Vector Database solution with NetApp

NetApp Solutions

NetApp
April 26, 2024

Table of Contents

Vector Database Solution with NetApp

Karthikeyan Nagalingam and Rodrigo Nascimento, NetApp

This document provides a thorough exploration of the deployment and management of vector databases, such as Milvus, and pgvecto an open-source PostgreSQL extension, using NetApp's storage solutions. It details the infrastructure guidelines for using NetApp ONTAP and StorageGRID object storage and validates the application of Milvus database in AWS FSX for NetApp ONTAP. The document elucidates NetApp's file-object duality and its utility for vector databases and applications that support vector embeddings. It emphasizes the capabilities of SnapCenter, NetApp's enterprise management product, in offering backup and restore functionalities for vector databases, ensuring data integrity and availability. The document further delves into NetApp's hybrid cloud solution, discussing its role in data replication and protection across on-premises and cloud environments. It includes insights into the performance validation of vector databases on NetApp ONTAP, and concludes with two practical use cases on generative AI : RAG with LLM and the NetApp's internal ChatAI. This document serves as a comprehensive guide for leveraging NetApp's storage solutions for managing vector databases.

The Reference Architecture focus on the following:

Introduction

Introduction

Vector databases effectively address the challenges that are designed to handle the complexities of semantic search in Large Language Models (LLMs) and generative Artificial Intelligence (AI). Unlike traditional data management systems, vector databases are capable of processing and searching through various types of data, including images, videos, text, audio, and other forms of unstructured data, by using the content of the data itself rather than labels or tags.

The limitations of Relational Database Management Systems (RDBMS) are well-documented, particularly their struggles with high-dimensional data representations and unstructured data common in AI applications. RDBMS often necessitate a time-consuming and error-prone process of flattening data into more manageable structures, leading to delays and inefficiencies in searches. Vector databases, however, are designed to circumvent these issues, offering a more efficient and accurate solution for managing and searching through complex and high-dimensional data, thus facilitating the advancement of AI applications.

This document serves as a comprehensive guide for customers who are currently using or planning to use vector databases, detailing the best practices for utilizing vector databases on platforms such as NetApp ONTAP, NetApp StorageGRID, Amazon FSxN for NetApp ONTAP, and SnapCenter. The content provided herein covers a range of topics:

- Infrastructure guidelines for vector databases, like Milvus, provided by NetApp storage through NetApp ONTAP and StorageGRID object storage.
- Validation of the Milvus database in AWS FSX for NetApp ONTAP through file and object store.
- Delves into NetApp's file-object duality, demonstrating its utility for data in vector databases as well as other applications.
- How NetApp's Data Protection Management product, SnapCenter, offers backup and restore functionalities for vector database data.
- How NetApp's Hybrid Cloud offers data replication and protection across on-premises and cloud environments.
- Provides insights into the performance validation of vector databases like Milvus and pgvector on NetApp ONTAP.
- Two specific use cases: Retrieval Augmented Generation (RAG) with Large Language Models(LLM) and the NetApp IT team's ChatAI, thereby offering practical examples of the concepts and practices outlined.

Solution Overview

Solution overview

This solution showcases the distinctive benefits and capabilities that NetApp brings to the table to tackle the challenges faced by vector database customers. By leveraging NetApp ONTAP, StorageGRID, NetApp's cloud solutions, and SnapCenter, customers can add significant value to their business operations. These tools not only address existing issues but also enhance efficiency and productivity, thereby contributing to overall business growth.

Why NetApp?

- NetApp's offerings, such as ONTAP and StorageGRID, allow for the separation of storage and compute, enabling optimal resource utilization based on specific requirements. This flexibility empowers customers to independently scale their storage using NetApp storage solutions.
- By leveraging NetApp's storage controllers, customers can efficiently serve data to their vector database using NFS and S3 protocols. These protocols facilitate customer data storage and manage the vector database index, eliminating the need for multiple copies of data accessed through file and object methods.
- NetApp ONTAP provides native support for NAS and Object storage across leading cloud service providers like AWS, Azure, and Google Cloud. This wide compatibility ensures seamless integration, enabling customer data mobility, global accessibility, disaster recovery, dynamic scalability, and high performance.
- With NetApp's robust data management capabilities, customers can rest assured knowing that their data is well-protected against potential risks and threats. NetApp prioritizes data security, offering peace of mind to

customers regarding the safety and integrity of their valuable information.

Vector Database

Vector Database

A vector database is a specialized type of database designed to handle, index, and search unstructured data using embeddings from machine learning models. Instead of organizing data in a traditional tabular format, it arranges data as high-dimensional vectors, also known as vector embeddings. This unique structure allows the database to handle complex, multi-dimensional data more efficiently and accurately.

One of the key capabilities of a vector database is its use of generative AI to perform analytics. This includes similarity searches, where the database identifies data points that are like a given input, and anomaly detection, where it can spot data points that deviate significantly from the norm.

Furthermore, vector databases are well-suited to handle temporal data, or time-stamped data. This type of data provides information about ‘what’ happened and when it happened, in sequence and in relation to all other events within a given IT system. This ability to handle and analyze temporal data makes vector databases particularly useful for applications that require an understanding of events over time.

Advantages of vector database for ML and AI:

- High-dimensional Search: Vector databases excel in managing and retrieving high-dimensional data, which is often generated in AI and ML applications.
- Scalability: They can efficiently scale to handle large volumes of data, supporting the growth and expansion of AI and ML projects.
- Flexibility: Vector databases offer a high degree of flexibility, allowing for the accommodation of diverse data types and structures.
- Performance: They provide high-performance data management and retrieval, critical for the speed and efficiency of AI and ML operations.
- Customizable Indexing: Vector databases offer customizable indexing options, enabling optimized data organization and retrieval based on specific needs.

Vector databases and use cases.

This section provides varies vector databases and their use case details.

Faiss and ScaNN

They are libraries that serve as crucial tools in the realm of vector search. These libraries provide functionality that is instrumental in managing and searching through vector data, making them invaluable resources in this specialized area of data management.

Elasticsearch

It’s a widely used search and analytics engine, has recently incorporated vector search capabilities. This new feature enhances its functionality, enabling it to handle and search through vector data more effectively.

Pinecone

It is a robust vector database with a unique set of features. It supports both dense and sparse vectors in its indexing functionality, which enhances its flexibility and adaptability. One of its key strengths lies in its ability to

combine traditional search methods with AI-based dense vector search, creating a hybrid search approach that leverages the best of both worlds.

Primarily cloud-based, Pinecone is designed for machine learning applications and integrates well with a variety of platforms, including GCP, AWS, Open AI, GPT-3, GPT-3.5, GPT-4, Catgut Plus, Elasticsearch, Haystack, and more. It's important to note that Pinecone is a closed-source platform and is available as a Software as a Service (SaaS) offering.

Given its advanced capabilities, Pinecone is particularly well-suited for the cybersecurity industry, where its high-dimensional search and hybrid search capabilities can be leveraged effectively to detect and respond to threats.

Chroma

It's a vector database that has a Core-API with four primary functions, one of which includes an in-memory document-vector store. It also utilizes the Face Transformers library to vectorize documents, enhancing its functionality and versatility.
Chroma is designed to operate both in the cloud and on-premises, offering flexibility based on user needs. Particularly, it excels in audio-related applications, making it an excellent choice for audio-based search engines, music recommendation systems, and other audio-related use cases.

Weaviate

It's a versatile vector database that allows users to vectorize their content using either its built-in modules or custom modules, providing flexibility based on specific needs. It offers both fully managed and self-hosted solutions, catering to a variety of deployment preferences.

One of Weaviate's key features is its ability to store both vectors and objects, enhancing its data handling capabilities. It is widely used for a range of applications, including semantic search and data classification in ERP systems. In the e-commerce sector, it powers search and recommendation engines. Weaviate is also used for image search, anomaly detection, automated data harmonization, and cybersecurity threat analysis, showcasing its versatility across multiple domains.

Redis

Redis is a high-performing vector database known for its fast in-memory storage, offering low latency for read-write operations. This makes it an excellent choice for recommendation systems, search engines, and data analytics applications that require quick data access.

Redis supports various data structures for vectors, including lists, sets, and sorted sets. It also provides vector operations such as calculating distances between vectors or finding intersections and unions. These features are particularly useful for similarity search, clustering, and content-based recommendation systems.

In terms of scalability and availability, Redis excels in handling high throughput workloads and offers data replication. It also integrates well with other data types, including traditional relational databases (RDBMS). Redis includes a Publish/Subscribe (Pub/Sub) feature for real-time updates, which is beneficial for managing real-time vectors. Moreover, Redis is lightweight and simple to use, making it a user-friendly solution for managing vector data.

Milvus

It's a versatile vector database that offers an API like a document store, much like MongoDB. It stands out due to its support for a wide variety of data types, making it a popular choice in the data science and machine learning fields.

One of Milvus' unique features is its multi-vectorization capability, which allows users to specify at runtime the type of vector to use for the search. Furthermore, it utilizes Knowwhere, a library that sits atop other libraries like Faiss, to manage communication between queries and the vector search algorithms.

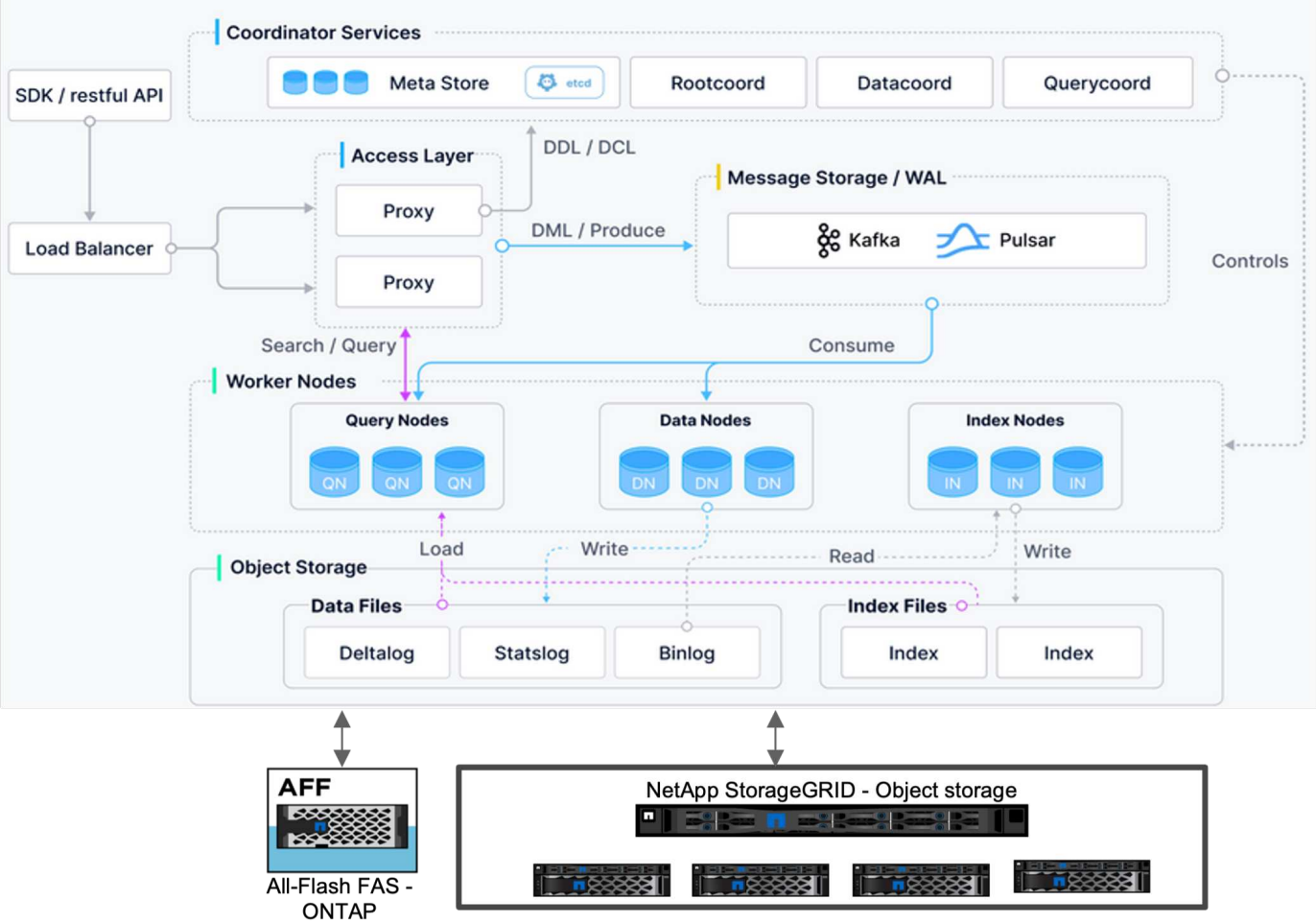
Milvus also offers seamless integration with machine learning workflows, thanks to its compatibility with PyTorch and TensorFlow. This makes it an excellent tool for a range of applications, including e-commerce, image and video analysis, object recognition, image similarity search, and content-based image retrieval. In the realm of natural language processing, Milvus is used for document clustering, semantic search, and question-answering systems.

For this solution, we picked milvus for the solution validation. For performance, we used both milvus and postgres(pgvecto.rs).

Why we chose milvus for this solution?

- Open-Source: Milvus is an open-source vector database, encouraging community-driven development and improvements.
- AI Integration: It leverages embedding similarity search and AI applications to enhance vector database functionality.
- Large Volume Handling: Milvus has the capacity to store, index, and manage over a billion embedding vectors generated by Deep Neural Networks (DNN) and Machine Learning (ML) models.
- User-Friendly: It is easy to use, with setup taking less than a minute. Milvus also offers SDKs for different programming languages.
- Speed: It offers blazing fast retrieval speeds, up to 10 times faster than some alternatives.
- Scalability and Availability: Milvus is highly scalable, with options to scale up and out as needed.
- Feature-Rich: It supports different data types, attribute filtering, User-Defined Function (UDF) support, configurable consistency levels, and travel time, making it a versatile tool for various applications.

Milvus architecture overview



This section provides higher lever components and services are used in Milvus architecture.
* Access layer – It's composed of a group of stateless proxies and serves as the front layer of the system and endpoint to users.
* Coordinator service – it assigns the tasks to the worker nodes and act as a system's brain. It has three coordinator types: root coord,data coord and query coord.
* Worker nodes : It follows the instruction from coordinator service and execute user triggered DML/DDL commands.it has three types of worker nodes such as query node, data node and index node.
* Storage: it's responsible for data persistence. It comprises meta storage, log broker, and object storage. NetApp storage such as ONTAP and StorageGRID provides object storage and File based storage to Milvus for both customer data and vector database data.

Technology Requirement

Technology Requirement

The hardware and software configurations outlined below were utilized for the majority of the validations performed in this document, with the exception of performance. These configurations serve as a guideline to help you set up your environment. However, please note that the specific components may vary depending on individual customer requirements.

Hardware requirements

Hardware	Details
NetApp AFF Storage array HA Pair	* A800 * ONTAP 9.14.1 * 48 x 3.49TB SSD-NVM * Two Flexible group volumes: metadata and data. * Metadata NFS volume has 12 x Persistent Volumes with 250GB. * Data is a ONTAP NAS S3 volume
6 x FUJITSU PRIMERGY RX2540 M4	* 64 CPUs * Intel® Xeon® Gold 6142 CPU @ 2.60GHz * 256 GM Physical Memory * 1 x 100GbE network port
Networking	100 GbE
StorageGRID	* 1 x SG100, 3xSGF6024 * 3 x 24 x 7.68TB

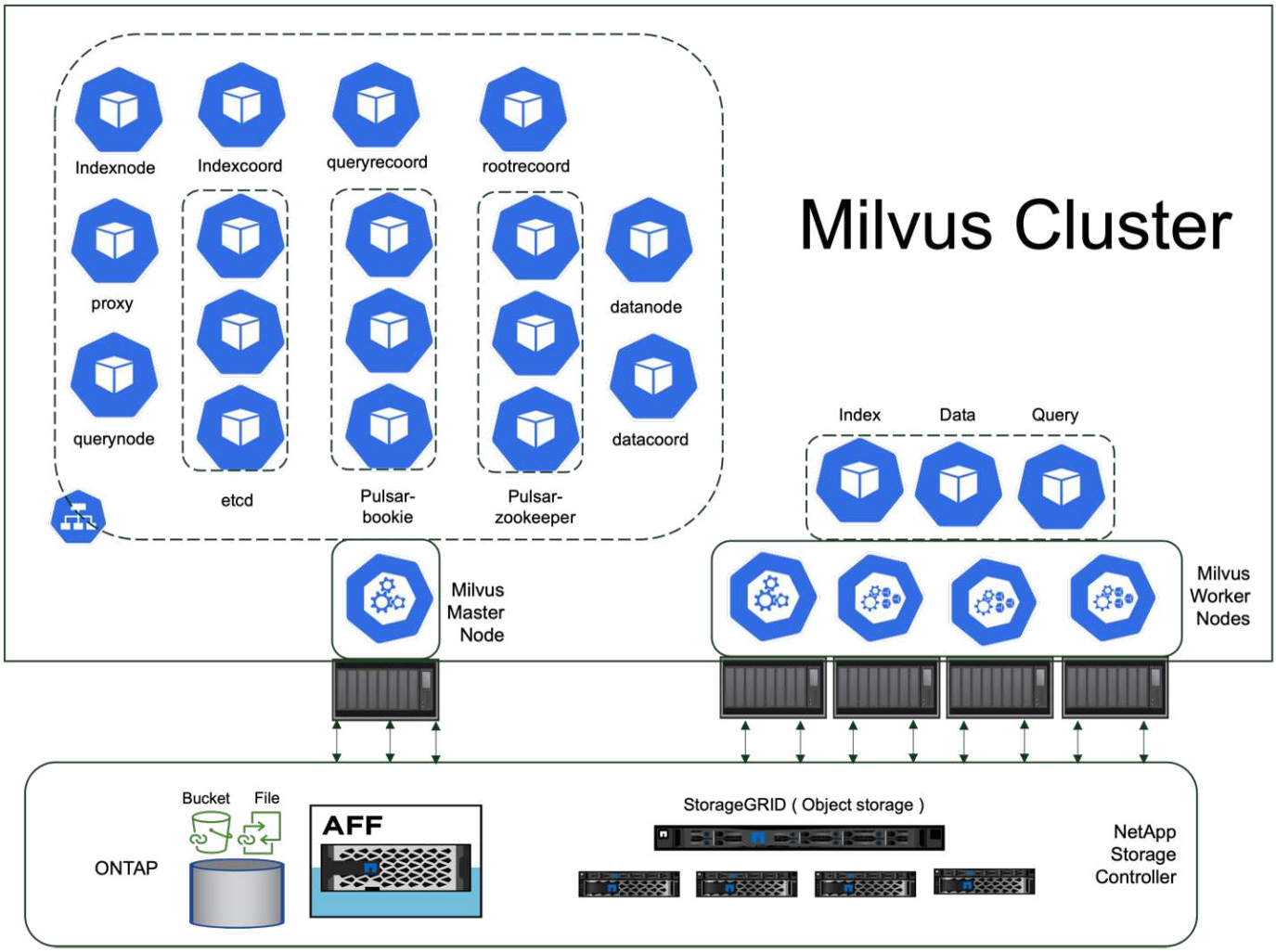
Software requirements

Software	Details
Milvus cluster	* CHART - milvus-4.1.11. * APP Version – 2.3.4 * Dependent bundles such as bookkeeper, zookeeper, pulsar, etcd, proxy, querynode, worker
Kubernetes	* 5 node K8s cluster * 1 Master node and 4 Worker nodes * Version – 1.7.2
Python	*3.10.12.

Deployment Procedure

Deployment procedure

In this deployment section, we used milvus vector database with Kubernetes for the lab setup as below.



The netapp storage provides the storage for the cluster to keep customers data and milvus cluster data.

NetApp storage setup – ONTAP

- Storage system initialization
- Storage virtual machine (SVM) creation
- Assignment of logical network interfaces
- NFS, S3 configuration and licensing

Please follow the steps below for NFS (Network File System):

1. Create a FlexGroup volume for NFSv4. In our set up for this validation, we have used 48 SSDs, 1 SSD dedicated for the controller's root volume and 47 SSDs spread across for NFSv4]].Verify that the NFS export policy for the FlexGroup volume has read/write permissions for the Kubernetes (K8s) nodes network. If these permissions are not in place, grant read/write (rw) permissions for the K8s nodes network.
2. On all K8s nodes, create a folder and mount the FlexGroup volume onto this folder through a Logical Interface (LIF) on each K8s nodes.

Please follow the steps below for NAS S3 (Network Attached Storage Simple Storage Service):

1. Create a FlexGroup volume for NFS.

2. Set up an object-store-server with HTTP enabled and the admin status set to 'up' using the "vserver object-store-server create" command. You have the option to enable HTTPS and set a custom listener port.
3. Create an object-store-server user using the "vserver object-store-server user create -user <username>" command.
4. To obtain the access key and secret key, you can run the following command: "set diag; vserver object-store-server user show -user <username>". However, moving forward, these keys will be supplied during the user creation process or can be retrieved using REST API calls.
5. Establish an object-store-server group using the user created in step 2 and grant access. In this example, we have provided "FullAccess".
6. Create a NAS bucket by setting its type to "nas" and supplying the path to the NFSv3 volume. It's also possible to utilize an S3 bucket for this purpose.

NetApp storage setup – StorageGRID

1. Install the storageGRID software.
2. Create a tenant and bucket.
3. Create user with required permission.

Please check more details in https://docs.netapp.com/us-en/storagegrid-116/primer/index.html

Solution Overview

We have conducted a comprehensive solution validation focused on five key areas, the details of which are outlined below. Each section delves into the challenges faced by customers, the solutions provided by NetApp, and the subsequent benefits to the customer.

1. Milvus cluster setup with Kubernetes in on-premises
   Customer challenges to scale independently on storage and compute, effective infrastructure management and data management. In this section, we detail the process of installing a Milvus cluster on Kubernetes, utilizing a NetApp storage controller for both cluster data and customer data.
2. Milvus with Amazon FSxN for NetApp ONTAP – file and object duality
   In this section, Why we need to deploy vector database in cloud as well as steps to deploy vector database (milvus standalone) in Amazon FSxN for NetApp ONTAP within docker containers.
3. Vector database protection using NetApp SnapCenter.
   In this section, we delve into how SnapCenter safeguards the vector database data and Milvus data residing in ONTAP. For this example, we utilized a NAS bucket (milvusdbvol1) derived from an NFS ONTAP volume (vol1) for customer data, and a separate NFS volume (vectordbpv) for Milvus cluster configuration data.
4. Disaster Recovery using NetApp SnapMirror
   In this section, we discuss about the importance of Disaster recovery(DR) for vector database and how netapp disaster recovery product snapmirror provides DR solution to vector database.
5. Performance validation
   In this section, we aim to delve into the performance validation of vector databases, such as Milvus and pgvecto.rs, focusing on their storage performance characteristics such as I/O profile and netapp storage controller behavious in support of RAG and inference workloads within the LLM Lifecycle. We will evaluate and identify any performance differentiators when these databases are combined with the ONTAP storage solution. Our analysis will be based on key performance indicators, such as the number of queries processed per second(QPS).

Milvus Cluster Setup with Kubernetes in on-premises

Milvus cluster setup with Kubernetes in on-premises

Customer challenges to scale independently on storage and compute, effective infrastructure management and data management,
Kubernetes and vector databases together form a powerful, scalable solution for managing large data operations. Kubernetes optimizes resources and manages containers, while vector databases efficiently handle high-dimensional data and similarity searches. This combination enables swift processing of complex queries on large datasets and seamlessly scales with growing data volumes, making it ideal for big data applications and AI workloads.

1. In this section, we detail the process of installing a Milvus cluster on Kubernetes, utilizing a NetApp storage controller for both cluster data and customer data.
2. To install a Milvus cluster, Persistent Volumes (PVs) are required for storing data from various Milvus cluster components. These components include etcd (three instances), pulsar-bookie-journal (three instances), pulsar-bookie-ledgers (three instances), and pulsar-zookeeper-data (three instances).
3. We created a single NFS volume from NetApp ONTAP and established 12 persistent volumes, each with 250GB of storage. The storage size can vary depending on the cluster size; for instance, we have another cluster where each PV has 50GB. Please refer below to one of the PV YAML files for more details; we had 12 such files in total. In each file, the storageClassName is set to 'default', and the storage and path are unique to each PV.

```
root@node2:~# cat sai_nfs_to_default_pv1.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: karthik-pv1
spec:
  capacity:
    storage: 250Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: default
  local:
    path: /vectordbsc/milvus/milvus1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - node2
          - node3
          - node4
          - node5
          - node6
root@node2:~#
```

4. Execute the 'kubectl apply' command for each PV YAML file to create the Persistent Volumes, and then verify their creation using ‘kubectl get pv’

```
root@node2:~# for i in $( seq 1 12 ); do kubectl apply -f
sai_nfs_to_default_pv$i.yaml; done
persistentvolume/karthik-pv1 created
persistentvolume/karthik-pv2 created
persistentvolume/karthik-pv3 created
persistentvolume/karthik-pv4 created
persistentvolume/karthik-pv5 created
persistentvolume/karthik-pv6 created
persistentvolume/karthik-pv7 created
persistentvolume/karthik-pv8 created
persistentvolume/karthik-pv9 created
persistentvolume/karthik-pv10 created
persistentvolume/karthik-pv11 created
persistentvolume/karthik-pv12 created
root@node2:~#
```

5. For storing customer data, Milvus supports object storage solutions such as MinIO, Azure Blob, and S3. In this guide, we utilize S3. The following steps apply to both ONTAP S3 and StorageGRID object store. We use Helm to deploy the Milvus cluster. Download the configuration file, values.yaml, from the Milvus download location. Please refer to the appendix for the values.yaml file we used in this document.
6. Ensure that the 'storageClass' is set to 'default' in each section, including those for the log, etcd, zookeeper, and bookkeeper.
7. In the MinIO section, disable MinIO.
8. Create a NAS bucket from ONTAP or StorageGRID object storage and include them in an External S3 with the object storage credentials.

```
###################################
# External S3
# - these configs are only used when `externalS3.enabled` is true
###################################
externalS3:
  enabled: true
  host: "192.168.150.167"
  port: "80"
  accessKey: "24G4C1316APP2BIPDE5S"
  secretKey: "Zd28p43rgZaU44PX_ftT279z9nt4jBSro97j87Bx"
  useSSL: false
  bucketName: "milvusdbvol1"
  rootPath: ""
  useIAM: false
  cloudProvider: "aws"
  iamEndpoint: ""
  region: ""
  useVirtualHost: false
```

9. Before creating the Milvus cluster, ensure that the PersistentVolumeClaim (PVC) does not have any pre-existing resources.

```
root@node2:~# kubectl get pvc
No resources found in default namespace.
root@node2:~#
```

10. Utilize Helm and the values.yaml configuration file to install and start the Milvus cluster.

```
root@node2:~# helm upgrade --install my-release milvus/milvus --set
global.storageClass=default  -f values.yaml
Release "my-release" does not exist. Installing it now.
NAME: my-release
LAST DEPLOYED: Thu Mar 14 15:00:07 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
root@node2:~#
```

11. Verify the status of the PersistentVolumeClaims (PVCs).

```
root@node2:~# kubectl get pvc
NAME                                                                    STATUS
VOLUME         CAPACITY   ACCESS MODES   STORAGECLASS   AGE
data-my-release-etcd-0                                                  Bound
karthik-pv8    250Gi      RWO            default        3s
data-my-release-etcd-1                                                  Bound
karthik-pv5    250Gi      RWO            default        2s
data-my-release-etcd-2                                                  Bound
karthik-pv4    250Gi      RWO            default        3s
my-release-pulsar-bookie-journal-my-release-pulsar-bookie-0      Bound
karthik-pv10   250Gi      RWO            default        3s
my-release-pulsar-bookie-journal-my-release-pulsar-bookie-1      Bound
karthik-pv3    250Gi      RWO            default        3s
my-release-pulsar-bookie-journal-my-release-pulsar-bookie-2      Bound
karthik-pv1    250Gi      RWO            default        3s
my-release-pulsar-bookie-ledgers-my-release-pulsar-bookie-0      Bound
karthik-pv2    250Gi      RWO            default        3s
my-release-pulsar-bookie-ledgers-my-release-pulsar-bookie-1      Bound
karthik-pv9    250Gi      RWO            default        3s
my-release-pulsar-bookie-ledgers-my-release-pulsar-bookie-2      Bound
karthik-pv11   250Gi      RWO            default        3s
my-release-pulsar-zookeeper-data-my-release-pulsar-zookeeper-0   Bound
karthik-pv7    250Gi      RWO            default        3s
root@node2:~#
```

12. Check the status of the pods.

```
root@node2:~# kubectl get pods -o wide
NAME                                                READY   STATUS
RESTARTS        AGE    IP              NODE    NOMINATED NODE
READINESS GATES
<content removed to save page space>
```

Please make sure the pods status are ‘running’ and working as expected

13. Test data writing and reading in Milvus and NetApp object storage.
    - Write data using the "prepare_data_netapp_new.py" Python program.

```
root@node2:~# date;python3 prepare_data_netapp_new.py ;date
Thu Apr  4 04:15:35 PM UTC 2024
=== start connecting to Milvus     ===
=== Milvus host: localhost         ===
Does collection hello_milvus_ntapnew_update2_sc exist in Milvus:
False
=== Drop collection - hello_milvus_ntapnew_update2_sc ===
=== Drop collection - hello_milvus_ntapnew_update2_sc2 ===
=== Create collection `hello_milvus_ntapnew_update2_sc` ===
=== Start inserting entities       ===
Number of entities in hello_milvus_ntapnew_update2_sc: 3000
Thu Apr  4 04:18:01 PM UTC 2024
root@node2:~#
```

- Read the data using the "verify_data_netapp.py" Python file.

```
root@node2:~# python3 verify_data_netapp.py
=== start connecting to Milvus     ===
=== Milvus host: localhost         ===

Does collection hello_milvus_ntapnew_update2_sc exist in Milvus: True
{'auto_id': False, 'description': 'hello_milvus_ntapnew_update2_sc',
'fields': [{'name': 'pk', 'description': '', 'type': <DataType.INT64:
5>, 'is_primary': True, 'auto_id': False}, {'name': 'random',
'description': '', 'type': <DataType.DOUBLE: 11>}, {'name': 'var',
'description': '', 'type': <DataType.VARCHAR: 21>, 'params':
{'max_length': 65535}}, {'name': 'embeddings', 'description': '',
'type': <DataType.FLOAT_VECTOR: 101>, 'params': {'dim': 16}}]}
Number of entities in Milvus: hello_milvus_ntapnew_update2_sc : 3000

=== Start Creating index IVF_FLAT  ===

=== Start loading                  ===

=== Start searching based on vector similarity ===

hit: id: 2998, distance: 0.0, entity: {'random': 0.9728033590489911},
random field: 0.9728033590489911
hit: id: 2600, distance: 0.602496862411499, entity: {'random':
0.3098157043984633}, random field: 0.3098157043984633
hit: id: 1831, distance: 0.6797959804534912, entity: {'random':
0.6331477114129169}, random field: 0.6331477114129169
hit: id: 2999, distance: 0.0, entity: {'random':
0.02316334456872482}, random field: 0.02316334456872482
hit: id: 2524, distance: 0.5918987989425659, entity: {'random':
```

```
0.285283165889066}, random field: 0.285283165889066
hit: id: 264, distance: 0.7254047393798828, entity: {'random':
0.3329096143562196}, random field: 0.3329096143562196
search latency = 0.4533s

=== Start querying with `random > 0.5` ===

query result:
-{'random': 0.6378742006852851, 'embeddings': [0.20963514,
0.39746657, 0.12019053, 0.6947492, 0.9535575, 0.5454552, 0.82360446,
0.21096309, 0.52323616, 0.8035404, 0.77824664, 0.80369574, 0.4914803,
0.8265614, 0.6145269, 0.80234545], 'pk': 0}
search latency = 0.4476s

=== Start hybrid searching with `random > 0.5` ===

hit: id: 2998, distance: 0.0, entity: {'random': 0.9728033590489911},
random field: 0.9728033590489911
hit: id: 1831, distance: 0.6797959804534912, entity: {'random':
0.6331477114129169}, random field: 0.6331477114129169
hit: id: 678, distance: 0.7351570129394531, entity: {'random':
0.5195484662306603}, random field: 0.5195484662306603
hit: id: 2644, distance: 0.8620758056640625, entity: {'random':
0.9785952878381153}, random field: 0.9785952878381153
hit: id: 1960, distance: 0.9083120226860046, entity: {'random':
0.6376039340439571}, random field: 0.6376039340439571
hit: id: 106, distance: 0.9792704582214355, entity: {'random':
0.9679994241326673}, random field: 0.9679994241326673
search latency = 0.1232s
Does collection hello_milvus_ntapnew_update2_sc2 exist in Milvus:
True
{'auto_id': True, 'description': 'hello_milvus_ntapnew_update2_sc2',
'fields': [{'name': 'pk', 'description': '', 'type': <DataType.INT64:
5>, 'is_primary': True, 'auto_id': True}, {'name': 'random',
'description': '', 'type': <DataType.DOUBLE: 11>}, {'name': 'var',
'description': '', 'type': <DataType.VARCHAR: 21>, 'params':
{'max_length': 65535}}, {'name': 'embeddings', 'description': '',
'type': <DataType.FLOAT_VECTOR: 101>, 'params': {'dim': 16}}]}
```

Based on the above validation, the integration of Kubernetes with a vector database, as demonstrated through the deployment of a Milvus cluster on Kubernetes using a NetApp storage controller, offers customers a robust, scalable, and efficient solution for managing large-scale data operations. This setup provides customers with the ability to handle high-dimensional data and execute complex queries rapidly and efficiently, making it an ideal solution for big data applications and AI workloads. The use of Persistent Volumes (PVs) for various cluster components, along with the creation of a single NFS volume from NetApp ONTAP, ensures optimal resource utilization and data management. The process of verifying the status of PersistentVolumeClaims (PVCs) and pods, as well as testing data writing and

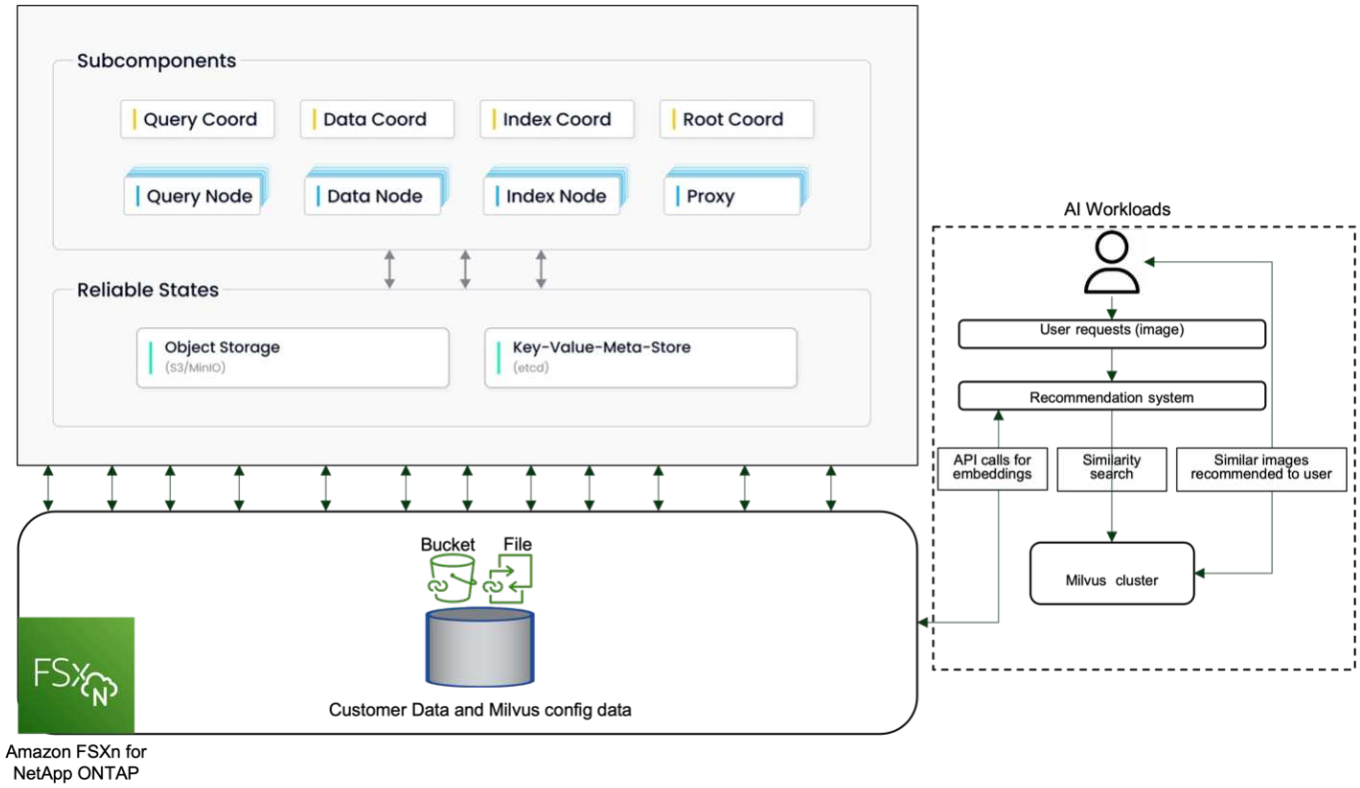
reading, provides customers with the assurance of reliable and consistent data operations. The use of ONTAP or StorageGRID object storage for customer data further enhances data accessibility and security. Overall, this setup empowers customers with a resilient and high-performing data management solution that can seamlessly scale with their growing data needs.

Milvus with Amazon FSxN for NetApp ONTAP - file and object duality

Milvus with Amazon FSxN for NetApp ONTAP – file and object duality

In this section, Why we need to deploy vector database in cloud as well as steps to deploy vector database (milvus standalone) in Amazon FSxN for NetApp ONTAP within docker containers.

Deploying a vector database in the cloud provides several significant benefits, particularly for applications that require handling high-dimensional data and executing similarity searches. First, cloud-based deployment offers scalability, allowing for the easy adjustment of resources to match the growing data volumes and query loads. This ensures that the database can efficiently handle increased demand while maintaining high performance. Second, cloud deployment provides high availability and disaster recovery, as data can be replicated across different geographical locations, minimizing the risk of data loss, and ensuring continuous service even during unexpected events. Third, it provides cost-effectiveness, as you only pay for the resources you use, and can scale up or down based on demand, avoiding the need for substantial upfront investment in hardware. Finally, deploying a vector database in the cloud can enhance collaboration, as data can be accessed and shared from anywhere, facilitating team-based work and data-driven decision making.
Please check the architecture of the milvus standalone with Amazon FSxN for NetApp ONTAP used in this validation.



1. Create an Amazon FSxN for NetApp ONTAP instance and note down the details of the VPC, VPC security groups, and subnet. This information will be required when creating an EC2 instance. You can find more details here - https://us-east-1.console.aws.amazon.com/fsx/home?region=us-east-1#file-system-create
2. Create an EC2 instance, ensuring that the VPC, Security Groups, and subnet match those of the Amazon

FSxN for NetApp ONTAP instance.

3. Install nfs-common using the command 'apt-get install nfs-common' and update the package information using 'sudo apt-get update'.
4. Create a mount folder and mount the Amazon FSxN for NetApp ONTAP on it.

```
ubuntu@ip-172-31-29-98:~$ mkdir /home/ubuntu/milvusvectordb
ubuntu@ip-172-31-29-98:~$ sudo mount 172.31.255.228:/vol1
/home/ubuntu/milvusvectordb
ubuntu@ip-172-31-29-98:~$ df -h /home/ubuntu/milvusvectordb
Filesystem            Size  Used Avail Use% Mounted on
172.31.255.228:/vol1  973G  126G  848G  13% /home/ubuntu/milvusvectordb
ubuntu@ip-172-31-29-98:~$
```

5. Install Docker and Docker Compose using 'apt-get install'.
6. Set up a Milvus cluster based on the docker-compose.yaml file, which can be downloaded from the Milvus website.

```
root@ip-172-31-22-245:~# wget https://github.com/milvus-
io/milvus/releases/download/v2.0.2/milvus-standalone-docker-compose.yml
-O docker-compose.yml
--2024-04-01 14:52:23--  https://github.com/milvus-
io/milvus/releases/download/v2.0.2/milvus-standalone-docker-compose.yml
<removed some output to save page space>
```

7. In the 'volumes' section of the docker-compose.yml file, map the NetApp NFS mount point to the corresponding Milvus container path, specifically in etcd, minio, and standalone.Check Appendix D: docker-compose.yml for details about changes in yml
8. Verify the mounted folders and files.

```
ubuntu@ip-172-31-29-98:~/milvusvectordb$ ls -ltrh
/home/ubuntu/milvusvectordb
total 8.0K
-rw-r--r-- 1 root root 1.8K Apr  2 16:35 s3_access.py
drwxrwxrwx 2 root root 4.0K Apr  4 20:19 volumes
ubuntu@ip-172-31-29-98:~/milvusvectordb$ ls -ltrh
/home/ubuntu/milvusvectordb/volumes/
total 0
ubuntu@ip-172-31-29-98:~/milvusvectordb$ cd
ubuntu@ip-172-31-29-98:~$ ls
docker-compose.yml  docker-compose.yml~  milvus.yaml  milvusvectordb
vectordbvol1
ubuntu@ip-172-31-29-98:~$
```

9. Run 'docker-compose up -d' from the directory containing the docker-compose.yml file.
10. Check the status of the Milvus container.

```
ubuntu@ip-172-31-29-98:~$ sudo docker-compose ps
      Name                     Command                  State
Ports
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
----------
milvus-etcd         etcd -advertise-client-url ...   Up (healthy)
2379/tcp, 2380/tcp
milvus-minio        /usr/bin/docker-entrypoint ...   Up (healthy)
0.0.0.0:9000->9000/tcp,:::9000->9000/tcp, 0.0.0.0:9001-
>9001/tcp,:::9001->9001/tcp
milvus-standalone   /tini -- milvus run standalone   Up (healthy)
0.0.0.0:19530->19530/tcp,:::19530->19530/tcp, 0.0.0.0:9091-
>9091/tcp,:::9091->9091/tcp
ubuntu@ip-172-31-29-98:~$
ubuntu@ip-172-31-29-98:~$ ls -ltrh /home/ubuntu/milvusvectordb/volumes/
total 12K
drwxr-xr-x 3 root root 4.0K Apr  4 20:21 etcd
drwxr-xr-x 4 root root 4.0K Apr  4 20:21 minio
drwxr-xr-x 5 root root 4.0K Apr  4 20:21 milvus
ubuntu@ip-172-31-29-98:~$
```

11. To validate the read and write functionality of vector database and it's data in Amazon FSxN for NetApp ONTAP, we used the Python Milvus SDK and a sample program from PyMilvus. Install the necessary packages using 'apt-get install python3-numpy python3-pip' and install PyMilvus using 'pip3 install pymilvus'.
12. Validate data writing and reading operations from Amazon FSxN for NetApp ONTAP in the vector database.

```
root@ip-172-31-29-98:~/pymilvus/examples# python3
prepare_data_netapp_new.py
=== start connecting to Milvus     ===
=== Milvus host: localhost         ===
Does collection hello_milvus_ntapnew_sc exist in Milvus: True
=== Drop collection - hello_milvus_ntapnew_sc ===
=== Drop collection - hello_milvus_ntapnew_sc2 ===
=== Create collection `hello_milvus_ntapnew_sc` ===
=== Start inserting entities       ===
Number of entities in hello_milvus_ntapnew_sc: 9000
root@ip-172-31-29-98:~/pymilvus/examples# find
/home/ubuntu/milvusvectordb/
…
```

```
<removed content to save page space >
…
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/103/448789845791411923/b3def25f-c117-4fba-8256-96cb7557cd6c
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/103/448789845791411923/b3def25f-c117-4fba-8256-96cb7557cd6c/part.1
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/103/448789845791411923/xl.meta
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/0
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/0/448789845791411924
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/0/448789845791411924/xl.meta
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/1
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/1/448789845791411925
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/1/448789845791411925/xl.meta
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/100
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/100/448789845791411920
/home/ubuntu/milvusvectordb/volumes/minio/a-bucket/files/insert_log
/448789845791611912/448789845791611913/448789845791611939/100/448789845791411920/xl.meta
```

13. Check the reading operation using the verify_data_netapp.py script.

```
root@ip-172-31-29-98:~/pymilvus/examples# python3 verify_data_netapp.py
=== start connecting to Milvus     ===

=== Milvus host: localhost         ===

Does collection hello_milvus_ntapnew_sc exist in Milvus: True
{'auto_id': False, 'description': 'hello_milvus_ntapnew_sc', 'fields':
```

```
[{'name': 'pk', 'description': '', 'type': <DataType.INT64: 5>,
'is_primary': True, 'auto_id': False}, {'name': 'random', 'description':
'', 'type': <DataType.DOUBLE: 11>}, {'name': 'var', 'description': '',
'type': <DataType.VARCHAR: 21>, 'params': {'max_length': 65535}},
{'name': 'embeddings', 'description': '', 'type': <DataType.
FLOAT_VECTOR: 101>, 'params': {'dim': 8}}], 'enable_dynamic_field':
False}
Number of entities in Milvus: hello_milvus_ntapnew_sc : 9000

=== Start Creating index IVF_FLAT  ===


=== Start loading                  ===


=== Start searching based on vector similarity ===

hit: id: 2248, distance: 0.0, entity: {'random': 0.2777646777746381},
random field: 0.2777646777746381
hit: id: 4837, distance: 0.07805602252483368, entity: {'random':
0.6451650959930306}, random field: 0.6451650959930306
hit: id: 7172, distance: 0.07954417169094086, entity: {'random':
0.6141351712303128}, random field: 0.6141351712303128
hit: id: 2249, distance: 0.0, entity: {'random': 0.7434908973629817},
random field: 0.7434908973629817
hit: id: 830, distance: 0.05628090724349022, entity: {'random':
0.8544487225667627}, random field: 0.8544487225667627
hit: id: 8562, distance: 0.07971227169036865, entity: {'random':
0.4464554280115878}, random field: 0.4464554280115878
search latency = 0.1266s

=== Start querying with `random > 0.5` ===

query result:
-{'random': 0.6378742006852851, 'embeddings': [0.3017092, 0.74452263,
0.8009826, 0.4927033, 0.12762444, 0.29869467, 0.52859956, 0.23734547],
'pk': 0}
search latency = 0.3294s

=== Start hybrid searching with `random > 0.5` ===

hit: id: 4837, distance: 0.07805602252483368, entity: {'random':
0.6451650959930306}, random field: 0.6451650959930306
hit: id: 7172, distance: 0.07954417169094086, entity: {'random':
0.6141351712303128}, random field: 0.6141351712303128
hit: id: 515, distance: 0.09590047597885132, entity: {'random':
0.8013175797590888}, random field: 0.8013175797590888
```

```
hit: id: 2249, distance: 0.0, entity: {'random': 0.7434908973629817},
random field: 0.7434908973629817
hit: id: 830, distance: 0.05628090724349022, entity: {'random':
0.8544487225667627}, random field: 0.8544487225667627
hit: id: 1627, distance: 0.08096684515476227, entity: {'random':
0.9302397069516164}, random field: 0.9302397069516164
search latency = 0.2674s
Does collection hello_milvus_ntapnew_sc2 exist in Milvus: True
{'auto_id': True, 'description': 'hello_milvus_ntapnew_sc2', 'fields':
[{'name': 'pk', 'description': '', 'type': <DataType.INT64: 5>,
'is_primary': True, 'auto_id': True}, {'name': 'random', 'description':
'', 'type': <DataType.DOUBLE: 11>}, {'name': 'var', 'description': '',
'type': <DataType.VARCHAR: 21>, 'params': {'max_length': 65535}},
{'name': 'embeddings', 'description': '', 'type': <DataType.
FLOAT_VECTOR: 101>, 'params': {'dim': 8}}], 'enable_dynamic_field':
False}
```

14. If the customer wants to access (read) NFS data tested in the vector database via the S3 protocol for AI workloads, this can be validated using a straightforward Python program. An example of this could be a similarity search of images from another application as mentioned in the picture that is in the beginning of this section.

```
root@ip-172-31-29-98:~/pymilvus/examples# sudo python3
/home/ubuntu/milvusvectordb/s3_access.py -i 172.31.255.228 --bucket
milvusnasvol --access-key PY6UF318996I86NBYNDD --secret-key
hoPctr9aD88c1j0SkIYZ2uPa03vlbqKA0c5feK6F
OBJECTS in the bucket milvusnasvol are :
************************************
…
<output content removed to save page space>
…
bucket/files/insert_log/448789845791611912/448789845791611913/4487898457
91611920/0/448789845791411917/xl.meta
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611920/1/448789845791411918/xl.meta
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611920/100/448789845791411913/xl.meta
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611920/101/448789845791411914/xl.meta
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611920/102/448789845791411915/xl.meta
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611920/103/448789845791411916/1c48ab6e-
1546-4503-9084-28c629216c33/part.1
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611920/103/448789845791411916/xl.meta
```

```
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611939/0/448789845791411924/xl.meta
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611939/1/448789845791411925/xl.meta
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611939/100/448789845791411920/xl.meta
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611939/101/448789845791411921/xl.meta
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611939/102/448789845791411922/xl.meta
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611939/103/448789845791411923/b3def25f-
c117-4fba-8256-96cb7557cd6c/part.1
volumes/minio/a-bucket/files/insert_log/448789845791611912
/448789845791611913/448789845791611939/103/448789845791411923/xl.meta
volumes/minio/a-bucket/files/stats_log/448789845791211880
/448789845791211881/448789845791411889/100/1/xl.meta
volumes/minio/a-bucket/files/stats_log/448789845791211880
/448789845791211881/448789845791411889/100/448789845791411912/xl.meta
volumes/minio/a-bucket/files/stats_log/448789845791611912
/448789845791611913/448789845791611920/100/1/xl.meta
volumes/minio/a-bucket/files/stats_log/448789845791611912
/448789845791611913/448789845791611920/100/448789845791411919/xl.meta
volumes/minio/a-bucket/files/stats_log/448789845791611912
/448789845791611913/448789845791611939/100/1/xl.meta
volumes/minio/a-bucket/files/stats_log/448789845791611912
/448789845791611913/448789845791611939/100/448789845791411926/xl.meta
************************************
root@ip-172-31-29-98:~/pymilvus/examples#
```

This section effectively demonstrates how customers can deploy and operate a standalone Milvus setup within Docker containers, utilizing Amazon's NetApp FSxN for NetApp ONTAP data storage. This setup allows customers to leverage the power of vector databases for handling high-dimensional data and executing complex queries, all within the scalable and efficient environment of Docker containers. By creating an Amazon FSxN for NetApp ONTAP instance and matching EC2 instance, customers can ensure optimal resource utilization and data management. The successful validation of data writing and reading operations from FSxN in the vector database provides customers with the assurance of reliable and consistent data operations. Additionally, the ability to list (read) data from AI workloads via the S3 protocol offers enhanced data accessibility. This comprehensive process, therefore, provides customers with a robust and efficient solution for managing their large-scale data operations, leveraging the capabilities of Amazon's FSxN for NetApp ONTAP.

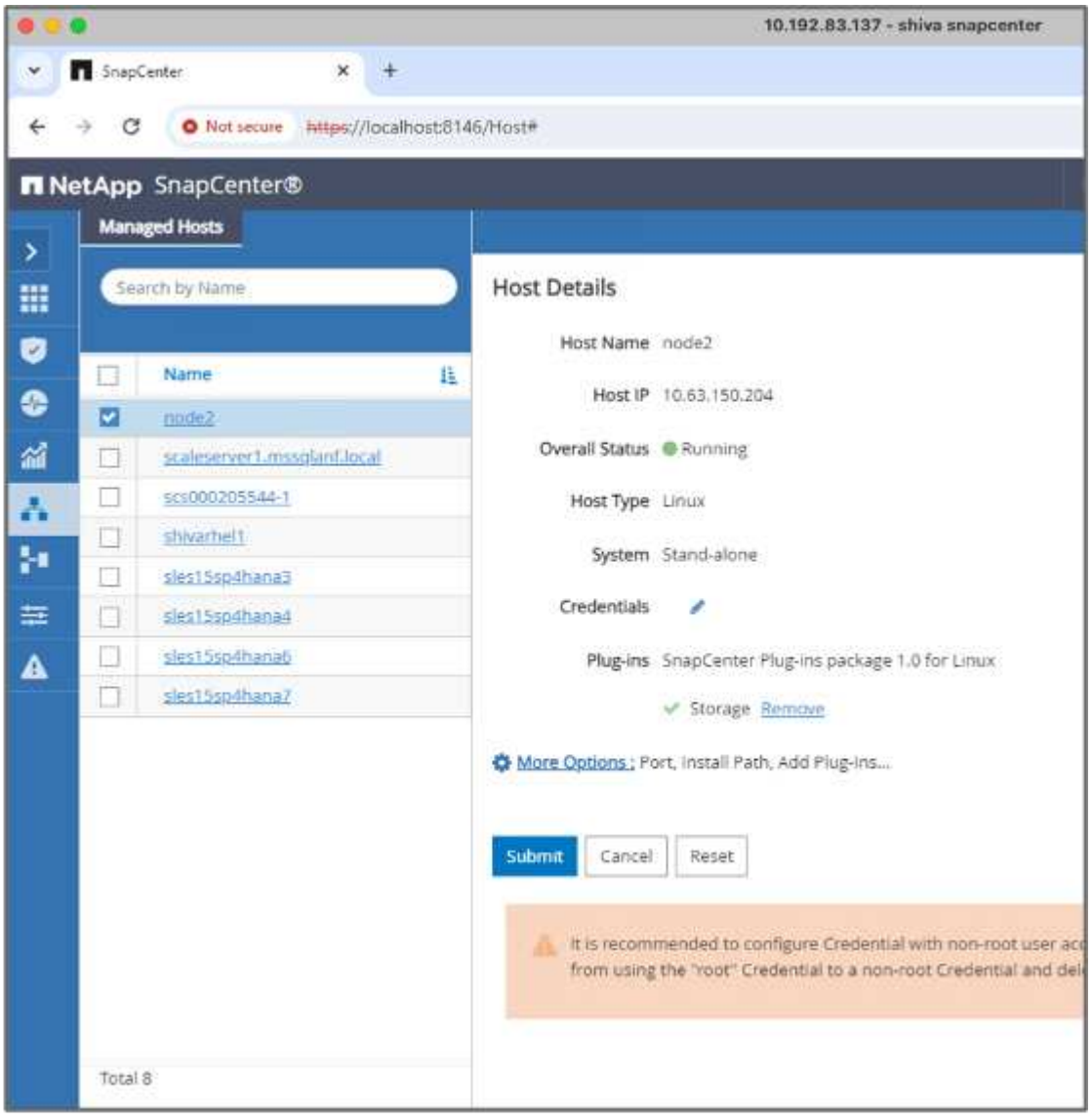
Vector Database Protection using SnapCenter

Vector database protection using NetApp SnapCenter.

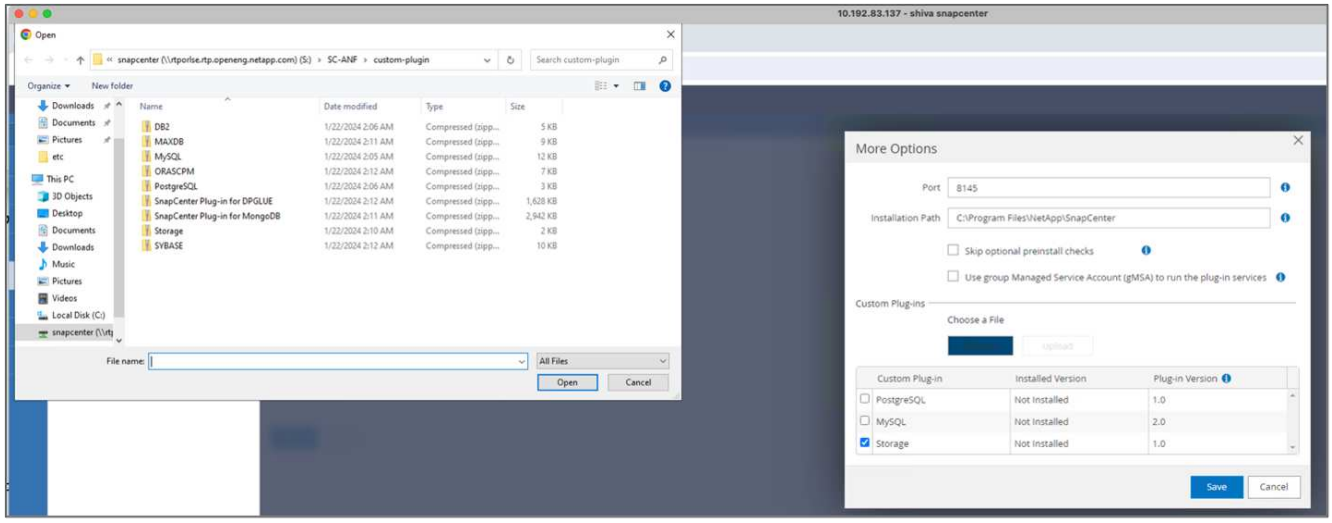
For example, in the film production industry, customers often possess critical embedded data such as video and audio files. Loss of this data, due to issues like hard drive failures, can have a significant impact on their

operations, potentially jeopardizing multimillion-dollar ventures. We have encountered instances where invaluable content was lost, causing substantial disruption and financial loss. Ensuring the security and integrity of this essential data is therefore of paramount importance in this industry.
In this section, we delve into how SnapCenter safeguards the vector database data and Milvus data residing in ONTAP. For this example, we utilized a NAS bucket (milvusdbvol1) derived from an NFS ONTAP volume (vol1) for customer data, and a separate NFS volume (vectordbpv) for Milvus cluster configuration data. please check the here for the snapcenter backup workflow

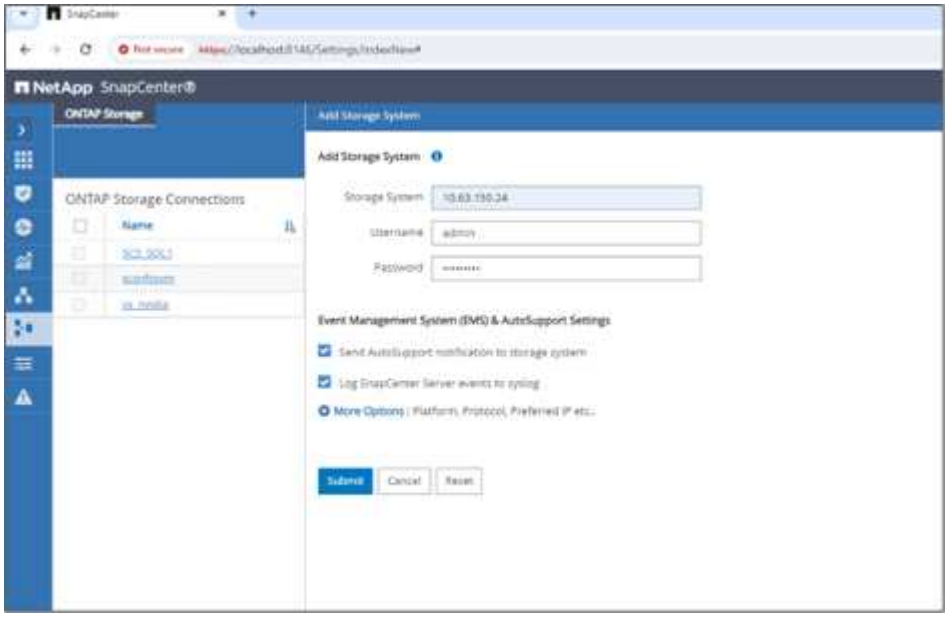
1. Set up the host that will be used to execute SnapCenter commands.



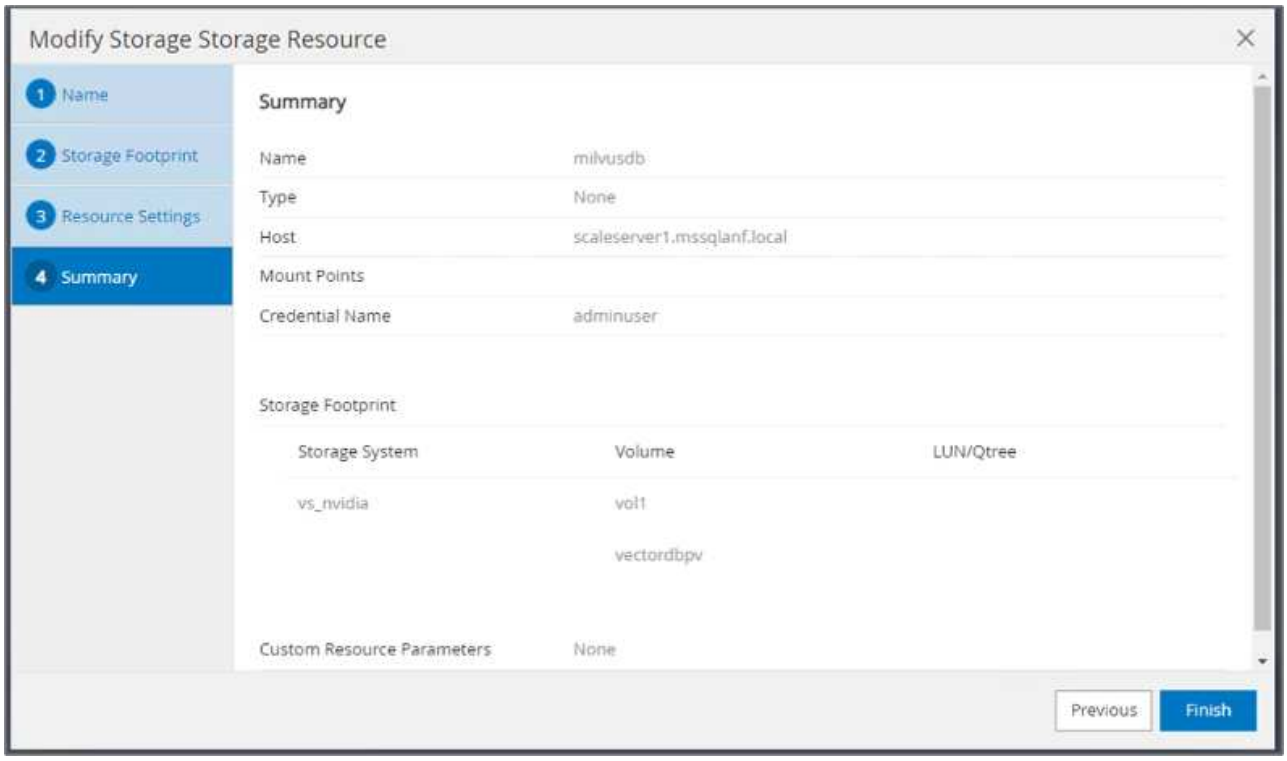
2. Install and configure the storage plugin. From the added host, select "More Options". Navigate to and select the downloaded storage plugin from the NetApp Automation Store. Install the plugin and save the configuration.



3. Set up the storage system and volume: Add the storage system under "Storage System" and select the SVM (Storage Virtual Machine). In this example, we've chosen "vs_nvidia".



4. Establish a resource for the vector database, incorporating a backup policy and a custom snapshot name.
   - Enable Consistency Group Backup with default values and enable SnapCenter without filesystem consistency.
   - In the Storage Footprint section, select the volumes associated with the vector database customer data and Milvus cluster data. In our example, these are "vol1" and "vectordbpv".
   - Create policy for vector database protection and protect vector database resource using the policy.



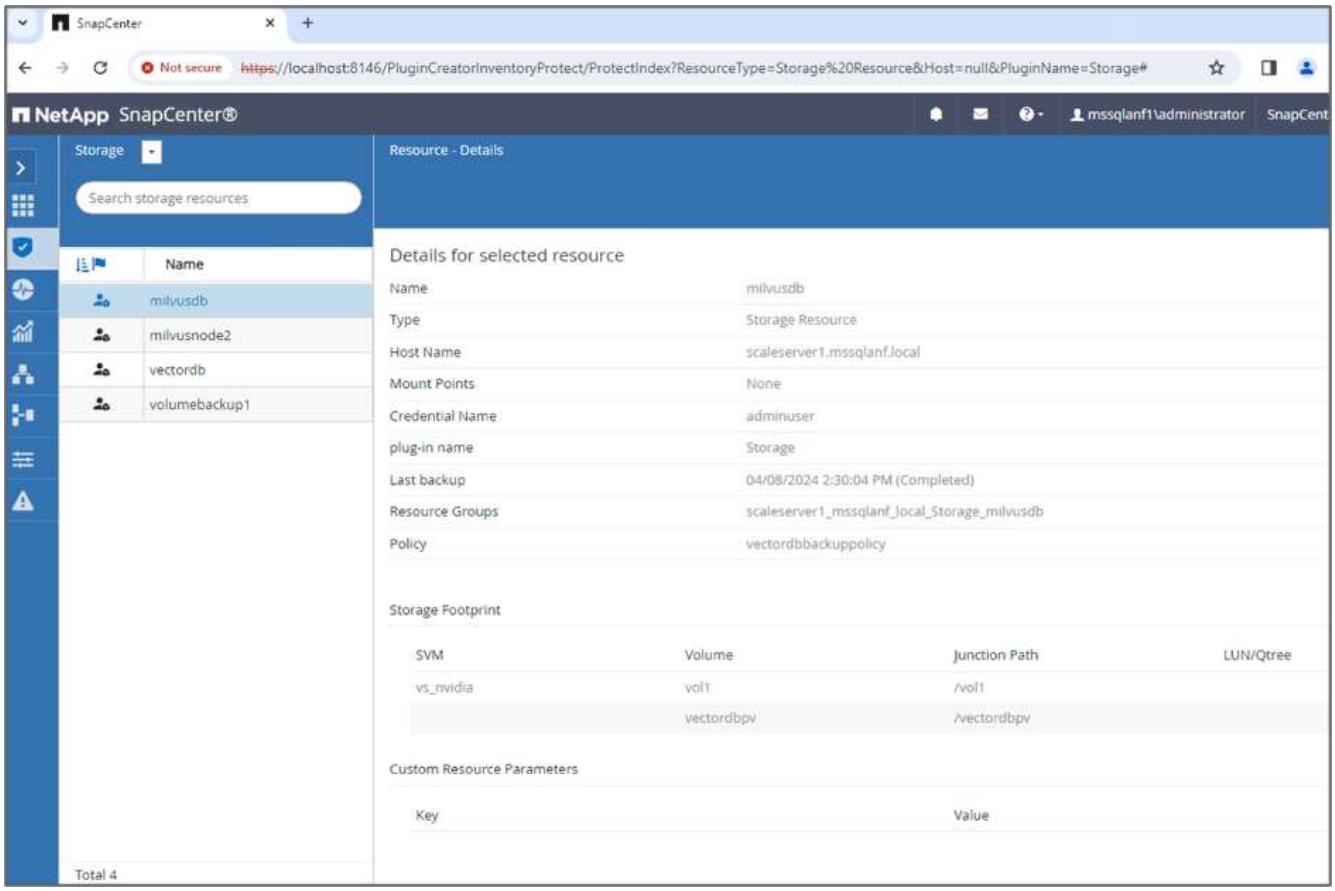
5. Insert data into the S3 NAS bucket using a Python script. In our case, we modified the backup script provided by Milvus, namely 'prepare_data_netapp.py', and executed the 'sync' command to flush the data from the operating system.

```
root@node2:~# python3 prepare_data_netapp.py

=== start connecting to Milvus     ===


=== Milvus host: localhost         ===

Does collection hello_milvus_netapp_sc_test exist in Milvus: False

=== Create collection `hello_milvus_netapp_sc_test` ===


=== Start inserting entities       ===

Number of entities in hello_milvus_netapp_sc_test: 3000

=== Create collection `hello_milvus_netapp_sc_test2` ===

Number of entities in hello_milvus_netapp_sc_test2: 6000
root@node2:~# for i in 2 3 4 5 6   ; do ssh node$i "hostname; sync; echo
'sync executed';" ; done
node2
sync executed
node3
sync executed
node4
sync executed
node5
sync executed
node6
sync executed
root@node2:~#
```

6. Verify the data in the S3 NAS bucket. In our example, the files with the timestamp '2024-04-08 21:22' were created by the 'prepare_data_netapp.py' script.

```
root@node2:~# aws s3 ls --profile ontaps3  s3://milvusdbvol1/
--recursive | grep '2024-04-08'

<output content removed to save page space>
2024-04-08 21:18:14       5656
stats_log/448950615991000809/448950615991000810/448950615991001854/100/1
2024-04-08 21:18:12       5654
stats_log/448950615991000809/448950615991000810/448950615991001854/100/4
48950615990800869
2024-04-08 21:18:17       5656
stats_log/448950615991000809/448950615991000810/448950615991001872/100/1
2024-04-08 21:18:15       5654
stats_log/448950615991000809/448950615991000810/448950615991001872/100/4
48950615990800876
2024-04-08 21:22:46       5625
stats_log/448950615991003377/448950615991003378/448950615991003385/100/1
2024-04-08 21:22:45       5623
stats_log/448950615991003377/448950615991003378/448950615991003385/100/4
48950615990800899
2024-04-08 21:22:49       5656
stats_log/448950615991003408/448950615991003409/448950615991003416/100/1
2024-04-08 21:22:47       5654
stats_log/448950615991003408/448950615991003409/448950615991003416/100/4
48950615990800906
2024-04-08 21:22:52       5656
stats_log/448950615991003408/448950615991003409/448950615991003434/100/1
2024-04-08 21:22:50       5654
stats_log/448950615991003408/448950615991003409/448950615991003434/100/4
48950615990800913
root@node2:~#
```

7. Initiate a backup using the Consistency Group (CG) snapshot from the 'milvusdb' resource

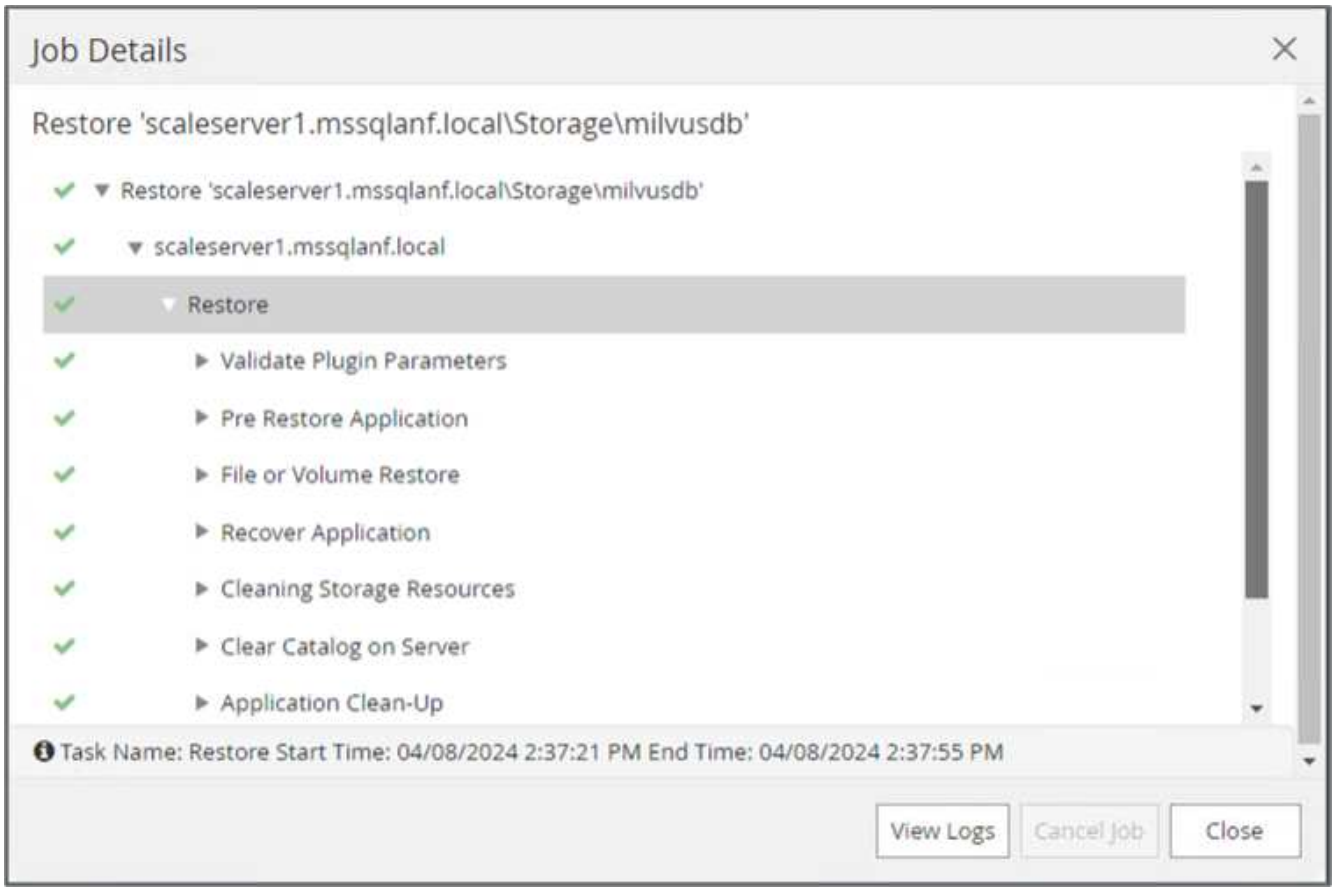


8. To test the backup functionality, we either added a new table after the backup process or removed some data from the NFS (S3 NAS bucket).

   For this test, imagine a scenario where someone created a new, unnecessary, or inappropriate collection after the backup. In such a case, we would need to revert the vector database to its state before the new collection was added. For instance, new collections such as 'hello_milvus_netapp_sc_testnew' and 'hello_milvus_netapp_sc_testnew2' have been inserted.

```
root@node2:~# python3 prepare_data_netapp.py

=== start connecting to Milvus     ===


=== Milvus host: localhost         ===

Does collection hello_milvus_netapp_sc_testnew exist in Milvus: False

=== Create collection `hello_milvus_netapp_sc_testnew` ===


=== Start inserting entities       ===

Number of entities in hello_milvus_netapp_sc_testnew: 3000

=== Create collection `hello_milvus_netapp_sc_testnew2` ===

Number of entities in hello_milvus_netapp_sc_testnew2: 6000
root@node2:~#
```

9. Execute a full restore of the S3 NAS bucket from the previous snapshot.



10. Use a Python script to verify the data from the 'hello_milvus_netapp_sc_test' and 'hello_milvus_netapp_sc_test2' collections.

```
root@node2:~# python3 verify_data_netapp.py

=== start connecting to Milvus     ===


=== Milvus host: localhost         ===

Does collection hello_milvus_netapp_sc_test exist in Milvus: True
{'auto_id': False, 'description': 'hello_milvus_netapp_sc_test',
'fields': [{'name': 'pk', 'description': '', 'type': <DataType.INT64: 5
>, 'is_primary': True, 'auto_id': False}, {'name': 'random',
'description': '', 'type': <DataType.DOUBLE: 11>}, {'name': 'var',
'description': '', 'type': <DataType.VARCHAR: 21>, 'params':
{'max_length': 65535}}, {'name': 'embeddings', 'description': '',
'type': <DataType.FLOAT_VECTOR: 101>, 'params': {'dim': 8}}]}
Number of entities in Milvus: hello_milvus_netapp_sc_test : 3000

=== Start Creating index IVF_FLAT  ===


=== Start loading                  ===


=== Start searching based on vector similarity ===

hit: id: 2998, distance: 0.0, entity: {'random': 0.9728033590489911},
random field: 0.9728033590489911
hit: id: 1262, distance: 0.08883658051490784, entity: {'random':
0.2978858685751561}, random field: 0.2978858685751561
hit: id: 1265, distance: 0.09590047597885132, entity: {'random':
0.3042039939240304}, random field: 0.3042039939240304
hit: id: 2999, distance: 0.0, entity: {'random': 0.02316334456872482},
random field: 0.02316334456872482
hit: id: 1580, distance: 0.05628091096878052, entity: {'random':
0.3855988746044062}, random field: 0.3855988746044062
hit: id: 2377, distance: 0.08096685260534286, entity: {'random':
0.8745922204004368}, random field: 0.8745922204004368
search latency = 0.2832s

=== Start querying with `random > 0.5` ===

query result:
-{'random': 0.6378742006852851, 'embeddings': [0.20963514, 0.39746657,
```

```
0.12019053, 0.6947492, 0.9535575, 0.5454552, 0.82360446, 0.21096309],
'pk': 0}
search latency = 0.2257s

=== Start hybrid searching with `random > 0.5` ===

hit: id: 2998, distance: 0.0, entity: {'random': 0.9728033590489911},
random field: 0.9728033590489911
hit: id: 747, distance: 0.14606499671936035, entity: {'random':
0.5648774800635661}, random field: 0.5648774800635661
hit: id: 2527, distance: 0.1530652642250061, entity: {'random':
0.8928974315571507}, random field: 0.8928974315571507
hit: id: 2377, distance: 0.08096685260534286, entity: {'random':
0.8745922204004368}, random field: 0.8745922204004368
hit: id: 2034, distance: 0.20354536175727844, entity: {'random':
0.5526117606328499}, random field: 0.5526117606328499
hit: id: 958, distance: 0.21908017992973328, entity: {'random':
0.6647383716417955}, random field: 0.6647383716417955
search latency = 0.5480s
Does collection hello_milvus_netapp_sc_test2 exist in Milvus: True
{'auto_id': True, 'description': 'hello_milvus_netapp_sc_test2',
'fields': [{'name': 'pk', 'description': '', 'type': <DataType.INT64: 5
>, 'is_primary': True, 'auto_id': True}, {'name': 'random',
'description': '', 'type': <DataType.DOUBLE: 11>}, {'name': 'var',
'description': '', 'type': <DataType.VARCHAR: 21>, 'params':
{'max_length': 65535}}, {'name': 'embeddings', 'description': '',
'type': <DataType.FLOAT_VECTOR: 101>, 'params': {'dim': 8}}]}
Number of entities in Milvus: hello_milvus_netapp_sc_test2 : 6000

=== Start Creating index IVF_FLAT  ===


=== Start loading                  ===


=== Start searching based on vector similarity ===

hit: id: 448950615990642008, distance: 0.07805602252483368, entity:
{'random': 0.5326684390871348}, random field: 0.5326684390871348
hit: id: 448950615990645009, distance: 0.07805602252483368, entity:
{'random': 0.5326684390871348}, random field: 0.5326684390871348
hit: id: 448950615990640618, distance: 0.13562293350696564, entity:
{'random': 0.7864676926688837}, random field: 0.7864676926688837
hit: id: 448950615990642314, distance: 0.10414951294660568, entity:
{'random': 0.2209597460821181}, random field: 0.2209597460821181
hit: id: 448950615990645315, distance: 0.10414951294660568, entity:
```

```
{'random': 0.2209597460821181}, random field: 0.2209597460821181
hit: id: 448950615990640004, distance: 0.11571306735277176, entity:
{'random': 0.7765521996186631}, random field: 0.7765521996186631
search latency = 0.2381s

=== Start querying with `random > 0.5` ===

query result:
-{'embeddings': [0.15983285, 0.72214717, 0.7414838, 0.44471496,
0.50356466, 0.8750043, 0.316556, 0.7871702], 'pk': 448950615990639798,
'random': 0.7820620141382767}
search latency = 0.3106s

=== Start hybrid searching with `random > 0.5` ===

hit: id: 448950615990642008, distance: 0.07805602252483368, entity:
{'random': 0.5326684390871348}, random field: 0.5326684390871348
hit: id: 448950615990645009, distance: 0.07805602252483368, entity:
{'random': 0.5326684390871348}, random field: 0.5326684390871348
hit: id: 448950615990640618, distance: 0.13562293350696564, entity:
{'random': 0.7864676926688837}, random field: 0.7864676926688837
hit: id: 448950615990640004, distance: 0.11571306735277176, entity:
{'random': 0.7765521996186631}, random field: 0.7765521996186631
hit: id: 448950615990643005, distance: 0.11571306735277176, entity:
{'random': 0.7765521996186631}, random field: 0.7765521996186631
hit: id: 448950615990640402, distance: 0.13665105402469635, entity:
{'random': 0.9742541034109935}, random field: 0.9742541034109935
search latency = 0.4906s
root@node2:~#
```

11. Verify that the unnecessary or inappropriate collection is no longer present in the database.

```
root@node2:~# python3 verify_data_netapp.py

=== start connecting to Milvus     ===


=== Milvus host: localhost         ===

Does collection hello_milvus_netapp_sc_testnew exist in Milvus: False
Traceback (most recent call last):
  File "/root/verify_data_netapp.py", line 37, in <module>
    recover_collection = Collection(recover_collection_name)
  File "/usr/local/lib/python3.10/dist-
packages/pymilvus/orm/collection.py", line 137, in __init__
    raise SchemaNotReadyException(
pymilvus.exceptions.SchemaNotReadyException: <SchemaNotReadyException:
(code=1, message=Collection 'hello_milvus_netapp_sc_testnew' not exist,
or you can pass in schema to create one.)>
root@node2:~#
```

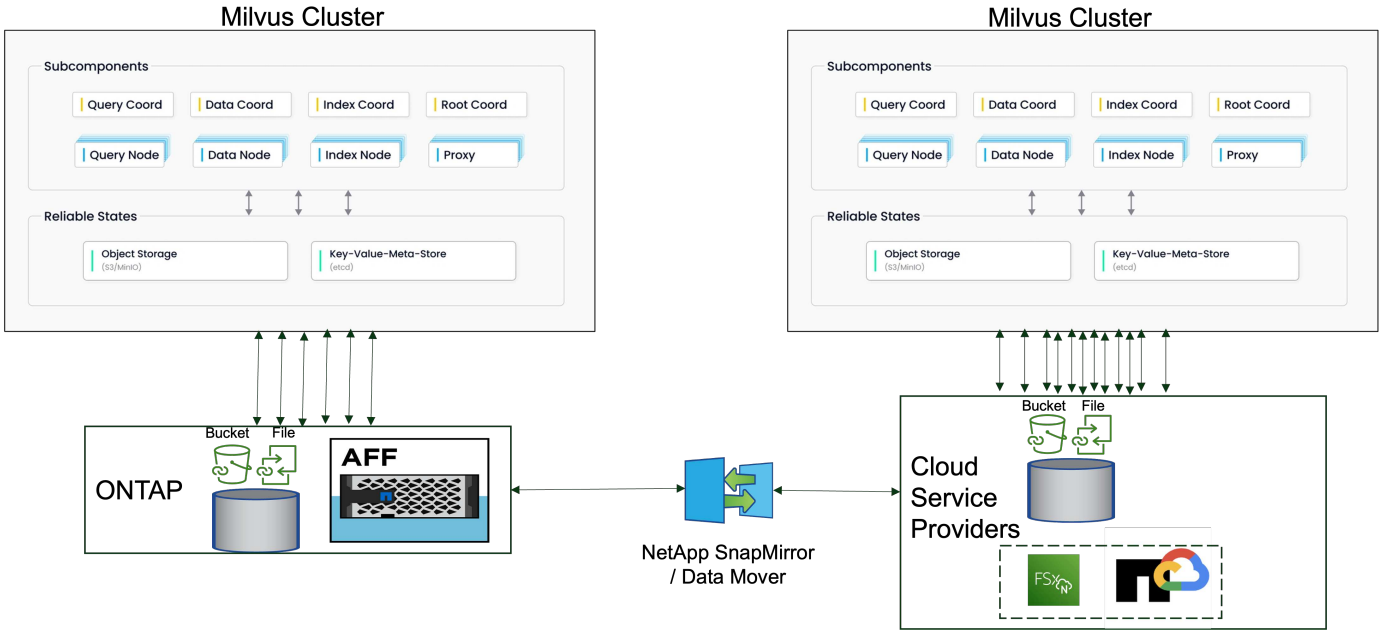
In conclusion, the use of NetApp's SnapCenter to safeguard vector database data and Milvus data residing in ONTAP offers significant benefits to customers, particularly in industries where data integrity is paramount, such as film production. SnapCenter's ability to create consistent backups and perform full data restores ensures that critical data, such as embedded video and audio files, are protected against loss due to hard drive failures or other issues. This not only prevents operational disruption but also safeguards against substantial financial loss.

In this section, we demonstrated how SnapCenter can be configured to protect data residing in ONTAP, including the setup of hosts, installation and configuration of storage plugins, and the creation of a resource for the vector database with a custom snapshot name. We also showcased how to perform a backup using the Consistency Group snapshot and verify the data in the S3 NAS bucket.

Furthermore, we simulated a scenario where an unnecessary or inappropriate collection was created after the backup. In such cases, SnapCenter's ability to perform a full restore from a previous snapshot ensures that the vector database can be reverted to its state before the addition of the new collection, thus maintaining the integrity of the database. This capability to restore data to a specific point in time is invaluable for customers, providing them with the assurance that their data is not only secure but also correctly maintained. Thus, NetApp's SnapCenter product offers customers a robust and reliable solution for data protection and management.

Disaster Recovery using NetApp SnapMirror

Disaster Recovery using NetApp SnapMirror



Disaster recovery is crucial for maintaining the integrity and availability of a vector database, especially given its role in managing high-dimensional data and executing complex similarity searches. A well-planned and implemented disaster recovery strategy ensures that data is not lost or compromised in the event of unforeseen incidents, such as hardware failures, natural disasters, or cyber-attacks. This is particularly significant for applications relying on vector databases, where the loss or corruption of data could lead to significant operational disruptions and financial losses. Moreover, a robust disaster recovery plan also ensures business continuity by minimizing downtime and allowing for the quick restoration of services. This is achieved through NetApp data replication product SnapMirrror across different geographical locations, regular backups, and failover mechanisms. Therefore, disaster recovery is not just a protective measure, but a critical component of responsible and efficient vector database management.

NetApp's SnapMirror provides data replication from one NetApp ONTAP storage controller to another, primarily used for disaster recovery (DR) and hybrid solutions. In the context of a vector database, this tool facilitates the smooth transition of data between on-premises and cloud environments. This transition is achieved without necessitating any data conversions or application refactoring, thereby enhancing the efficiency and flexibility of data management across multiple platforms.

NetApp Hybrid solution in a vector database scenario can bring about more advantages:

1. Scalability: NetApp's hybrid cloud solution offers the ability to scale your resources as per your requirements. You can utilize on-premises resources for regular, predictable workloads and cloud resources such as Amazon FSxN for NetApp ONTAP and Google Cloud NetApp Volume (GCNV) for peak times or unexpected loads.
2. Cost Efficiency: NetApp's hybrid cloud model allows you to optimize your costs by using on-premises resources for regular workloads and only paying for cloud resources when you need them. This pay-as-you-go model can be quite cost-effective with a NetApp instaclustr service offering. For on-prem and major cloud service providers, instaclustr provids support and consultation.
3. Flexibility: NetApp's hybrid cloud gives you the flexibility to choose where to process your data. For example, you might choose to perform complex vector operations on-premises where you have more powerful hardware, and less intensive operations in the cloud.
4. Business Continuity: In the event of a disaster, having your data in a NetApp hybrid cloud can ensure business continuity. You can quickly switch to the cloud if your on-premises resources are affected. We can leverage NetApp SnapMirror to move the data from on-prem to cloud and vice versa.

5. Innovation: NetApp's hybrid cloud solutions can also enable faster innovation by providing access to cutting-edge cloud services and technologies. NetApp innovations in cloud such as Amazon FSxN for NetApp ONTAP, Azure NetApp Files and Google Cloud NetApp Volumes are cloud service providers innovative products and preferred NAS.

Vector Database Performance Validation

Performance validation

Performance validation plays a critical role in both vector databases and storage systems, serving as a key factor in ensuring optimal operation and efficient resource utilization. Vector databases, known for handling high-dimensional data and executing similarity searches, need to maintain high performance levels to process complex queries swiftly and accurately. Performance validation helps identify bottlenecks, fine-tune configurations, and ensure the system can handle expected loads without degradation in service. Similarly, in storage systems, performance validation is essential to ensure data is stored and retrieved efficiently, without latency issues or bottlenecks that could impact overall system performance. It also aids in making informed decisions about necessary upgrades or changes in storage infrastructure. Therefore, performance validation is a crucial aspect of system management, contributing significantly to maintaining high service quality, operational efficiency, and overall system reliability.

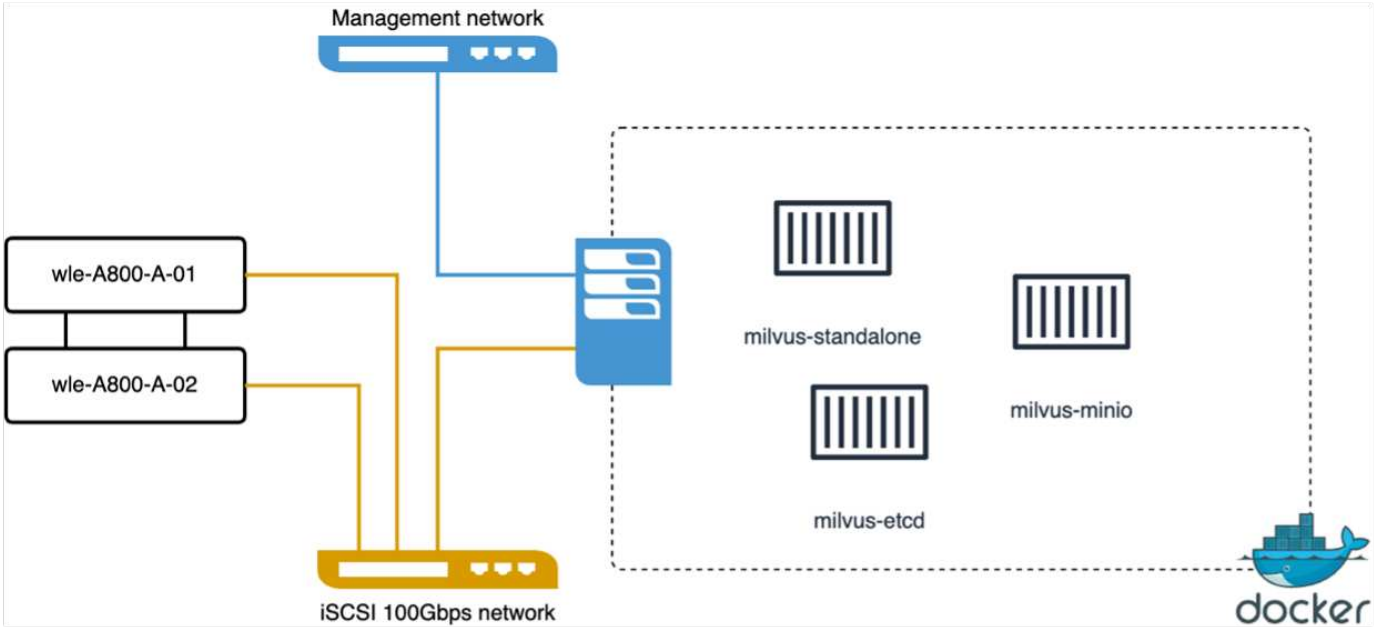
In this section, we aim to delve into the performance validation of vector databases, such as Milvus and pgvecto.rs, focusing on their storage performance characteristics such as I/O profile and netapp storage controller behavious in support of RAG and inference workloads within the LLM Lifecycle. We will evaluate and identify any performance differentiators when these databases are combined with the ONTAP storage solution. Our analysis will be based on key performance indicators, such as the number of queries processed per second(QPS).

Please check the methodology used for milvus and progress below.

Details	Milvus (Standalone and Cluster)	Postgres(pgvecto.rs)
version	2.3.2	0.2.0
Filesystem	XFS on iSCSI LUNs	
Workload Generator	VectorDB-Bench – v0.0.5	
Datasets	LAION Dataset * 10Million Embeddings * 768 Dimensions * ~300GB dataset size	

VectorDB-Bench with Milvus standalone cluster

we did the following performance validation on milvus standalone cluster with vectorDB-Bench.
The network and server connectivity of the milvus standalone cluster is below.



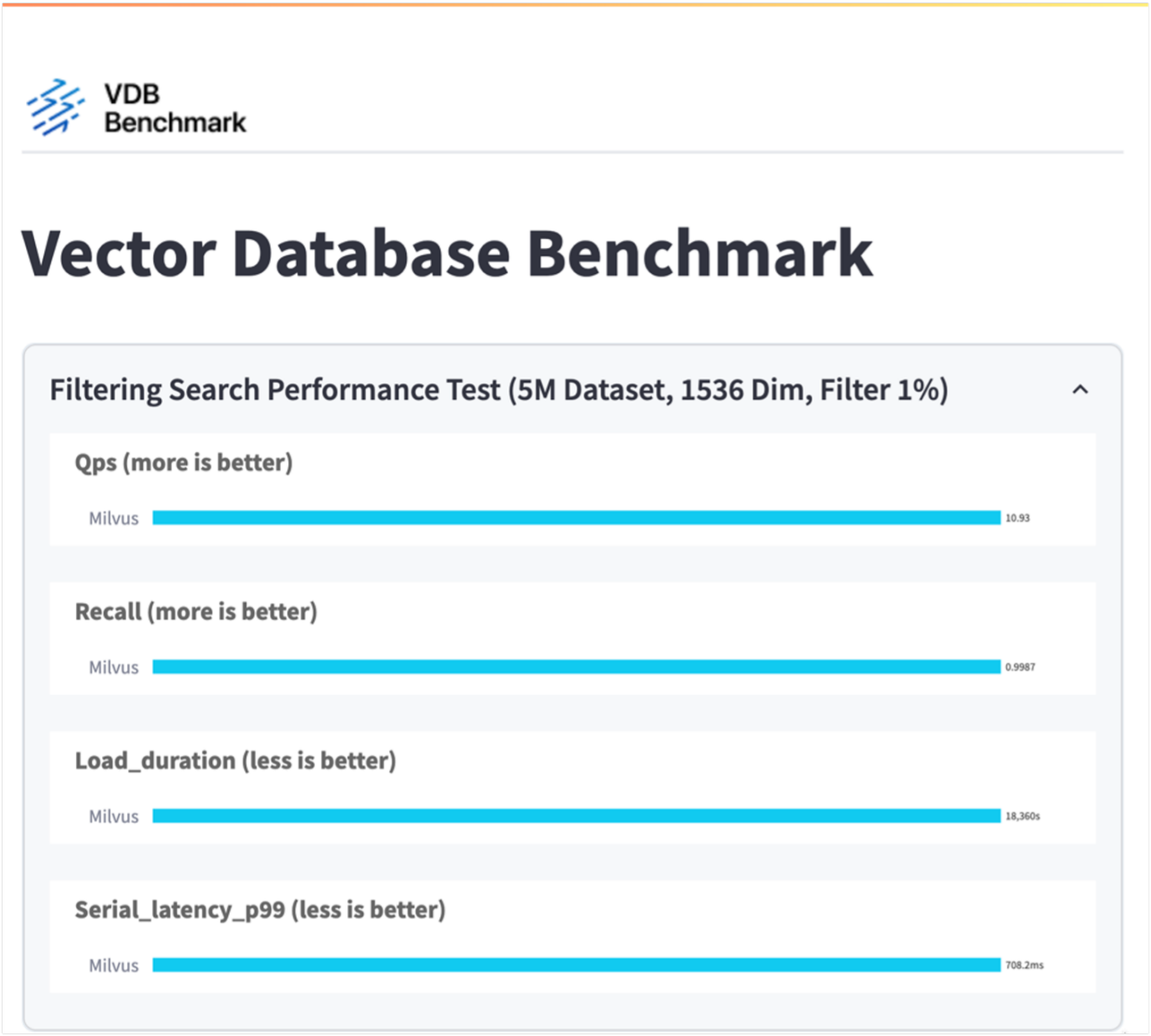
In this section, we share our observations and results from testing the Milvus standalone database.
. We selected DiskANN as the index type for these tests.
. Ingesting, optimizing, and creating indexes for a dataset of approximately 100GB took around 5 hours. For most of this duration, the Milvus server, equipped with 20 cores (which equates to 40 vcpus when Hyper-Threading is enabled), was operating at its maximum CPU capacity of 100%.We found that DiskANN is particularly important for large datasets that exceed the system memory size.
. In the query phase, we observed a Queries per Second (QPS) rate of 10.93 with a recall of 0.9987. The 99th percentile latency for queries was measured at 708.2 milliseconds.

From the storage perspective, the database issued about 1,000 ops/sec during the ingest, post-insert optimization, and index creation phases. In the query phase, it demanded 32,000 ops/sec.

The following section presents the storage performance metrics.

Workload Phase	Metric	Value
Data Ingestion and Post insert optimization	IOPS	< 1,000
	Latency	< 400 usecs
	Workload	Read/Write mix, mostly writes
	IO size	64KB
Query	IOPS	Peak at 32,000
	Latency	< 400 usecs
	Workload	100% cached read
	IO size	Mainly 8KB

The vectorDB-bench result is below.

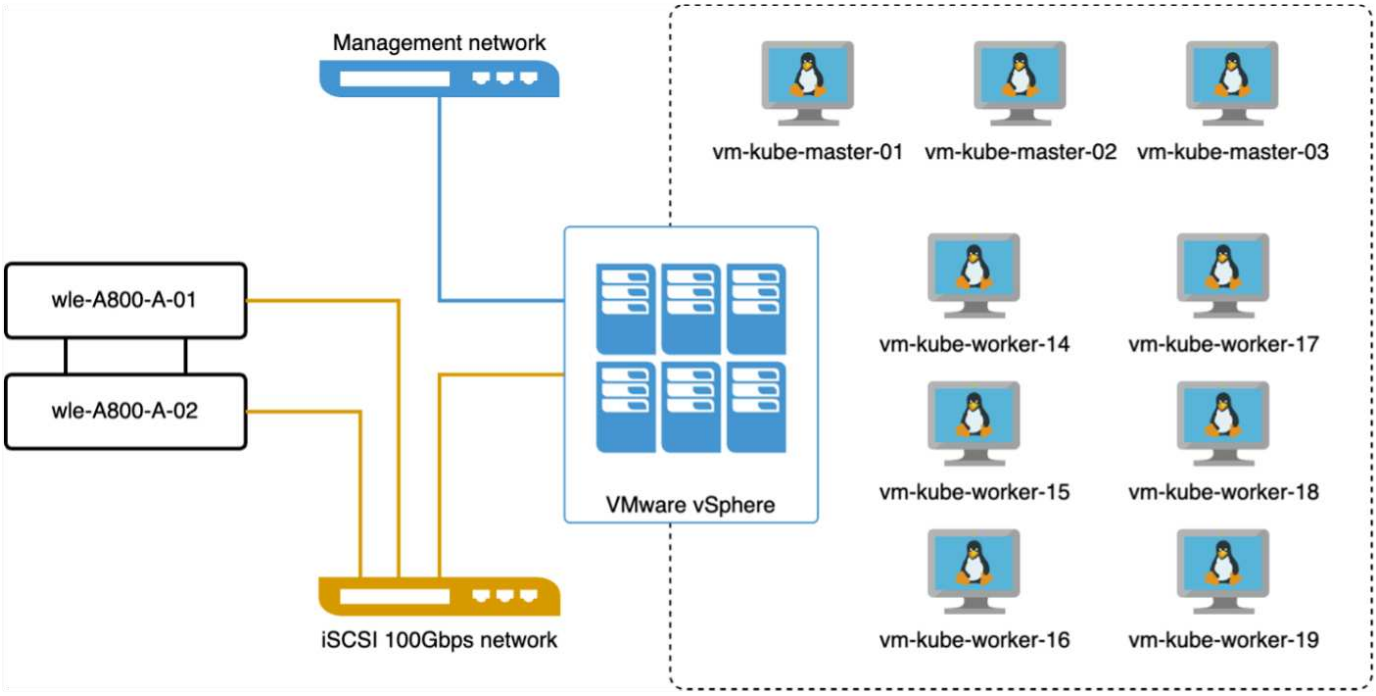


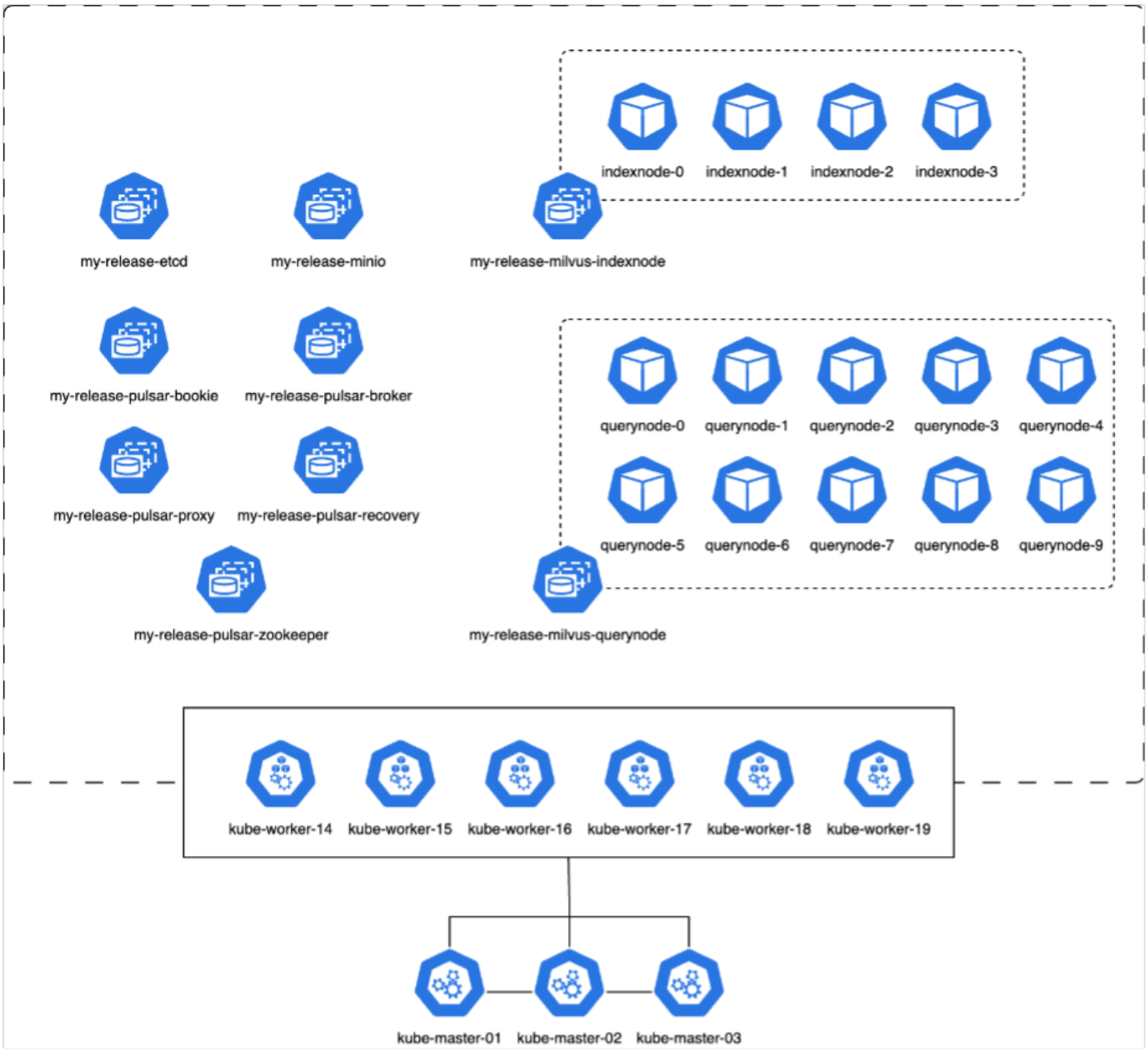
From the performance validation of the standalone Milvus instance, it's evident that the current setup is insufficient to support a dataset of 5 million vectors with a dimensionality of 1536. we've determined that the storage possesses adequate resources and does not constitute a bottleneck in the system.

VectorDB-Bench with milvus cluster

In this section, we discuss the deployment of a Milvus cluster within a Kubernetes environment. This Kubernetes setup was constructed atop a VMware vSphere deployment, which hosted the Kubernetes master and worker nodes.

The details of the VMware vSphere and Kubernetes deployments are presented in the following sections.


Management network
vm-kube-master-01
vm-kube-master-02
vm-kube-master-03
wle-A800-A-01
wle-A800-A-02
VMware vSphere
vm-kube-worker-14
vm-kube-worker-17
vm-kube-worker-15
vm-kube-worker-18
vm-kube-worker-16
vm-kube-worker-19
iSCSI 100Gbps network



In this section, we present our observations and results from testing the Milvus database.
* The index type used was DiskANN.
* The table below provides a comparison between the standalone and cluster deployments when working with 5 million vectors at a dimensionality of 1536. We observed that the time taken for data ingestion and post-insert optimization was lower in the cluster deployment. The 99th percentile latency for queries was reduced by six times in the cluster deployment compared to the standalone setup.
* Although the Queries per Second (QPS) rate was higher in the cluster deployment, it was not at the desired level.

Metric	Milvus Standalone	Milvus Cluster	Difference
QPS @ Recall	10.93 @ 0.9987	18.42 @ 0.9952	+40%
p99 Latency (less is better)	708.2 ms	117.6 ms	-83%
Load Duration time (less is better)	18,360 secs	12,730 secs	-30%

The images below provide a view of various storage metrics, including storage cluster latency and total IOPS

(Input/Output Operations Per Second).

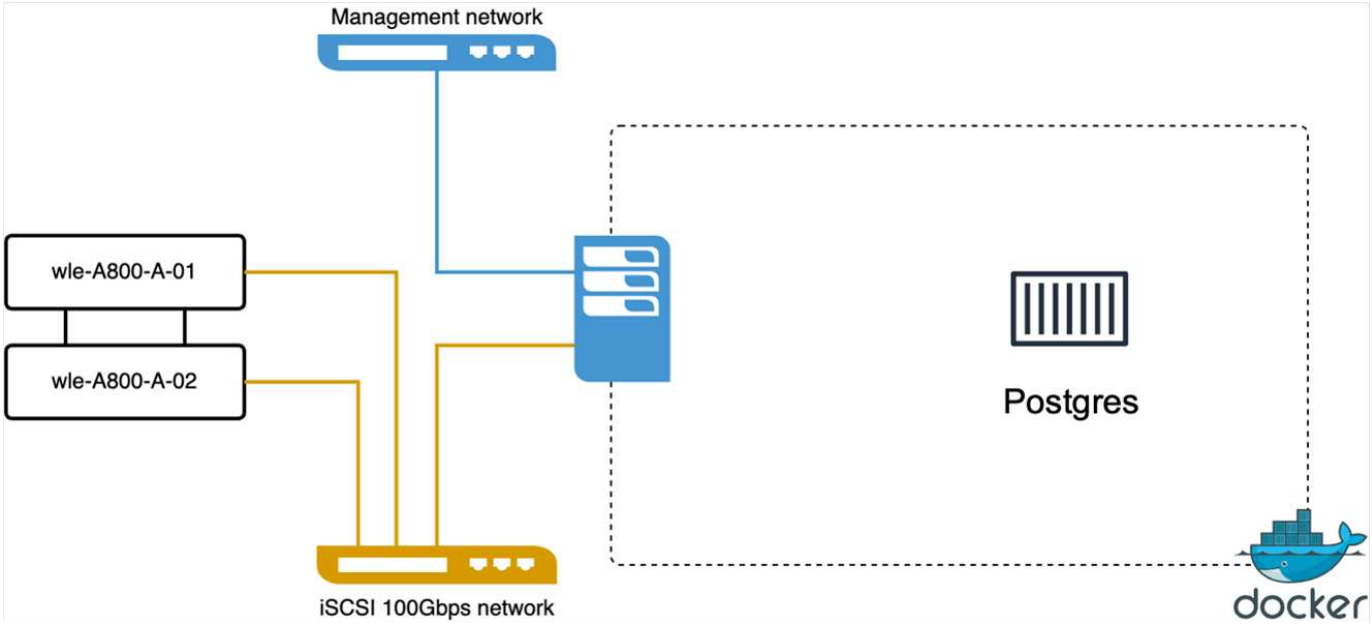
The following section presents the key storage performance metrics.

Workload Phase	Metric	Value
Data Ingestion and Post insert optimization	IOPS	< 1,000
	Latency	< 400 usecs
	Workload	Read/Write mix, mostly writes
	IO size	64KB
Query	IOPS	Peak at 147,000
	Latency	< 400 usecs
	Workload	100% cached read
	IO size	Mainly 8KB

Based on the performance validation of both the standalone Milvus and the Milvus cluster, we present the details of the storage I/O profile.
* We observed that the I/O profile remains consistent across both standalone and cluster deployments.
* The observed difference in peak IOPS can be attributed to the larger number of clients in the cluster deployment.

vectorDB-Bench with Postgres (pgvecto.rs)

We conducted the following actions on PostgreSQL(pgvecto.rs) using VectorDB-Bench:
The details regarding the network and server connectivity of PostgreSQL (specifically, pgvecto.rs) are as follows:



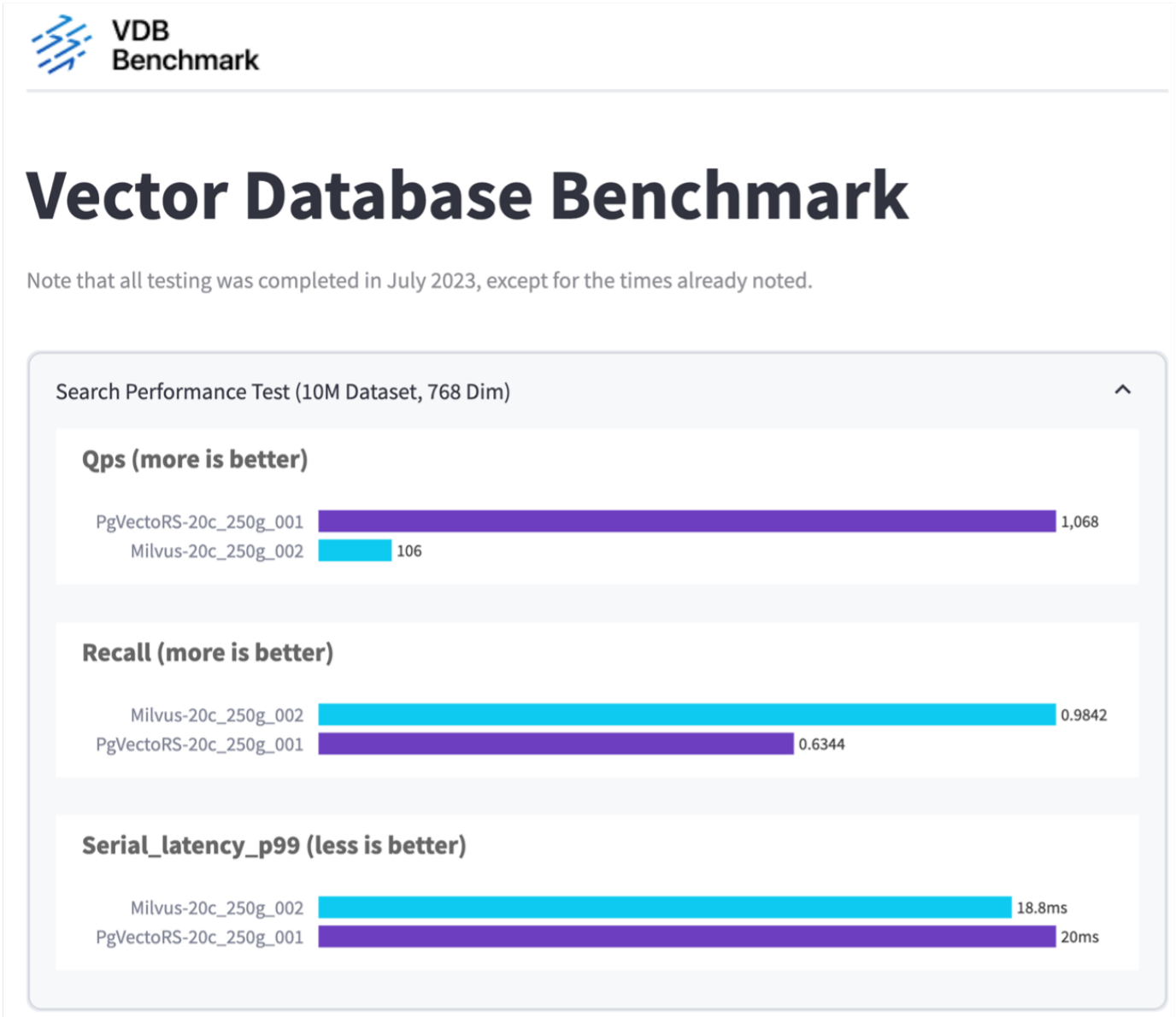
In this section, we share our observations and results from testing the PostgreSQL database, specifically using pgvecto.rs.
* We selected HNSW as the index type for these tests because at the time of testing, DiskANN wasn't available for pgvecto.rs.
* During the data ingestion phase, we loaded the Cohere dataset, which consists of 10 million vectors at a dimensionality of 768. This process took approximately 4.5 hours.
* In the query phase, we observed a Queries per Second (QPS) rate of 1,068 with a recall of 0.6344. The 99th percentile latency for queries was measured at 20 milliseconds. Throughout most of the runtime, the client CPU was operating at 100% capacity.

The images below provide a view of various storage metrics, including storage cluster latency total IOPS (Input/Output Operations Per Second).


```
The following section presents the key storage performance metrics.
```

Workload Phase	Metric	Milvus Standalone	Milvus Cluster
Data Ingestion and Post-Optimization	IOPS	< 1,000	< 1,000
	Latency	< 400 usecs	< 400 usecs
	Workload Mix	Read/Write mix, mostly writes	Read/Write mix, mostly writes
	IO Size	64 KB	64 KB
Query	IOPS	Peak at 32,000	Peak at 147,000
	Latency	< 400 usecs	< 400 usecs
	Workload Mix	100% cache reads	100% cache reads
	IO Size	Mainly 8KB	Mainly 8KB

Performance comparison between milvus and postgres on vector DB Bench



Based on our performance validation of Milvus and PostgreSQL using VectorDBBench, we observed the following:

- Index Type: HNSW
- Dataset: Cohere with 10 million vectors at 768 dimensions

We found that pgvecto.rs achieved a Queries per Second (QPS) rate of 1,068 with a recall of 0.6344, while Milvus achieved a QPS rate of 106 with a recall of 0.9842.

If high precision in your queries is a priority, Milvus outperforms pgvecto.rs as it retrieves a higher proportion of relevant items per query. However, if the number of queries per second is a more crucial factor, pgvecto.rs exceeds Milvus. It's important to note, though, that the quality of the data retrieved via pgvecto.rs is lower, with around 37% of the search results being irrelevant items.

Observation based on our performance validations:

Based on our performance validations, we have made the following observations:

In Milvus, the I/O profile closely resembles an OLTP workload, such as that seen with Oracle SLOB. The benchmark consists of three phases: Data Ingestion, Post-Optimization, and Query. The initial stages are primarily characterized by 64KB write operations, while the query phase predominantly involves 8KB reads. We expect ONTAP to handle the Milvus I/O load proficiently.

The PostgreSQL I/O profile does not present a challenging storage workload. Given the in-memory implementation currently in progress, we didn't observe any disk I/O during the query phase.

DiskANN emerges as a crucial technology for storage differentiation. It enables the efficient scaling of vector DB search beyond the system memory boundary. However, it's unlikely to establish storage performance differentiation with in-memory vector DB indices such as HNSW.

It's also worth noting that storage does not play a critical role during the query phase when the index type is HSNW, which is the most important operating phase for vector databases supporting RAG applications. The implication here is that the storage performance does not significantly impact the overall performance of these applications.

Vector Database with Instaclustr using PostgreSQL: pgvector

Vector Database with Instaclustr using PostgreSQL: pgvector

In this section, we delve into the specifics of how instaclustr product integrates with postgreSQL on pgvector fuctionality. We have an example of "How To Improve Your LLM Accuracy and Performance With PGVector and PostgreSQL®: Introduction to Embeddings and the Role of PGVector". Please check the blog to get more information.

Vector Database Use Cases

Vector Database Use Cases

In this section, we discuss about two use cases such as Retrieval Augmented Generation with Large Language Models and NetApp IT chatbot.

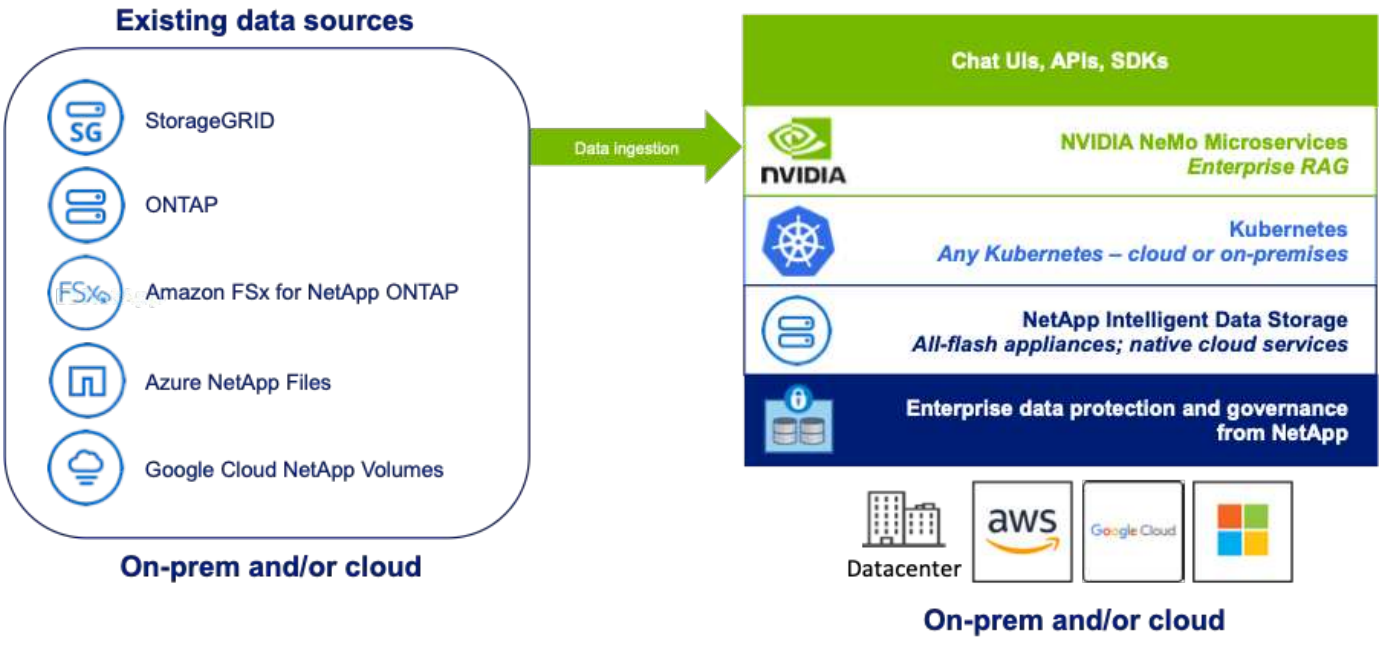
Retrieval Augmented Generation (RAG) with Large Language Models (LLMs)

Retrieval-augmented generation, or RAG, is a technique for enhancing the accuracy and reliability of Large Language Models, or LLMs, by augmenting prompts with facts fetched from external sources. In a traditional RAG deployment, vector embeddings are generated from an existing dataset and then stored in a vector database, often referred to as a knowledgebase. Whenever a user submits a prompt to the LLM, a vector embedding representation of the prompt is generated, and the vector database is searched using that embedding as the search query. This search operation returns similar vectors from the knowledgebase, which are then fed to the LLM as context alongside the original user prompt. In this way, an LLM can be augmented with additional information that was not part of its original training dataset.

The NVIDIA Enterprise RAG LLM Operator is a useful tool for implementing RAG in the enterprise. This operator can be used to deploy a full RAG pipeline. The RAG pipeline can be customized to utilize either Milvus or pgvecto as the vector database for storing knowledgebase embeddings. Refer to the documentation for details.

NetApp has validated an enterprise RAG architecture powered by the NVIDIA Enterprise RAG LLM Operator alongside NetApp storage. Refer to our blog post for more information and to see a demo. Figure 1 provides an overview of this architecture.

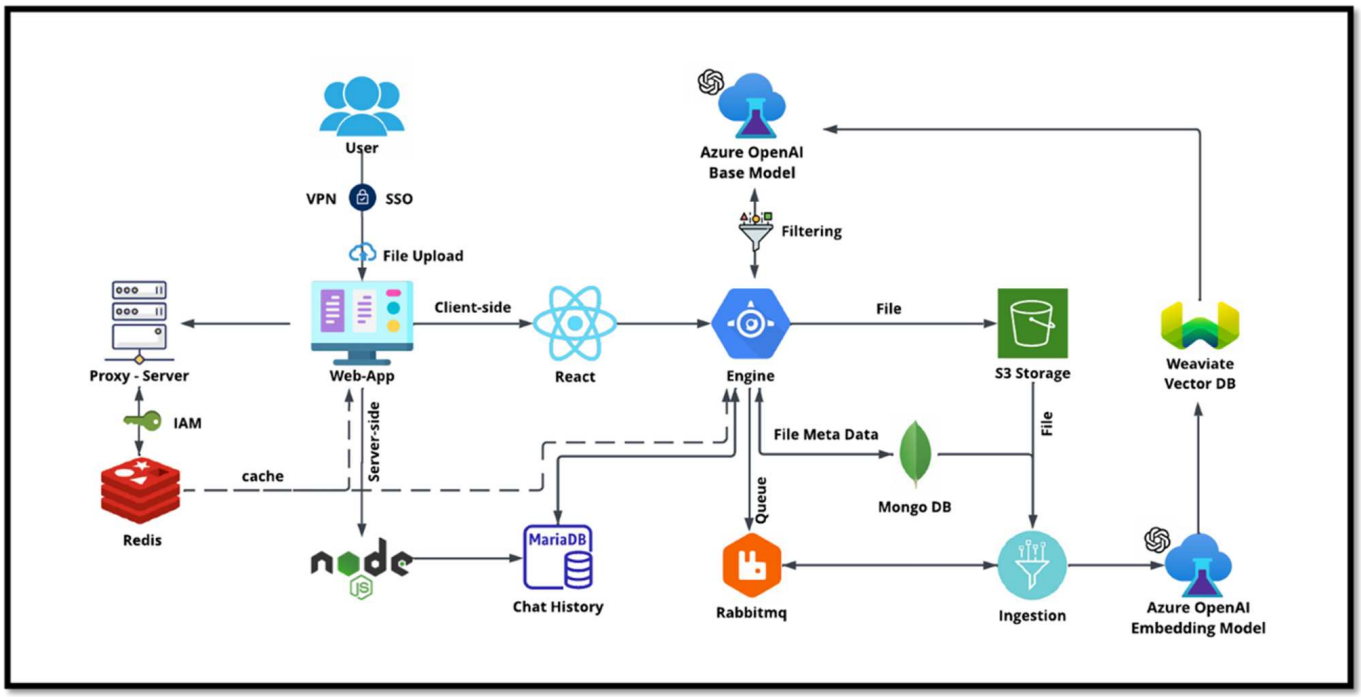
Figure 1) Enterprise RAG powered by NVIDIA NeMo Microservices and NetApp



NetApp IT chatbot use case

NetApp's chatbot serves as another real-time use case for the vector database. In this instance, the NetApp

Private OpenAI Sandbox provides an effective, secure, and efficient platform for managing queries from NetApp's internal users. By incorporating stringent security protocols, efficient data management systems, and sophisticated AI processing capabilities, it guarantees high-quality, precise responses to users based on their roles and responsibilities in the organization via SSO authentication. This architecture highlights the potential of merging advanced technologies to create user-focused, intelligent systems.



The use case can be divided into four primary sections.

User Authentication and Verification:

- User queries first go through the NetApp Single Sign-On (SSO) process to confirm the user's identity.
- After successful authentication, the system checks the VPN connection to ensure a secure data transmission.

Data Transmission and Processing:

- Once the VPN is validated, the data is sent to MariaDB through the NetAIChat or NetAICreate web applications. MariaDB is a fast and efficient database system used to manage and store user data.
- MariaDB then sends the information to the NetApp Azure instance, which connects the user data to the AI processing unit.

Interaction with OpenAI and Content Filtering:

- The Azure instance sends the user's questions to a content filtering system. This system cleans up the query and prepares it for processing.
- The cleaned-up input is then sent to the Azure OpenAI base model, which generates a response based on the input.

Response Generation and Moderation:

- The response from the base model is first checked to ensure it is accurate and meets content standards.

- After passing the check, the response is sent back to the user. This process ensures that the user receives a clear, accurate, and appropriate answer to their query.

Conclusion

Conclusion

In conclusion, this document provides a comprehensive overview of deploying and managing vector databases, such as Milvus and pgvector, on NetApp storage solutions. We discussed the infrastructure guidelines for leveraging NetApp ONTAP and StorageGRID object storage and validated the Milvus database in AWS FSX for NetApp ONTAP through file and object store.

We explored NetApp's file-object duality, demonstrating its utility not only for data in vector databases but also for other applications. We also highlighted how SnapCenter, NetApp's enterprise management product, offers backup, restore, and clone functionalities for vector database data, ensuring data integrity and availability.

The document also delves into how NetApp's Hybrid Cloud solution offers data replication and protection across on-premises and cloud environments, providing a seamless and secure data management experience. We provided insights into the performance validation of vector databases like Milvus and pgvecto on NetApp ONTAP, offering valuable information on their efficiency and scalability.

Finally, we discussed two generative AI use cases: RAG with LLM and the NetApp's internal ChatAI. These practical examples underscore the real-world applications and benefits of the concepts and practices outlined in this document. Overall, this document serves as a comprehensive guide for anyone looking to leverage NetApp's powerful storage solutions for managing vector databases.

Acknowledgments

The author like to heartfelt thanks to the below contributors, others who provided their feedback and comments to make this paper valuable to NetApp customers and NetApp fields.

1. Sathish Thyagarajan, Technical Marketing Engineer, ONTAP AI & Analytics, NetApp
2. Mike Oglesby, Technical Marketing Engineer, NetApp
3. AJ Mahajan, Senior Director, NetApp
4. Joe Scott, Manager, Workload Performance Engineering, NetApp
5. Puneet Dhawan, Senior Director, Product Management Fsx, NetApp
6. Yuval Kalderon, Senior Product Manager, FSx Product Team, NetApp

Where to find additional information

To learn more about the information that is described in this document, review the following documents and/or websites:

- Milvus documentation - https://milvus.io/docs/overview.md
- Milvus standalone documentation - https://milvus.io/docs/v2.0.x/install_standalone-docker.md
- NetApp Product Documentation https://www.netapp.com/support-and-training/documentation/
- instaclustr - instalclustr documentation

Version history

Version	Date	Document version history
Version 1.0	April 2024	Initial release

Appendix A: Values.yaml

Appendix A: Values.yaml

```
root@node2:~# cat  values.yaml
## Enable or disable Milvus Cluster mode
cluster:
  enabled: true

image:
  all:
    repository: milvusdb/milvus
    tag: v2.3.4
    pullPolicy: IfNotPresent
    ## Optionally specify an array of imagePullSecrets.
    ## Secrets must be manually created in the namespace.
    ## ref: https://kubernetes.io/docs/tasks/configure-pod-container/pull-
image-private-registry/
    ##
    # pullSecrets:
    #   - myRegistryKeySecretName
  tools:
    repository: milvusdb/milvus-config-tool
    tag: v0.1.2
    pullPolicy: IfNotPresent

# Global node selector
# If set, this will apply to all milvus components
# Individual components can be set to a different node selector
nodeSelector: {}

# Global tolerations
# If set, this will apply to all milvus components
# Individual components can be set to a different tolerations
tolerations: []

# Global affinity
# If set, this will apply to all milvus components
# Individual components can be set to a different affinity
affinity: {}
```

```
# Global labels and annotations
# If set, this will apply to all milvus components
labels: {}
annotations: {}

# Extra configs for milvus.yaml
# If set, this config will merge into milvus.yaml
# Please follow the config structure in the milvus.yaml
# at https://github.com/milvus-io/milvus/blob/master/configs/milvus.yaml
# Note: this config will be the top priority which will override the
config
# in the image and helm chart.
extraConfigFiles:
  user.yaml: |+
    #    For example enable rest http for milvus proxy
    #    proxy:
    #      http:
    #        enabled: true
    ##  Enable tlsMode and set the tls cert and key
    #  tls:
    #    serverPemPath: /etc/milvus/certs/tls.crt
    #    serverKeyPath: /etc/milvus/certs/tls.key
    #   common:
    #     security:
    #       tlsMode: 1

## Expose the Milvus service to be accessed from outside the cluster
(LoadBalancer service).
## or access it from within the cluster (ClusterIP service). Set the
service type and the port to serve it.
## ref: http://kubernetes.io/docs/user-guide/services/
##
service:
  type: ClusterIP
  port: 19530
  portName: milvus
  nodePort: ""
  annotations: {}
  labels: {}

  ## List of IP addresses at which the Milvus service is available
  ## Ref: https://kubernetes.io/docs/user-guide/services/#external-ips
  ##
  externalIPs: []
  #   - externalIp1
```

```
  # LoadBalancerSourcesRange is a list of allowed CIDR values, which are
combined with ServicePort to
  # set allowed inbound rules on the security group assigned to the master
load balancer
  loadBalancerSourceRanges:
  - 0.0.0.0/0
  # Optionally assign a known public LB IP
  # loadBalancerIP: 1.2.3.4

ingress:
  enabled: false
  annotations:
    # Annotation example: set nginx ingress type
    # kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/backend-protocol: GRPC
    nginx.ingress.kubernetes.io/listen-ports-ssl: '[19530]'
    nginx.ingress.kubernetes.io/proxy-body-size: 4m
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
  labels: {}
  rules:
    - host: "milvus-example.local"
      path: "/"
      pathType: "Prefix"
    # - host: "milvus-example2.local"
    #   path: "/otherpath"
    #   pathType: "Prefix"
  tls: []
  #  - secretName: chart-example-tls
  #    hosts:
  #      - milvus-example.local

serviceAccount:
  create: false
  name:
  annotations:
  labels:

metrics:
  enabled: true

  serviceMonitor:
    # Set this to `true` to create ServiceMonitor for Prometheus operator
    enabled: false
    interval: "30s"
    scrapeTimeout: "10s"
```

```
    # Additional labels that can be used so ServiceMonitor will be
discovered by Prometheus
    additionalLabels: {}

livenessProbe:
  enabled: true
  initialDelaySeconds: 90
  periodSeconds: 30
  timeoutSeconds: 5
  successThreshold: 1
  failureThreshold: 5

readinessProbe:
  enabled: true
  initialDelaySeconds: 90
  periodSeconds: 10
  timeoutSeconds: 5
  successThreshold: 1
  failureThreshold: 5

log:
  level: "info"
  file:
    maxSize: 300    # MB
    maxAge: 10    # day
    maxBackups: 20
  format: "text"    # text/json

  persistence:
    mountPath: "/milvus/logs"
    ## If true, create/use a Persistent Volume Claim
    ## If false, use emptyDir
    ##
    enabled: false
    annotations:
      helm.sh/resource-policy: keep
    persistentVolumeClaim:
      existingClaim: ""
      ## Milvus Logs Persistent Volume Storage Class
      ## If defined, storageClassName: <storageClass>
      ## If set to "-", storageClassName: "", which disables dynamic
provisioning
      ## If undefined (the default) or set to null, no storageClassName
spec is
      ##   set, choosing the default provisioner.
      ## ReadWriteMany access mode required for milvus cluster.
```

```
      ##
      storageClass: default
      accessModes: ReadWriteMany
      size: 10Gi
      subPath: ""

## Heaptrack traces all memory allocations and annotates these events with
stack traces.
## See more: https://github.com/KDE/heaptrack
## Enable heaptrack in production is not recommended.
heaptrack:
  image:
    repository: milvusdb/heaptrack
    tag: v0.1.0
    pullPolicy: IfNotPresent

standalone:
  replicas: 1  # Run standalone mode with replication disabled
  resources: {}
  # Set local storage size in resources
  # limits:
  #    ephemeral-storage: 100Gi
  nodeSelector: {}
  affinity: {}
  tolerations: []
  extraEnv: []
  heaptrack:
    enabled: false
  disk:
    enabled: true
    size:
      enabled: false  # Enable local storage size limit
  profiling:
    enabled: false  # Enable live profiling

  ## Default message queue for milvus standalone
  ## Supported value: rocksmq, natsmq, pulsar and kafka
  messageQueue: rocksmq
  persistence:
    mountPath: "/var/lib/milvus"
    ## If true, alertmanager will create/use a Persistent Volume Claim
    ## If false, use emptyDir
    ##
    enabled: true
    annotations:
      helm.sh/resource-policy: keep
```

```
    persistentVolumeClaim:
      existingClaim: ""
      ## Milvus Persistent Volume Storage Class
      ## If defined, storageClassName: <storageClass>
      ## If set to "-", storageClassName: "", which disables dynamic
provisioning
      ## If undefined (the default) or set to null, no storageClassName
spec is
      ##   set, choosing the default provisioner.
      ##
      storageClass:
      accessModes: ReadWriteOnce
      size: 50Gi
      subPath: ""

proxy:
  enabled: true
  # You can set the number of replicas to -1 to remove the replicas field
in case you want to use HPA
  replicas: 1
  resources: {}
  nodeSelector: {}
  affinity: {}
  tolerations: []
  extraEnv: []
  heaptrack:
    enabled: false
  profiling:
    enabled: false  # Enable live profiling
  http:
    enabled: true  # whether to enable http rest server
    debugMode:
      enabled: false
  # Mount a TLS secret into proxy pod
  tls:
    enabled: false
## when enabling proxy.tls, all items below should be uncommented and the
key and crt values should be populated.
#    enabled: true
#    secretName: milvus-tls
## expecting base64 encoded values here: i.e. $(cat tls.crt | base64 -w 0)
and $(cat tls.key | base64 -w 0)
#    key: LS0tLS1CRUdJTiBQU--REDUCT
#    crt: LS0tLS1CRUdJTiBDR--REDUCT
#  volumes:
#  - secret:
```

```
#      secretName: milvus-tls
#    name: milvus-tls
#  volumeMounts:
#  - mountPath: /etc/milvus/certs/
#    name: milvus-tls

rootCoordinator:
  enabled: true
  # You can set the number of replicas greater than 1, only if enable
active standby
  replicas: 1  # Run Root Coordinator mode with replication disabled
  resources: {}
  nodeSelector: {}
  affinity: {}
  tolerations: []
  extraEnv: []
  heaptrack:
    enabled: false
  profiling:
    enabled: false  # Enable live profiling
  activeStandby:
    enabled: false  # Enable active-standby when you set multiple replicas
for root coordinator

  service:
    port: 53100
    annotations: {}
    labels: {}
    clusterIP: ""

queryCoordinator:
  enabled: true
  # You can set the number of replicas greater than 1, only if enable
active standby
  replicas: 1  # Run Query Coordinator mode with replication disabled
  resources: {}
  nodeSelector: {}
  affinity: {}
  tolerations: []
  extraEnv: []
  heaptrack:
    enabled: false
  profiling:
    enabled: false  # Enable live profiling
  activeStandby:
    enabled: false  # Enable active-standby when you set multiple replicas
```

```
for query coordinator

  service:
    port: 19531
    annotations: {}
    labels: {}
    clusterIP: ""

queryNode:
  enabled: true
  # You can set the number of replicas to -1 to remove the replicas field
in case you want to use HPA
  replicas: 1
  resources: {}
  # Set local storage size in resources
  # limits:
  #    ephemeral-storage: 100Gi
  nodeSelector: {}
  affinity: {}
  tolerations: []
  extraEnv: []
  heaptrack:
    enabled: false
  disk:
    enabled: true  # Enable querynode load disk index, and search on disk
index
    size:
      enabled: false  # Enable local storage size limit
  profiling:
    enabled: false  # Enable live profiling

indexCoordinator:
  enabled: true
  # You can set the number of replicas greater than 1, only if enable
active standby
  replicas: 1   # Run Index Coordinator mode with replication disabled
  resources: {}
  nodeSelector: {}
  affinity: {}
  tolerations: []
  extraEnv: []
  heaptrack:
    enabled: false
  profiling:
    enabled: false  # Enable live profiling
  activeStandby:
```

```
    enabled: false  # Enable active-standby when you set multiple replicas
for index coordinator

  service:
    port: 31000
    annotations: {}
    labels: {}
    clusterIP: ""

indexNode:
  enabled: true
  # You can set the number of replicas to -1 to remove the replicas field
in case you want to use HPA
  replicas: 1
  resources: {}
  # Set local storage size in resources
  # limits:
  #    ephemeral-storage: 100Gi
  nodeSelector: {}
  affinity: {}
  tolerations: []
  extraEnv: []
  heaptrack:
    enabled: false
  profiling:
    enabled: false  # Enable live profiling
  disk:
    enabled: true  # Enable index node build disk vector index
    size:
      enabled: false  # Enable local storage size limit

dataCoordinator:
  enabled: true
  # You can set the number of replicas greater than 1, only if enable
active standby
  replicas: 1           # Run Data Coordinator mode with replication
disabled
  resources: {}
  nodeSelector: {}
  affinity: {}
  tolerations: []
  extraEnv: []
  heaptrack:
    enabled: false
  profiling:
    enabled: false  # Enable live profiling
```

```
  activeStandby:
    enabled: false  # Enable active-standby when you set multiple replicas
for data coordinator

  service:
    port: 13333
    annotations: {}
    labels: {}
    clusterIP: ""

dataNode:
  enabled: true
  # You can set the number of replicas to -1 to remove the replicas field
in case you want to use HPA
  replicas: 1
  resources: {}
  nodeSelector: {}
  affinity: {}
  tolerations: []
  extraEnv: []
  heaptrack:
    enabled: false
  profiling:
    enabled: false  # Enable live profiling

## mixCoordinator contains all coord
## If you want to use mixcoord, enable this and disable all of other
coords
mixCoordinator:
  enabled: false
  # You can set the number of replicas greater than 1, only if enable
active standby
  replicas: 1           # Run Mixture Coordinator mode with replication
disabled
  resources: {}
  nodeSelector: {}
  affinity: {}
  tolerations: []
  extraEnv: []
  heaptrack:
    enabled: false
  profiling:
    enabled: false  # Enable live profiling
  activeStandby:
    enabled: false  # Enable active-standby when you set multiple replicas
for Mixture coordinator
```

```
  service:
    annotations: {}
    labels: {}
    clusterIP: ""

attu:
  enabled: false
  name: attu
  image:
    repository: zilliz/attu
    tag: v2.2.8
    pullPolicy: IfNotPresent
  service:
    annotations: {}
    labels: {}
    type: ClusterIP
    port: 3000
    # loadBalancerIP: ""
  resources: {}
  podLabels: {}
  ingress:
    enabled: false
    annotations: {}
    # Annotation example: set nginx ingress type
    # kubernetes.io/ingress.class: nginx
    labels: {}
    hosts:
      - milvus-attu.local
    tls: []
    #  - secretName: chart-attu-tls
    #    hosts:
    #      - milvus-attu.local


## Configuration values for the minio dependency
## ref: https://github.com/minio/charts/blob/master/README.md
##

minio:
  enabled: false
  name: minio
  mode: distributed
  image:
    tag: "RELEASE.2023-03-20T20-16-18Z"
    pullPolicy: IfNotPresent
```

```
  accessKey: minioadmin
  secretKey: minioadmin
  existingSecret: ""
  bucketName: "milvus-bucket"
  rootPath: file
  useIAM: false
  iamEndpoint: ""
  region: ""
  useVirtualHost: false
  podDisruptionBudget:
    enabled: false
  resources:
    requests:
      memory: 2Gi

  gcsgateway:
    enabled: false
    replicas: 1
    gcsKeyJson: "/etc/credentials/gcs_key.json"
    projectId: ""

  service:
    type: ClusterIP
    port: 9000

  persistence:
    enabled: true
    existingClaim: ""
    storageClass:
    accessMode: ReadWriteOnce
    size: 500Gi

  livenessProbe:
    enabled: true
    initialDelaySeconds: 5
    periodSeconds: 5
    timeoutSeconds: 5
    successThreshold: 1
    failureThreshold: 5

  readinessProbe:
    enabled: true
    initialDelaySeconds: 5
    periodSeconds: 5
    timeoutSeconds: 1
    successThreshold: 1
```

```
    failureThreshold: 5

  startupProbe:
    enabled: true
    initialDelaySeconds: 0
    periodSeconds: 10
    timeoutSeconds: 5
    successThreshold: 1
    failureThreshold: 60

## Configuration values for the etcd dependency
## ref: https://artifacthub.io/packages/helm/bitnami/etcd
##

etcd:
  enabled: true
  name: etcd
  replicaCount: 3
  pdb:
    create: false
  image:
    repository: "milvusdb/etcd"
    tag: "3.5.5-r2"
    pullPolicy: IfNotPresent

  service:
    type: ClusterIP
    port: 2379
    peerPort: 2380

  auth:
    rbac:
      enabled: false

  persistence:
    enabled: true
    storageClass: default
    accessMode: ReadWriteOnce
    size: 10Gi

  ## Change default timeout periods to mitigate zoobie probe process
  livenessProbe:
    enabled: true
    timeoutSeconds: 10

  readinessProbe:
```

```
    enabled: true
    periodSeconds: 20
    timeoutSeconds: 10

  ## Enable auto compaction
  ## compaction by every 1000 revision
  ##
  autoCompactionMode: revision
  autoCompactionRetention: "1000"

  ## Increase default quota to 4G
  ##
  extraEnvVars:
  - name: ETCD_QUOTA_BACKEND_BYTES
    value: "4294967296"
  - name: ETCD_HEARTBEAT_INTERVAL
    value: "500"
  - name: ETCD_ELECTION_TIMEOUT
    value: "2500"

## Configuration values for the pulsar dependency
## ref: https://github.com/apache/pulsar-helm-chart
##

pulsar:
  enabled: true
  name: pulsar

  fullnameOverride: ""
  persistence: true

  maxMessageSize: "5242880"  # 5 * 1024 * 1024 Bytes, Maximum size of each
message in pulsar.

  rbac:
    enabled: false
    psp: false
    limit_to_namespace: true

  affinity:
    anti_affinity: false

## enableAntiAffinity: no

  components:
    zookeeper: true
    bookkeeper: true
```

```
    # bookkeeper - autorecovery
    autorecovery: true
    broker: true
    functions: false
    proxy: true
    toolset: false
    pulsar_manager: false

  monitoring:
    prometheus: false
    grafana: false
    node_exporter: false
    alert_manager: false

  images:
    broker:
      repository: apachepulsar/pulsar
      pullPolicy: IfNotPresent
      tag: 2.8.2
    autorecovery:
      repository: apachepulsar/pulsar
      tag: 2.8.2
      pullPolicy: IfNotPresent
    zookeeper:
      repository: apachepulsar/pulsar
      pullPolicy: IfNotPresent
      tag: 2.8.2
    bookie:
      repository: apachepulsar/pulsar
      pullPolicy: IfNotPresent
      tag: 2.8.2
    proxy:
      repository: apachepulsar/pulsar
      pullPolicy: IfNotPresent
      tag: 2.8.2
    pulsar_manager:
      repository: apachepulsar/pulsar-manager
      pullPolicy: IfNotPresent
      tag: v0.1.0

  zookeeper:
    volumes:
      persistence: true
      data:
        name: data
        size: 20Gi   #SSD Required
```

```
      storageClassName: default
    resources:
      requests:
        memory: 1024Mi
        cpu: 0.3
    configData:
      PULSAR_MEM: >
        -Xms1024m
        -Xmx1024m
      PULSAR_GC: >
         -Dcom.sun.management.jmxremote
         -Djute.maxbuffer=10485760
         -XX:+ParallelRefProcEnabled
         -XX:+UnlockExperimentalVMOptions
         -XX:+DoEscapeAnalysis
         -XX:+DisableExplicitGC
         -XX:+PerfDisableSharedMem
         -Dzookeeper.forceSync=no
    pdb:
      usePolicy: false

  bookkeeper:
    replicaCount: 3
    volumes:
      persistence: true
      journal:
        name: journal
        size: 100Gi
        storageClassName: default
      ledgers:
        name: ledgers
        size: 200Gi
        storageClassName: default
    resources:
      requests:
        memory: 2048Mi
        cpu: 1
    configData:
      PULSAR_MEM: >
        -Xms4096m
        -Xmx4096m
        -XX:MaxDirectMemorySize=8192m
      PULSAR_GC: >
        -Dio.netty.leakDetectionLevel=disabled
        -Dio.netty.recycler.linkCapacity=1024
        -XX:+UseG1GC -XX:MaxGCPauseMillis=10
```

```
        -XX:+ParallelRefProcEnabled
        -XX:+UnlockExperimentalVMOptions
        -XX:+DoEscapeAnalysis
        -XX:ParallelGCThreads=32
        -XX:ConcGCThreads=32
        -XX:G1NewSizePercent=50
        -XX:+DisableExplicitGC
        -XX:-ResizePLAB
        -XX:+ExitOnOutOfMemoryError
        -XX:+PerfDisableSharedMem
        -XX:+PrintGCDetails
      nettyMaxFrameSizeBytes: "104867840"
    pdb:
      usePolicy: false

  broker:
    component: broker
    podMonitor:
      enabled: false
    replicaCount: 1
    resources:
      requests:
        memory: 4096Mi
        cpu: 1.5
    configData:
      PULSAR_MEM: >
        -Xms4096m
        -Xmx4096m
        -XX:MaxDirectMemorySize=8192m
      PULSAR_GC: >
        -Dio.netty.leakDetectionLevel=disabled
        -Dio.netty.recycler.linkCapacity=1024
        -XX:+ParallelRefProcEnabled
        -XX:+UnlockExperimentalVMOptions
        -XX:+DoEscapeAnalysis
        -XX:ParallelGCThreads=32
        -XX:ConcGCThreads=32
        -XX:G1NewSizePercent=50
        -XX:+DisableExplicitGC
        -XX:-ResizePLAB
        -XX:+ExitOnOutOfMemoryError
      maxMessageSize: "104857600"
      defaultRetentionTimeInMinutes: "10080"
      defaultRetentionSizeInMB: "-1"
      backlogQuotaDefaultLimitGB: "8"
      ttlDurationDefaultInSeconds: "259200"
```

```
      subscriptionExpirationTimeMinutes: "3"
      backlogQuotaDefaultRetentionPolicy: producer_exception
    pdb:
      usePolicy: false

  autorecovery:
    resources:
      requests:
        memory: 512Mi
        cpu: 1

  proxy:
    replicaCount: 1
    podMonitor:
      enabled: false
    resources:
      requests:
        memory: 2048Mi
        cpu: 1
    service:
      type: ClusterIP
    ports:
      pulsar: 6650
    configData:
      PULSAR_MEM: >
        -Xms2048m -Xmx2048m
      PULSAR_GC: >
        -XX:MaxDirectMemorySize=2048m
      httpNumThreads: "100"
    pdb:
      usePolicy: false

  pulsar_manager:
    service:
      type: ClusterIP

  pulsar_metadata:
    component: pulsar-init
    image:
      # the image used for running `pulsar-cluster-initialize` job
      repository: apachepulsar/pulsar
      tag: 2.8.2


## Configuration values for the kafka dependency
## ref: https://artifacthub.io/packages/helm/bitnami/kafka
```

```
##

kafka:
  enabled: false
  name: kafka
  replicaCount: 3
  image:
    repository: bitnami/kafka
    tag: 3.1.0-debian-10-r52
  ## Increase graceful termination for kafka graceful shutdown
  terminationGracePeriodSeconds: "90"
  pdb:
    create: false

  ## Enable startup probe to prevent pod restart during recovering
  startupProbe:
    enabled: true

  ## Kafka Java Heap size
  heapOpts: "-Xmx4096m -Xms4096m"
  maxMessageBytes: _10485760
  defaultReplicationFactor: 3
  offsetsTopicReplicationFactor: 3
  ## Only enable time based log retention
  logRetentionHours: 168
  logRetentionBytes: _-1
  extraEnvVars:
  - name: KAFKA_CFG_MAX_PARTITION_FETCH_BYTES
    value: "5242880"
  - name: KAFKA_CFG_MAX_REQUEST_SIZE
    value: "5242880"
  - name: KAFKA_CFG_REPLICA_FETCH_MAX_BYTES
    value: "10485760"
  - name: KAFKA_CFG_FETCH_MESSAGE_MAX_BYTES
    value: "5242880"
  - name: KAFKA_CFG_LOG_ROLL_HOURS
    value: "24"

  persistence:
    enabled: true
    storageClass:
    accessMode: ReadWriteOnce
    size: 300Gi

  metrics:
    ## Prometheus Kafka exporter: exposes complimentary metrics to JMX
```

```
exporter
    kafka:
      enabled: false
      image:
        repository: bitnami/kafka-exporter
        tag: 1.4.2-debian-10-r182

    ## Prometheus JMX exporter: exposes the majority of Kafkas metrics
    jmx:
      enabled: false
      image:
        repository: bitnami/jmx-exporter
        tag: 0.16.1-debian-10-r245

    ## To enable serviceMonitor, you must enable either kafka exporter or
jmx exporter.
    ## And you can enable them both
    serviceMonitor:
      enabled: false

  service:
    type: ClusterIP
    ports:
      client: 9092

  zookeeper:
    enabled: true
    replicaCount: 3

###################################
# External S3
# - these configs are only used when `externalS3.enabled` is true
###################################
externalS3:
  enabled: true
  host: "192.168.150.167"
  port: "80"
  accessKey: "24G4C1316APP2BIPDE5S"
  secretKey: "Zd28p43rgZaU44PX_ftT279z9nt4jBSro97j87Bx"
  useSSL: false
  bucketName: "milvusdbvol1"
  rootPath: ""
  useIAM: false
  cloudProvider: "aws"
  iamEndpoint: ""
  region: ""
```

```
  useVirtualHost: false

###################################
# GCS Gateway
# - these configs are only used when `minio.gcsgateway.enabled` is true
###################################
externalGcs:
  bucketName: ""

###################################
# External etcd
# - these configs are only used when `externalEtcd.enabled` is true
###################################
externalEtcd:
  enabled: false
  ## the endpoints of the external etcd
  ##
  endpoints:
    - localhost:2379

###################################
# External pulsar
# - these configs are only used when `externalPulsar.enabled` is true
###################################
externalPulsar:
  enabled: false
  host: localhost
  port: 6650
  maxMessageSize: "5242880"  # 5 * 1024 * 1024 Bytes, Maximum size of each
message in pulsar.
  tenant: public
  namespace: default
  authPlugin: ""
  authParams: ""

###################################
# External kafka
# - these configs are only used when `externalKafka.enabled` is true
###################################
externalKafka:
  enabled: false
  brokerList: localhost:9092
  securityProtocol: SASL_SSL
  sasl:
    mechanisms: PLAIN
    username: ""
```

```
    password: ""
root@node2:~#
```

Appendix B: prepare_data_netapp_new.py

Appendix B: prepare_data_netapp_new.py

```
root@node2:~# cat prepare_data_netapp_new.py
# hello_milvus.py demonstrates the basic operations of PyMilvus, a Python
SDK of Milvus.
# 1. connect to Milvus
# 2. create collection
# 3. insert data
# 4. create index
# 5. search, query, and hybrid search on entities
# 6. delete entities by PK
# 7. drop collection
import time
import os
import numpy as np
from pymilvus import (
    connections,
    utility,
    FieldSchema, CollectionSchema, DataType,
    Collection,
)

fmt = "\n=== {:30} ===\n"
search_latency_fmt = "search latency = {:.4f}s"
#num_entities, dim = 3000, 8
num_entities, dim = 3000, 16

################################################################################
#######
# 1. connect to Milvus
# Add a new connection alias `default` for Milvus server in
`localhost:19530`
# Actually the "default" alias is a buildin in PyMilvus.
# If the address of Milvus is the same as `localhost:19530`, you can omit
all
# parameters and call the method as: `connections.connect()`.
#
# Note: the `using` parameter of the following methods is default to
"default".
print(fmt.format("start connecting to Milvus"))
```

```
host = os.environ.get('MILVUS_HOST')
if host == None:
    host = "localhost"
print(fmt.format(f"Milvus host: {host}"))
#connections.connect("default", host=host, port="19530")
connections.connect("default", host=host, port="27017")

has = utility.has_collection("hello_milvus_ntapnew_update2_sc")
print(f"Does collection hello_milvus_ntapnew_update2_sc exist in Milvus:
{has}")

#drop the collection
print(fmt.format(f"Drop collection - hello_milvus_ntapnew_update2_sc"))
utility.drop_collection("hello_milvus_ntapnew_update2_sc")
#drop the collection
print(fmt.format(f"Drop collection - hello_milvus_ntapnew_update2_sc2"))
utility.drop_collection("hello_milvus_ntapnew_update2_sc2")

################################################################################
#######
# 2. create collection
# We're going to create a collection with 3 fields.
# +-+------------+------------+------------------
+------------------------------+
# | | field name | field type | other attributes |       field description
|
# +-+------------+------------+------------------
+------------------------------+
# |1|    "pk"    |    Int64   |  is_primary=True |      "primary field"
|
# | |            |            |   auto_id=False  |
|
# +-+------------+------------+------------------
+------------------------------+
# |2|  "random"  |    Double  |                  |      "a double field"
|
# +-+------------+------------+------------------
+------------------------------+
# |3|"embeddings"| FloatVector|     dim=8        |  "float vector with dim
8"   |
# +-+------------+------------+------------------
+------------------------------+
fields = [
    FieldSchema(name="pk", dtype=DataType.INT64, is_primary=True, auto_id
=False),
```

```
    FieldSchema(name="random", dtype=DataType.DOUBLE),
    FieldSchema(name="var", dtype=DataType.VARCHAR, max_length=65535),
    FieldSchema(name="embeddings", dtype=DataType.FLOAT_VECTOR, dim=dim)
]

schema = CollectionSchema(fields, "hello_milvus_ntapnew_update2_sc")

print(fmt.format("Create collection `hello_milvus_ntapnew_update2_sc`"))
hello_milvus_ntapnew_update2_sc = Collection
("hello_milvus_ntapnew_update2_sc", schema, consistency_level="Strong")

################################################################################
######
# 3. insert data
# We are going to insert 3000 rows of data into
`hello_milvus_ntapnew_update2_sc`
# Data to be inserted must be organized in fields.
#
# The insert() method returns:
# - either automatically generated primary keys by Milvus if auto_id=True
in the schema;
# - or the existing primary key field from the entities if auto_id=False
in the schema.

print(fmt.format("Start inserting entities"))
rng = np.random.default_rng(seed=19530)
entities = [
    # provide the pk field because `auto_id` is set to False
    [i for i in range(num_entities)],
    rng.random(num_entities).tolist(),  # field random, only supports list
    [str(i) for i in range(num_entities)],
    rng.random((num_entities, dim)),    # field embeddings, supports
numpy.ndarray and list
]

insert_result = hello_milvus_ntapnew_update2_sc.insert(entities)
hello_milvus_ntapnew_update2_sc.flush()
print(f"Number of entities in hello_milvus_ntapnew_update2_sc:
{hello_milvus_ntapnew_update2_sc.num_entities}")  # check the num_entites

# create another collection
fields2 = [
    FieldSchema(name="pk", dtype=DataType.INT64, is_primary=True, auto_id
=True),
    FieldSchema(name="random", dtype=DataType.DOUBLE),
    FieldSchema(name="var", dtype=DataType.VARCHAR, max_length=65535),
```

```
    FieldSchema(name="embeddings", dtype=DataType.FLOAT_VECTOR, dim=dim)
]


schema2 = CollectionSchema(fields2, "hello_milvus_ntapnew_update2_sc2")

print(fmt.format("Create collection `hello_milvus_ntapnew_update2_sc2`"))
hello_milvus_ntapnew_update2_sc2 = Collection
("hello_milvus_ntapnew_update2_sc2", schema2, consistency_level="Strong")

entities2 = [
    rng.random(num_entities).tolist(),  # field random, only supports list
    [str(i) for i in range(num_entities)],
    rng.random((num_entities, dim)),    # field embeddings, supports
numpy.ndarray and list
]

insert_result2 = hello_milvus_ntapnew_update2_sc2.insert(entities2)
hello_milvus_ntapnew_update2_sc2.flush()
insert_result2 = hello_milvus_ntapnew_update2_sc2.insert(entities2)
hello_milvus_ntapnew_update2_sc2.flush()

# index_params = {"index_type": "IVF_FLAT", "params": {"nlist": 128},
"metric_type": "L2"}
# hello_milvus_ntapnew_update2_sc.create_index("embeddings", index_params)
#
hello_milvus_ntapnew_update2_sc2.create_index(field_name="var",index_name=
"scalar_index")

# index_params2 = {"index_type": "Trie"}
# hello_milvus_ntapnew_update2_sc2.create_index("var", index_params2)

print(f"Number of entities in hello_milvus_ntapnew_update2_sc2:
{hello_milvus_ntapnew_update2_sc2.num_entities}")  # check the num_entites

root@node2:~#
```

Appendix C: verify_data_netapp.py

Appendix C: verify_data_netapp.py

```
root@node2:~# cat verify_data_netapp.py
import time
import os
import numpy as np
```

```
from pymilvus import (
    connections,
    utility,
    FieldSchema, CollectionSchema, DataType,
    Collection,
)

fmt = "\n=== {:30} ===\n"
search_latency_fmt = "search latency = {:.4f}s"
num_entities, dim = 3000, 16
rng = np.random.default_rng(seed=19530)
entities = [
    # provide the pk field because `auto_id` is set to False
    [i for i in range(num_entities)],
    rng.random(num_entities).tolist(),  # field random, only supports list
    rng.random((num_entities, dim)),    # field embeddings, supports
numpy.ndarray and list
]

###############################################################################
######
# 1. get recovered collection hello_milvus_ntapnew_update2_sc
print(fmt.format("start connecting to Milvus"))
host = os.environ.get('MILVUS_HOST')
if host == None:
    host = "localhost"
print(fmt.format(f"Milvus host: {host}"))
#connections.connect("default", host=host, port="19530")
connections.connect("default", host=host, port="27017")

recover_collections = ["hello_milvus_ntapnew_update2_sc",
"hello_milvus_ntapnew_update2_sc2"]

for recover_collection_name in recover_collections:
    has = utility.has_collection(recover_collection_name)
    print(f"Does collection {recover_collection_name} exist in Milvus:
{has}")
    recover_collection = Collection(recover_collection_name)
    print(recover_collection.schema)
    recover_collection.flush()

    print(f"Number of entities in Milvus: {recover_collection_name} :
{recover_collection.num_entities}")  # check the num_entites


###############################################################################
```

```
######
    # 4. create index
    # We are going to create an IVF_FLAT index for
hello_milvus_ntapnew_update2_sc collection.
    # create_index() can only be applied to `FloatVector` and
`BinaryVector` fields.
    print(fmt.format("Start Creating index IVF_FLAT"))
    index = {
        "index_type": "IVF_FLAT",
        "metric_type": "L2",
        "params": {"nlist": 128},
    }

    recover_collection.create_index("embeddings", index)


################################################################################
######
    # 5. search, query, and hybrid search
    # After data were inserted into Milvus and indexed, you can perform:
    # - search based on vector similarity
    # - query based on scalar filtering(boolean, int, etc.)
    # - hybrid search based on vector similarity and scalar filtering.
    #

    # Before conducting a search or a query, you need to load the data in
`hello_milvus` into memory.
    print(fmt.format("Start loading"))
    recover_collection.load()

    #
--------------------------------------------------------------------------
---
    # search based on vector similarity
    print(fmt.format("Start searching based on vector similarity"))
    vectors_to_search = entities[-1][-2:]
    search_params = {
        "metric_type": "L2",
        "params": {"nprobe": 10},
    }

    start_time = time.time()
    result = recover_collection.search(vectors_to_search, "embeddings",
search_params, limit=3, output_fields=["random"])
    end_time = time.time()
```

```
    for hits in result:
        for hit in hits:
            print(f"hit: {hit}, random field: {hit.entity.get('random')}")
    print(search_latency_fmt.format(end_time - start_time))

    #
-------------------------------------------------------------------------
---
    # query based on scalar filtering(boolean, int, etc.)
    print(fmt.format("Start querying with `random > 0.5`"))

    start_time = time.time()
    result = recover_collection.query(expr="random > 0.5", output_fields=
["random", "embeddings"])
    end_time = time.time()

    print(f"query result:\n-{result[0]}")
    print(search_latency_fmt.format(end_time - start_time))

    #
-------------------------------------------------------------------------
---
    # hybrid search
    print(fmt.format("Start hybrid searching with `random > 0.5`"))

    start_time = time.time()
    result = recover_collection.search(vectors_to_search, "embeddings",
search_params, limit=3, expr="random > 0.5", output_fields=["random"])
    end_time = time.time()

    for hits in result:
        for hit in hits:
            print(f"hit: {hit}, random field: {hit.entity.get('random')}")
    print(search_latency_fmt.format(end_time - start_time))


###########################################################################
#####
    # 7. drop collection
    # Finally, drop the hello_milvus, hello_milvus_ntapnew_update2_sc
collection

    #print(fmt.format(f"Drop collection {recover_collection_name}"))
    #utility.drop_collection(recover_collection_name)

root@node2:~#
```

Appendix D: docker-compose.yml

Appendix D: docker-compose.yml

```
version: '3.5'

services:
  etcd:
    container_name: milvus-etcd
    image: quay.io/coreos/etcd:v3.5.5
    environment:
      - ETCD_AUTO_COMPACTION_MODE=revision
      - ETCD_AUTO_COMPACTION_RETENTION=1000
      - ETCD_QUOTA_BACKEND_BYTES=4294967296
      - ETCD_SNAPSHOT_COUNT=50000
    volumes:
      - /home/ubuntu/milvusvectordb/volumes/etcd:/etcd
    command: etcd -advertise-client-urls=http://127.0.0.1:2379 -listen
-client-urls http://0.0.0.0:2379 --data-dir /etcd
    healthcheck:
      test: ["CMD", "etcdctl", "endpoint", "health"]
      interval: 30s
      timeout: 20s
      retries: 3

  minio:
    container_name: milvus-minio
    image: minio/minio:RELEASE.2023-03-20T20-16-18Z
    environment:
      MINIO_ACCESS_KEY: minioadmin
      MINIO_SECRET_KEY: minioadmin
    ports:
      - "9001:9001"
      - "9000:9000"
    volumes:
      - /home/ubuntu/milvusvectordb/volumes/minio:/minio_data
    command: minio server /minio_data --console-address ":9001"
    healthcheck:
      test: ["CMD", "curl", "-f",
"http://localhost:9000/minio/health/live"]
      interval: 30s
      timeout: 20s
      retries: 3

  standalone:
    container_name: milvus-standalone
```

```
    image: milvusdb/milvus:v2.4.0-rc.1
    command: ["milvus", "run", "standalone"]
    security_opt:
    - seccomp:unconfined
    environment:
      ETCD_ENDPOINTS: etcd:2379
      MINIO_ADDRESS: minio:9000
    volumes:
      - /home/ubuntu/milvusvectordb/volumes/milvus:/var/lib/milvus
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9091/healthz"]
      interval: 30s
      start_period: 90s
      timeout: 20s
      retries: 3
    ports:
      - "19530:19530"
      - "9091:9091"
    depends_on:
      - "etcd"
      - "minio"

networks:
  default:
    name: milvus
```

Copyright information

Trademark information