioning, especially as dataset size and number of partitions increase.

**Table 4: Recall comparison of block-based (Bl.) versus clustering-based (Cl.) data partitioning (DEEP10M dataset).**

| | Num. Partitions ($N$) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 16 | | 32 | | 64 | | 128 | |
| $N_{search}$ | Bl. | Cl. | Bl. | Cl. | Bl. | Cl. | Bl. | Cl. |
| **25%** | 99.24 | 96.13 | 99.26 | 96.67 | 99.44 | 96.25 | 99.30 | 95.40 |
| **50%** | | 97.69 | | 97.16 | | 96.43 | | 95.47 |
| **75%** | | 97.84 | | 97.23 | | 96.45 | | 95.48 |
| **100%** | | 97.84 | | 97.23 | | 96.45 | | 95.48 |

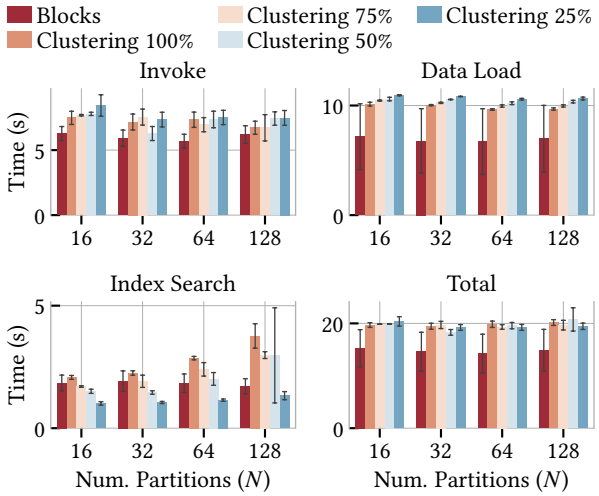


**Figure 8: Query time comparison of block-based and clustering-based data partitioning for different number of partitions ($N$) and $N_{search}$ at 100, 75, 50, and 25%.**

## 6.4 Clustering vs Blocks: Query Performance

In this section, we compare the implications of data partitioning on query performance: query recall (Table 4) and query times (Fig. 8).

Table 4 shows a query recall comparison between block-based and clustering-based data partitioning. We observe that the query recall of the block-based scheme consistently outperforms clustering-based (unbalanced) partitioning by 1.4% to 3.9% for different values of $N$ and $N_{search}$. However, these query recall differences are not very significant, even for low values of $N_{search}$. This indicates that data filtering in queries can be effectively leveraged while maintaining acceptable query recall. Interestingly, when $N = N_{search}$, the query recall of clustering-based data partitioning is slightly lower than the block-based scheme. This may be due to the interplay of vector distribution with IVF indexes in our prototype. To inspect

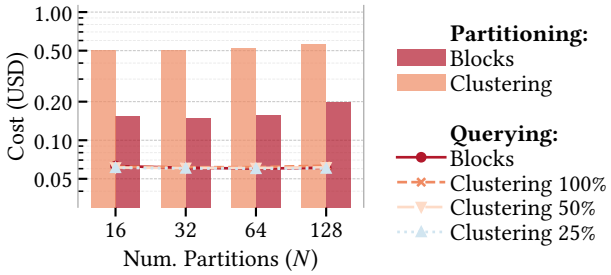


**Figure 9: Total indexing and querying costs of clustering-based and block-based data partitioning with 10 batches of 1000 queries (DEEP10M dataset).**

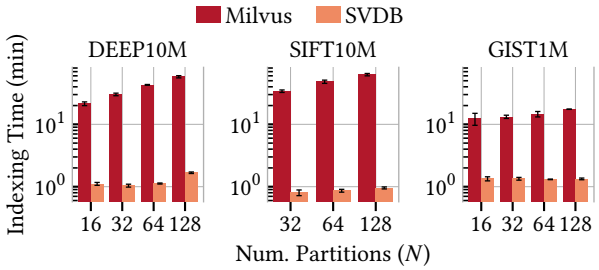


**Figure 10: Partitioning and indexing time for different datasets on Milvus and SVDB.**

this, we reproduced the experiment with HNSW indexes (as in Vexless) obtaining similar results (*i.e.*, block-based partitioning shows better recall than the clustering-based scheme by 0.09% to 3.25%).

Fig. 8 provides a breakdown of query latency in our prototype for both cluster-based and block-based data partitioning schemes. Interestingly, data partitioning has an impact of the query phases of a serverless vector DB. For example, the `invoke` phase is in charge of instantiating the cloud functions, whereas the `query` phase executes cloud functions on data partitions. In both phases, the clustering-based data partitioning induces overheads related to the system inferring the right partitions to query based on the input vector. This has a toll on the latency of these phases in most cases. As block-based data partitioning is stateless, the latency of these phases remains constant. Similarly, the `data load` phase exhibits higher latency for the clustering-based data partitioning. This is due to straggler functions impacted by data partition imbalance. This is not a problem for block-based data partitioning either. As a result, our prototype exhibits query times 5.6% to 10.7% faster using block-based partitioning compared to clustering-based partitioning.

*Conclusion.* The execution complexity and data partition imbalance in clustering-based partitioning prevents it to reduce query times in a serverless vector DB, even with query filtering.

## 6.5 Clustering vs Blocks: Cost Analysis

Next, we focus on understanding the partitioning and querying costs related to applying clustering-based and block-based data partitioning in a serverless vector DB. It is important to note that, due to the nature of our prototype, queries are executed in batches. Thus, query costs are calculated per batch (not individual queries).

In terms of data partitioning and indexing, Fig. 9 shows that the longer processing times of K-means clustering, plus the additional VM needed, incurs a 2.82× to 3.44× increase in partitioning costs. Note that we are evaluating the partitioning and indexing of a static dataset. If we consider dynamic data, clustering-based partitioning costs could significantly escalate due to re-processing exiting data.

Notably, our prototype reveals that the costs of queries using clustering-based data partitioning do not offset the costs of data partitioning, even when filtering data on a per-query basis. This is because batch query execution (1000 queries at a time) often requires access to the entire dataset to produce results for the batch. As a result, while data filtering may reduce cloud function execution costs for individual queries, its effectiveness is diminished when submitting multiple queries in a batch to minimize query latency. This insight highlights the trade-off between query costs, data filtering, and query latency in a serverless vector DB.

*Conclusion.* Clustering-based partitioning incurs higher indexing costs than the block-based scheme. Also, clustering-based partitioning costs may not be offset by query savings, even discarding data, if we consider query batches to amortize latency.

# 7 Milvus vs Block-based Serverless Vector DB

In this section, we compare our proposed serverless vector DB prototype ("SVDB" for short) with block-based data partitioning against a popular cluster vector DB system: Milvus. In particular, the following experiments aim to answer the following questions:

(1) How does SVDB's vector indexing performance compare to Milvus? (§7.1)
(2) How does querying with SVDB compare to Milvus? (§7.2)
(3) How does SVDB improve costs compared to Milvus? (§7.3)
(4) How does SVDB scale with data volume? (§7.4)

## 7.1 Partitioning and Indexing Performance

Next, we compare the data partitioning/indexing performance in Milvus and SVDB. To this end, Fig. 10 shows the processing time of both systems for multiple datasets and partition numbers ($N$).

Visibly, SVDB indexes all three datasets $x$× to $y$× faster than Milvus. This is because SVDB fully exploits the parallelism of cloud functions. In contrast, Milvus does not parallelize the ingestion of a static dataset, resulting in under-utilization of available resources. Note that we experimented with deploying Milvus on different VM sizes (*e.g.*, `8xlarge`, `4xlarge`), but observed similar partitioning performance results. This may indicate some internal limitation in the indexing implementation of the system. Moreover, SVDB partitioning time is not sensitive to the number of partitions. Instead, Milvus processing time increases with the number of partitions.

*Conclusion.* SVDB achieves faster vector partitioning and indexing compared to Milvus for an equivalent amount of resources.

## 7.2 Query Performance

In this section, we evaluate the query performance of Milvus and SVDB in terms of query times (Fig. 11) and recall (Table 5). As in the previous section, we evaluate query performance executing query batches (1000 vector queries/batch). For context, query times

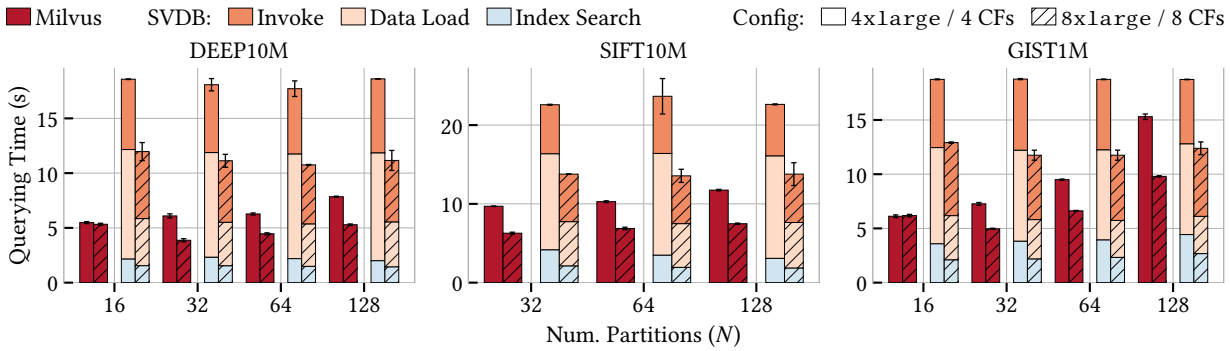


**Figure 11: Query times for 3 datasets on Milvus and SVDB. The plot splits the time of SVDB into cloud function invocation overhead (Invoke), index downloading (Data Load), and query execution (Index Search). Milvus has the query node running with the data loaded.**

**Table 5: Recall of top-10 similarity searches for a batch of 1000 queries in Milvus and SVDB.**

| Num. Partitions | | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|
| **DEEP** | Milvus | 99.27 | 99.43 | 99.38 | 99.28 |
| | SVDB | 99.24 | 99.26 | 99.44 | 99.30 |
| **SIFT** | Milvus | - | 99.80 | 99.70 | 99.60 |
| | SVDB | - | 99.37 | 99.41 | 99.50 |
| **GIST** | Milvus | 98.40 | 98.70 | 99.20 | 99.70 |
| | SVDB | 98.56 | 98.64 | 98.92 | 99.33 |

reported for Milvus imply that the cluster is already set up and the data loaded into memory. However, reaching this state takes Milvus over 45 s which are not accounted in our results.

Fig. 11 compares query times in Milvus and SVDB for two equivalent resource configurations (*i.e.,* `4xlarge`/`8xlarge` VMs and 4/16 cloud functions). As expected, Milvus offers query times $x\times$ to $y\times$ faster than SVDB. That is, Milvus is continuously running as a cluster service with data and indexes loaded in memory, whereas SVDB requires to start the functions and load data on each query batch. We also observe that, in both systems, adding more resources reduces query times. This implies that they are able to parallelize the execution of query and fully utilize the underlying resources.

Inspecting the latency breakdown, the `index search` phase of the query in SVDB is faster than Milvus. However, querying also requires coordinating cloud functions, which is part of the `invoke` phase overhead. While part of this overhead is also present in Milvus for coordinating the queries, SVDB introduces additional function invocation latency. The `data load` phase is exclusive to SVDB. This overhead can be improved with stateful FaaS services, as proposed in Vexless [28]. Finally, Table 5 shows that query recalls are equivalent in both systems[8], ensuring accurate vector searches.

*Conclusion.* Query times in SVDB are slower than Milvus due to the expected overheads in function invocation and data loading.

[8]Note that SIFT has higher dimensionality compared to DEEP and cannot be processed with 16 partitions, so the experiments start at 32 for this dataset.

**Table 6: Cost of indexing and querying on Milvus and SVDB for the DEEP10M dataset (SIFT10M and GIST1M show similar results). Q-Dense refers to 10 batches of 1000 queries run back to back. Q-Sparse-1 refers to the same 10 batches run in the span of 1 hour and Q-Sparse-24 in the span of 24 hours.**

| | Indexing | Q-Dense | Q-Sparse-1 | Q-Sparse-24 |
|---|---|---|---|---|
| **Milvus** | $10.71 | $0.07 | $1.43 | $34.39 |
| **SVDB** | $0.66 | $0.48 | $0.48 | $0.48 |

## 7.3 Cost analysis

Next, we focus on the economic costs of Milvus and SVDB. Table 6 shows that the indexing of the DEEP10M DATASET is 16.2× cheaper in SVDB (0.66$) compared to Milvus (10.71$). A reason is that Milvus does not parallelize data indexing irrespective of the VM flavor. This induces longer processing times that translate into monetary costs.

Querying is more cost-effective on Milvus only if queries are executed as a dense workload ($0.07 versus $0.48 for running 10 batches of 1000 queries back to back). However, a cluster vector DB must always be running, leading to higher costs than a serverless solution for sparse workloads. For instance, if the 10 batches of queries were spread over an hour (`Q-Sparse-1` in Table 6), Milvus cost $1.43, while SVDB would remain at $0.48. This cost difference increases with even sparser workloads. Also, the startup time for a Milvus cluster is much longer than for cloud functions (≈ 45 seconds versus ≈ 2 seconds), making it impractical to start dynamically like SVDB, as it would severely impact query latency.

*Conclusion.* Compared to Milvus, SVDB enables faster indexing and on-demand query function allocation. This results in better cost-effectiveness, especially for sparse workloads.

## 7.4 Scalability

Finally, we aim to evaluate the scalability of SVDB, both in terms of data partitioning and as a system. To this end, we run both indexing and querying for two subsets of the DEEP1B dataset, containing 10M and 100M vectors. Accordingly, we also increase the amount of resources by 10× to process the latter. Specifically, we create 32 and 320 data partitions, indexed on 16 and 160 cloud functions, and

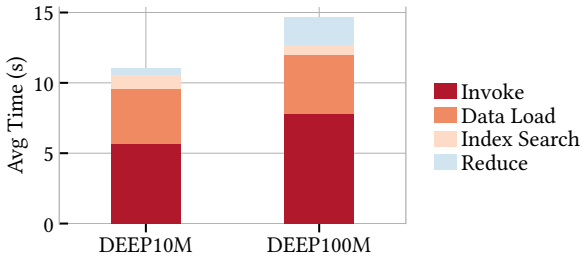


**Figure 12: Querying time breakdown for DEEP10M (32 partitions in 8 CFs) and DEEP100M (320 partitions in 80 CFs).**

use 8 and 80 functions for querying. Therefore, each partition is the same size in both configurations, and each function processes the same amount of data.

As expected, partitioning time is equal in both cases (≈ 55 seconds) because it is an embarrassingly parallel process. Specifically, partitioning and indexing the DEEP100M dataset is about a second slower due to the overhead of invoking more functions. In this sense, Fig. 12 presents a breakdown for the querying phase. The plot shows that loading the indexes and querying takes the same time in both cases, demonstrating the scalability of the task. As expected, the overhead of invoking and managing more functions is higher (40%) and reduce time for collecting the results also increases (4×). The latter aspect can be explained because Lithops by default uses as ingle reduce function [25]. This implementation limitation can be solved by using multiple reducers.

*Conclusion.* Our serverless vector DB with block-based data partitioning can scale the resources to arbitrarily large dataset sizes.

## 8 Related Work

Vector DBs, such as Pinecone [24], Weaviate [32], and Milvus [12, 31], have gained significant attention in recent years due to the increasing demand for efficient and scalable similarity search in various applications [13], including natural language processing [23], image recognition [14], and recommendation systems [34]. Nevertheless, most distributed vector DBs rely on traditional cluster architectures, which can be limiting when handling sparse and bursty workloads. The emergence of serverless vector DBs is a relatively recent development, and as such, none of the recent surveys in the field have provided specific coverage of this new family of systems [21, 22, 26]. This paper aims to provide a timely and specific overview of the architecture of serverless vector DBs, including a systematic comparison with cluster counterparts.

The convergence of the serverless paradigm with vector DBs is the primary focus of this paper. Recently, the industry has seen the emergence of vector DB services marketed as "serverless", such as Weaviate Serverless Cloud [32], Upstash [30], and Amazon OpenSearch Service as a Vector DB [1]. However, these services primarily aim to automate the provisioning of vector DB deployments. While simplifying operational complexity, this model only partially realizes the potential of a serverless architecture. A truly serverless vector DB system would involve distributing the vector DB engine across cloud functions, a concept that has only been explored in Vexless [28]: the first serverless vector DB of this kind to date.

To our knowledge, this paper is the first experimental analysis that evaluates in depth the design space of data management in serverless vector DBs, with especial emphasis on data partitioning. We believe that the observations from our analysis can help drive new generations of serverless vector DBs to achieve better performance, efficiency, and cost-effectiveness.

## 9 Discussion and Conclusions

*Discussion.* Our experiments validate the hypothesis that clustering-based data partitioning is generally impractical in serverless vector DBs. First, data partitioning and indexing is slower compared to our block-based scheme because it requires a clustering stage, which is stateful and cannot be efficiently parallelized. Furthermore, when revisiting the state-of-the-art method for performing clustering-based data partitioning in a serverless vector DB (Vexless [28]), we found interesting insights. For example, using a balanced K-means clustering for achieving balanced data partitions incurs partitioning times that are 35× to 96× higher than unbalanced K-means. We also found that vector redundancy —a technique to improve query recall when filtering data partitions— has limited benefits (and multiple costs) beyond querying very few data partitions, which generally leads to low recall. When considering executing query batches in a stateless FaaS system, clustering-based data partitioning introduces additional execution complexity that does not benefit from data filtering, either in terms of query times or cost amortization. Based on these observations, we conclude that block-based data partitioning is a more practical and effective way of managing large and dynamic datasets in serverless vector DBs.

As a natural next step of our analysis, we compared our serverless vector DB prototype with a popular cluster vector DB (Milvus). Our results show that our prototype achieves 3.6× to 8.6× faster data partitioning and indexing times than Milvus due to its ability to fully exploit the parallelism of cloud functions. While query times in our serverless vector DB are slower than Milvus due to the expected overheads in function invocation and data loading, this has a direct translation into economic costs. To wit, our serverless vector DB also offers better cost-effectiveness, especially for sparse workloads, as it does not require a service continuously running like Milvus. Overall, we believe that a serverless vector DB with block-based data partitioning offers an interesting alternative to cluster vector DBs, especially when considering sparse/bursty workloads and reduced infrastructure/operational costs.

*Conclusion.* Serverless vector DBs are a promising architecture for managing sparse and bursty vector workloads while reducing operational costs. However, they are still in their infancy. In this paper, we provide a timely overview of this new family of systems. Additionally, we analyze a key aspect of their operation: data partitioning. Through extensive experiments, we demonstrate that the current state-of-the-art approach for data partitioning (clustering-based) has significant limitations. To address these, we propose a simple yet practical block-based data partitioning scheme. Our findings show that a serverless vector DB utilizing block-based data partitioning is competitive compared to a cluster vector DB (Milvus) in various aspects (*e.g.*, indexing time, query recall). We hope that the insights from this work will help driving better performance and efficiency for the next generation of serverless vector DBs.

# References


[1] AWS. 2025. *Amazon OpenSearch Service as a Vector Database.* Amazon. Retrieved Mar. 12, 2025 from https://aws.amazon.com/es/opensearch-service/serverless-vector-database/

[2] AWS. 2025. *AWS Lambda.* Amazon. Retrieved Mar. 12, 2025 from https://aws.amazon.com/en/lambda/

[3] AWS. 2025. *Configure Lambda function memory.* Amazon. Retrieved Mar. 12, 2025 from https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html

[4] Azure. 2025. *Azure Functions overview.* Microsoft. Retrieved Mar. 12, 2025 from https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview

[5] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. 2023. FMI: Fast and Cheap Message Passing for Serverless Functions. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 373–385. https://doi.org/10.1145/3577193.3593718

[6] Rieke de Maeyer, Sami Sieranoja, and Pasi Fränti. 2023. Balanced k-means revisited. *Applied Computing and Intelligence* 3, 2 (2023), 145–179. https://doi.org/10.3934/aci.2023008

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database . In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPR Workshops)*. IEEE Computer Society, Los Alamitos, CA, USA, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[8] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).

[9] Rui Máximo Esteves, Thomas Hacker, and Chunming Rong. 2013. Competitive K-Means, a New Accurate and Distributed K-Means Algorithm for Large Datasets. In *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01 (CLOUDCOM '13)*. IEEE Computer Society, USA, 17–24. https://doi.org/10.1109/CloudCom.2013.89

[10] Google. 2025. *Cloud Run functions.* Google. Retrieved Mar. 12, 2025 from https://cloud.google.com/functions

[11] Martin Grohe. 2020. word2vec, node2vec, graph2vec, X2vec: Towards a Theory of Vector Embeddings of Structured Data. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Portland, OR, USA) *(PODS'20)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3375395.3387641

[12] 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 (Aug. 2022), 3548–3561. https://doi.org/10.14778/3554821.3554843

[13] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[14] Benjamin Klein, Guy Lev, Gil Sadeh, and Lior Wolf. 2015. Associating neural word embeddings with deep image representations using Fisher Vectors . In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 4437–4446. https://doi.org/10.1109/CVPR.2015.7299073

[15] Josh Levy-Kramer. 2018. *k-means-constrained.* https://github.com/joshlk/k-means-constrained

[16] 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* 32, 5 (2023), 1–29.

[17] David G Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision* 60, 2 (Nov. 2004), 91–110.

[18] Mikko I Malinen and Pasi Fränti. 2014. Balanced k-means for clustering. In *Structural, Syntactic, and Statistical Pattern Recognition: Joint IAPR International Workshop, S+ SSPR 2014, Joensuu, Finland, August 20-22, 2014. Proceedings*. Springer, Springer, Berlin, Heidelberg, Germany, 32–41.

[19] Johannes Manner, Martin EndreB, Tobias Heckel, and Guido Wirtz. 2018. Cold Start Influencing Factors in Function as a Service . In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE Computer Society, Los Alamitos, CA, USA, 181–188. https://doi.org/10.1109/UCC-Companion.2018.00054

[20] Aude Oliva and Antonio Torralba. 2001. Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope. *International Journal of Computer Vision* 42, 3 (May 2001), 145–175.

[21] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *The VLDB Journal* 33, 5 (2024), 1591–1615.

[22] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector database management techniques and systems. In *Companion of the 2024 International Conference on Management of Data*. 597–604.

[23] Mohammad Taher Pilehvar and Jose Camacho-Collados. 2020. *Embeddings in natural language processing: Theory and advances in vector representations of meaning*. Morgan & Claypool Publishers, Switzerland.

[24] Pinecone. 2025. *Pinecone.io.* Pinecone Systems. Retrieved Mar. 12, 2025 from https://www.pinecone.io

[25] Josep Sampé, Marc Sánchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro García-López. 2023. Outsourcing Data Processing Jobs With Lithops. *IEEE Transactions on Cloud Computing* 11, 1 (2023), 1026–1037. https://doi.org/10.1109/TCC.2021.3129000

[26] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Comput. Surv.* 54, 11s, Article 239 (Nov. 2022), 32 pages. https://doi.org/10.1145/3510611

[27] Richard Shan. 2024. A Deep Dive into Vector Stores: Classifying the Backbone of Retrieval-Augmented Generation . In *2024 IEEE International Conference on Big Data (BigData)*. IEEE Computer Society, Los Alamitos, CA, USA, 8831–8833. https://doi.org/10.1109/BigData62323.2024.10825992

[28] 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, Article 187 (May 2024), 26 pages. https://doi.org/10.1145/3654990

[29] 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 *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[30] Upstash. 2025. *Upstash - Serverless Data Platform.* Upstash. Retrieved Mar. 12, 2025 from https://upstash.com/

[31] 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 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2614–2627. https://doi.org/10.1145/3448016.3457550

[32] Weaviate. 2025. *Weaviate.io.* Weaviate. Retrieved Mar. 12, 2025 from https://weaviate.io/

[33] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 194–205.

[34] 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 (Aug. 2020), 3152–3165. https://doi.org/10.14778/3415478.3415541

[35] Artem Babenko Yandex and Victor Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors . In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 2055–2063. https://doi.org/10.1109/CVPR.2016.226

[36] Zilliz. 2025. *VectorDBBench: A Benchmark Tool for VectorDB.* https://github.com/zilliztech/VectorDBBench