NetApp的矢量数据库解决方案

NetApp artificial intelligence solutions

NetApp
August 18, 2025

目录

NetApp的矢量数据库解决方案

NetApp的矢量数据库解决方案

Karthikeyan Nagalingam 和 Rodrigo Nascimento， NetApp

本文档深入探讨了使用 NetApp 存储解决方案部署和管理矢量数据库（例如 Milvus 和开源 PostgreSQL 扩展 pgvecto）的方法。它详细介绍了使用NetApp ONTAP和StorageGRID对象存储的基础设施指南，并验证了 Milvus 数据库在 AWS FSx ONTAP中的应用。该文档阐明了 NetApp 的文件对象二元性及其对支持矢量嵌入的矢量数据库和应用程序的实用性。它强调了 NetApp 企业管理产品SnapCenter的功能，为矢量数据库提供备份和恢复功能，确保数据的完整性和可用性。该文档进一步深入探讨了 NetApp 的混合云解决方案，讨论了其在本地和云环境中的数据复制和保护中的作用。它包括对NetApp ONTAP上矢量数据库性能验证的见解，并总结了生成 AI 的两个实际用例：带有 LLM 的 RAG 和 NetApp 的内部 ChatAI。本文档是利用 NetApp 存储解决方案管理矢量数据库的综合指南。

参考架构重点关注以下内容：

简介

本节介绍NetApp的矢量数据库解决方案。

简介

向量数据库有效地解决了旨在处理大型语言模型 (LLM) 和生成人工智能 (AI) 中的语义搜索复杂性的挑战。与传统的数据管理系统不同，矢量数据库能够使用数据本身的内容而不是标签或标记来处理和搜索各种类型的数据，包括图像、视频、文本、音频和其他形式的非结构化数据。

关系数据库管理系统 (RDBMS) 的局限性是有据可查的，尤其是它们在处理人工智能应用中常见的高维数据表示和非结构化数据时遇到的困难。 RDBMS 通常需要将数据扁平化为更易于管理的结构，这一过程既耗时又容易出错，从而导致搜索延迟和效率低下。矢量数据库的出现解决了这些问题，为复杂高维数据的管理和搜索提供了更高效、更准确的解决方案，从而促进了人工智能应用的发展。

本文档为当前正在使用或计划使用向量数据库的客户提供全面的指南，详细介绍了在NetApp ONTAP、 NetApp StorageGRID、 Amazon FSx ONTAP for NetApp ONTAP和SnapCenter等平台上使用向量数据库的最佳实践。本文提供的内容涵盖了一系列主题：

- NetApp存储通过NetApp ONTAP和StorageGRID对象存储为 Milvus 等矢量数据库提供基础设施指南。
- 通过文件和对象存储验证 AWS FSx ONTAP中的 Milvus 数据库。
- 深入研究 NetApp 的文件对象二元性，展示其对矢量数据库以及其他应用程序中的数据的实用性。
- NetApp 的数据保护管理产品SnapCenter如何为矢量数据库数据提供备份和恢复功能。
- NetApp 的混合云如何在本地和云环境中提供数据复制和保护。
- 提供有关NetApp ONTAP上 Milvus 和 pgvector 等矢量数据库的性能验证的见解。
- 两个具体的用例：具有大型语言模型 (LLM) 的检索增强生成 (RAG) 和NetApp IT 团队的 ChatAI，从而提供所概述的概念和实践的实际示例。

解决方案概述

本节概述了NetApp矢量数据库解决方案。

解决方案概述

该解决方案展示了NetApp为应对矢量数据库客户面临的挑战所提供的独特优势和功能。通过利用NetApp ONTAP、 StorageGRID、NetApp 的云解决方案和SnapCenter，客户可以为其业务运营增加显著的价值。这些工具不仅解决了现有的问题，还提高了效率和生产力，从而促进了整体业务增长。

为什么选择**NetApp**?

- NetApp 的产品（例如ONTAP和StorageGRID）允许分离存储和计算，从而能够根据特定需求实现最佳资源利用率。这种灵活性使客户能够使用NetApp存储解决方案独立扩展其存储。
- 通过利用 NetApp 的存储控制器，客户可以使用 NFS 和 S3 协议高效地向其矢量数据库提供数据。这些协议方便了客户数据存储和管理矢量数据库索引，从而无需通过文件和对象方法访问多个数据副本。
- NetApp ONTAP为 AWS、Azure 和 Google Cloud 等领先的云服务提供商提供对 NAS 和对象存储的原生支持。这种广泛的兼容性确保了无缝集成，实现了客户数据移动性、全球可访问性、灾难恢复、动态可扩展性

和高性能。

- 借助 NetApp 强大的数据管理功能，客户可以放心，因为他们的数据受到良好的保护，不会受到潜在风险和威胁。 NetApp优先考虑数据安全，让客户对其宝贵信息的安全性和完整性感到放心。

矢量数据库

本节介绍NetApp AI 解决方案中向量数据库的定义和使用。

矢量数据库

矢量数据库是一种特殊类型的数据库，旨在使用机器学习模型的嵌入来处理、索引和搜索非结构化数据。它不以传统的表格格式组织数据，而是将数据排列为高维向量，也称为向量嵌入。这种独特的结构使得数据库能够更高效、更准确地处理复杂、多维的数据。

矢量数据库的关键功能之一是使用生成式人工智能进行分析。这包括相似性搜索，其中数据库识别类似于给定输入的数据点，以及异常检测，其中它可以发现与常态有显著偏差的数据点。

此外，矢量数据库非常适合处理时间数据或带时间戳的数据。这种类型的数据提供了有关“发生了什么”以及何时发生的信息，按顺序以及与给定 IT 系统中所有其他事件的关系。这种处理和分析时间数据的能力使得矢量数据库对于需要了解随时间推移的事件的应用程序特别有用。

矢量数据库对于**ML**和**AI**的优势：

- 高维搜索：向量数据库擅长管理和检索高维数据，这些数据通常在 AI 和 ML 应用程序中生成。
- 可扩展性：它们可以有效扩展以处理大量数据，支持 AI 和 ML 项目的增长和扩展。
- 灵活性：矢量数据库具有高度的灵活性，可以适应多种数据类型和结构。
- 性能：它们提供高性能数据管理和检索，这对于 AI 和 ML 操作的速度和效率至关重要。
- 可定制的索引：矢量数据库提供可定制的索引选项，从而能够根据特定需求优化数据组织和检索。

矢量数据库和用例。

本节提供各种矢量数据库及其用例详细信息。

Faiss和ScaNN

它们是向量搜索领域中的重要工具库。这些库提供的功能有助于管理和搜索矢量数据，使其成为数据管理这一专业领域的宝贵资源。

Elasticsearch

它是一种广泛使用的搜索和分析引擎，最近加入了矢量搜索功能。此新功能增强了其功能，使其能够更有效地处理和搜索矢量数据。

松果

它是一个具有一组独特功能的强大矢量数据库。它的索引功能同时支持密集和稀疏向量，从而增强了其灵活性和适应性。它的主要优势之一在于能够将传统搜索方法与基于人工智能的密集矢量搜索相结合，从而创造出一种兼具两全其美的混合搜索方法。

Pinecone 主要基于云，专为机器学习应用而设计，可与各种平台良好集成，包括 GCP、AWS、Open AI 、GPT-3、GPT-3.5、GPT-4、Catgut Plus、Elasticsearch、Haystack 等。值得注意的是，Pinecone 是一个闭源平台，可作为软件即服务 (SaaS) 产品使用。

鉴于其先进的功能，Pinecone 特别适合网络安全行业，其高维搜索和混合搜索功能可以有效地利用来检测和应对威胁。

色度

它是一个矢量数据库，具有包含四个主要功能的核心 API，其中一个功能包括内存文档矢量存储。它还利用 Face Transformers 库来矢量化文档，增强其功能和多功能性。 Chroma 的设计可在云端和本地运行，可根据用户需求提供灵活性。特别是在音频相关应用方面表现出色，使其成为基于音频的搜索引擎、音乐推荐系统和其他音频相关用例的绝佳选择。

威维特

它是一个多功能矢量数据库，允许用户使用其内置模块或自定义模块矢量化其内容，根据特定需求提供灵活性。它提供完全托管和自托管解决方案，满足各种部署偏好。

Weaviate 的主要功能之一是它能够同时存储矢量和对象，从而增强其数据处理能力。它广泛应用于一系列应用，包括 ERP 系统中的语义搜索和数据分类。在电子商务领域，它为搜索和推荐引擎提供支持。 Weaviate 还用于图像搜索、异常检测、自动数据协调和网络安全威胁分析，展示了其在多个领域的多功能性。

Redis

Redis 是一种高性能矢量数据库，以其快速的内存存储而闻名，可为读写操作提供低延迟。这使其成为需要快速数据访问的推荐系统、搜索引擎和数据分析应用程序的绝佳选择。

Redis 支持向量的各种数据结构，包括列表、集合和有序集。它还提供矢量运算，例如计算矢量之间的距离或查找交集和并集。这些功能对于相似性搜索、聚类和基于内容的推荐系统特别有用。

在可扩展性和可用性方面，Redis 擅长处理高吞吐量工作负载并提供数据复制。它还可以与其他数据类型很好地集成，包括传统的关系数据库（RDBMS）。 Redis 包含一个用于实时更新的发布/订阅（Pub/Sub）功能，这有利于管理实时向量。此外，Redis 轻量级且易于使用，使其成为管理矢量数据的用户友好型解决方案。

Milvus

它是一个多功能的矢量数据库，提供类似文档存储的 API，非常类似于 MongoDB。它因支持多种数据类型而脱颖而出，成为数据科学和机器学习领域的热门选择。

Milvus 的独特功能之一是其多矢量化功能，它允许用户在运行时指定用于搜索的矢量类型。此外，它利用 Knowwhere（一个位于 Faiss 等其他库之上的库）来管理查询和向量搜索算法之间的通信。

由于与 PyTorch 和 TensorFlow 兼容，Milvus 还提供与机器学习工作流程的无缝集成。这使其成为一系列应用的绝佳工具，包括电子商务、图像和视频分析、对象识别、图像相似性搜索和基于内容的图像检索。在自然语言处理领域，Milvus 用于文档聚类、语义搜索和问答系统。

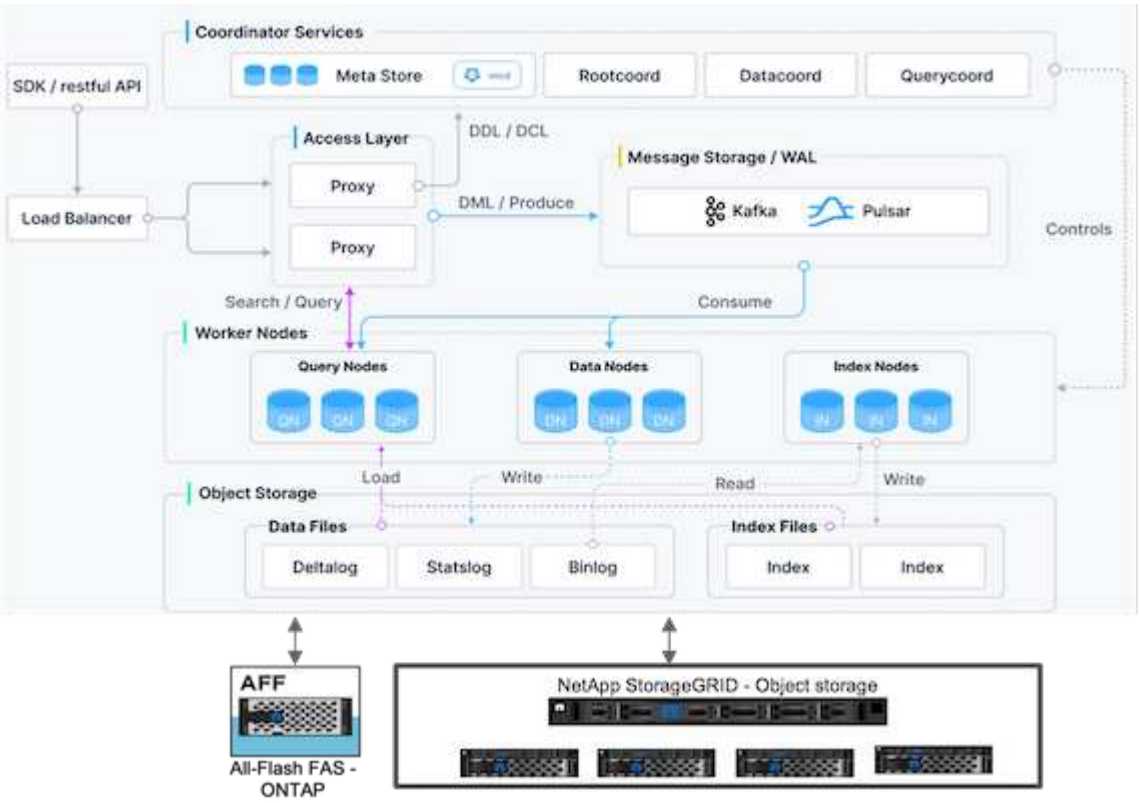
对于这个解决方案，我们选择了 milvus 进行解决方案验证。为了提高性能，我们同时使用了 milvus 和 postgres（pgvecto.rs）。

为什么我们选择 **milvus** 作为这个解决方案?

- 开源：Milvus 是一个开源矢量数据库，鼓励社区驱动的开发和改进。

- AI 集成：它利用嵌入相似性搜索和 AI 应用程序来增强矢量数据库功能。
- 大容量处理：Milvus 有能力存储、索引和管理由深度神经网络 (DNN) 和机器学习 (ML) 模型生成的超过十亿个嵌入向量。
- 用户友好：易于使用，设置只需不到一分钟。 Milvus 还为不同的编程语言提供 SDK。
- 速度：它提供极快的检索速度，比一些替代方案快 10 倍。
- 可扩展性和可用性：Milvus 具有高度可扩展性，可以根据需要进行扩展和缩小。
- 功能丰富：它支持不同的数据类型、属性过滤、用户定义函数 (UDF) 支持、可配置的一致性级别和旅行时间，使其成为各种应用的多功能工具。

Milvus 架构概述



本节提供 Milvus 架构中使用的更高级别的组件和服务。 * 访问层——由一组无状态代理组成，作为系统的前端层和用户的端点。 * 协调器服务——它将任务分配给工作节点并充当系统的大脑。它有三种协调器类型：根协调器、数据协调器和查询协调器。 * 工作节点：它遵循协调服务的指令并执行用户触发的DML / DDL命令。它有三种类型的工作节点，例如查询节点，数据节点和索引节点。 * 存储：负责数据持久化。它包括元存储、日志代理和对象存储。 NetApp存储（例如ONTAP和StorageGRID）为 Milvus 提供对象存储和基于文件的存储，用于客户数据和矢量数据库数据。

技术要求

本节概述了NetApp矢量数据库解决方案的要求。

技术要求

除性能外，下面概述的硬件和软件配置用于本文档中执行的大部分验证。这些配置可作为帮助您设置环境的指

南。但请注意，具体组件可能会因个别客户的要求而有所不同。

硬件要求

硬件	详细信息
NetApp AFF存储阵列 HA 对	* A800 * ONTAP 9.14.1 * 48 x 3.49TB SSD-NVM * 两个灵活组卷：元数据和数据。 * 元数据 NFS 卷有 12 个持久卷，每个卷为 250GB。 * 数据是ONTAP NAS S3 卷
6台富士通PRIMERGY RX2540 M4	* 64 个 CPU * Intel® Xeon® Gold 6142 CPU @ 2.60GHz * 256 GM 物理内存 * 1 x 100GbE 网络端口
网络连接	100 GbE
StorageGRID	* 1 x SG100，3xSGF6024 * 3 x 24 x 7.68TB

软件要求

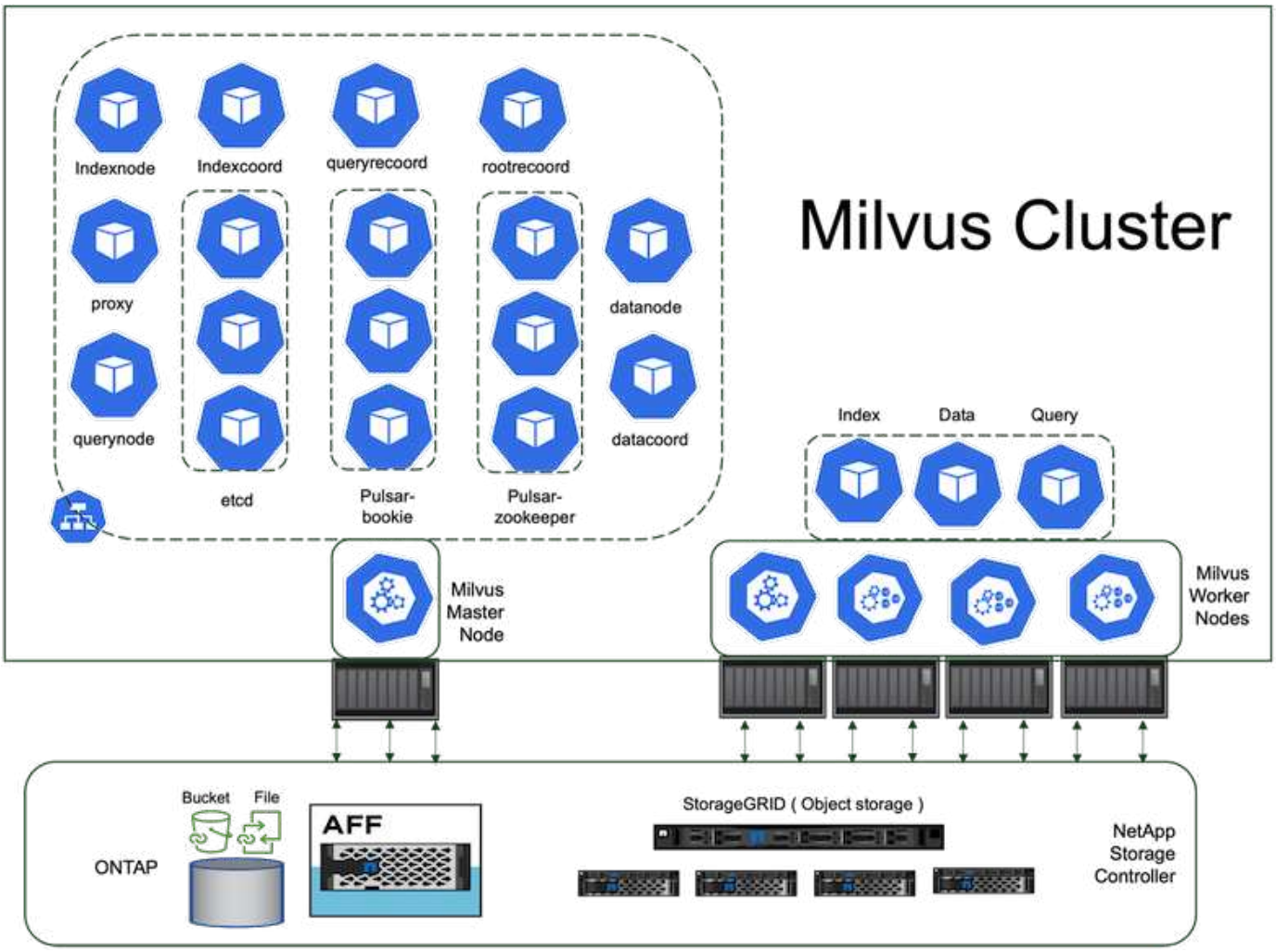
软件	详细信息
Milvus 集群	* 图表 - milvus-4.1.11。 * APP 版本 – 2.3.4 * 依赖的 bundles，例如 bookkeeper、zookeeper、pulsar 、etcd、proxy、querynode、worker
Kubernetes	* 5 节点 K8s 集群 * 1 个主节点和 4 个工作节点 * 版本 – 1.7.2
Python	*3.10.12.

部署流程

本节讨论NetApp矢量数据库解决方案的部署过程。

部署过程

在本部署部分中，我们使用 milvus 矢量数据库和 Kubernetes 进行如下实验设置。



NetApp 存储为集群提供存储，以保存客户数据和 Milvus 集群数据。

NetApp存储设置 – ONTAP

- 存储系统初始化
- 存储虚拟机 (SVM) 创建
- 逻辑网络接口的分配
- NFS、S3 配置和许可

对于 NFS（网络文件系统），请按照以下步骤操作：

1. 为 NFSv4 创建FlexGroup卷。在我们为此次验证所做的设置中，我们使用了 48 个 SSD，其中 1 个 SSD 专用于控制器的根卷，另外 47 个 SSD 分布用于 NFSv4]].验证FlexGroup卷的 NFS 导出策略是否对 Kubernetes（K8s）节点网络具有读/写权限。如果没有这些权限，请授予 K8s 节点网络的读/写 (rw) 权限。
2. 在所有 K8s 节点上，创建一个文件夹，并通过每个 K8s 节点上的逻辑接口 (LIF) 将FlexGroup卷挂载到该文件夹上。

对于 NAS S3（网络附加存储简单存储服务），请按照以下步骤操作：

1. 为 NFS 创建FlexGroup卷。

2. 使用“vserver object-store-server create”命令设置一个启用 HTTP 的对象存储服务器，并将管理状态设置为“up”。您可以选择启用 HTTPS 并设置自定义侦听器端口。
3. 使用“vserver object-store-server user create -user <username>”命令创建 object-store-server 用户。
4. 要获取访问密钥和密钥，可以运行以下命令：“set diag; vserver object-store-server user show -user <username>”。但是，今后这些密钥将在用户创建过程中提供，或者可以使用 REST API 调用来检索。
5. 使用步骤 2 中创建的用户建立对象存储服务器组并授予访问权限。在这个例子中，我们提供了“FullAccess”。
6. 通过将其类型设置为“nas”并提供 NFSv3 卷的路径来创建 NAS 存储桶。也可以利用 S3 存储桶来实现此目的。

NetApp存储设置 – StorageGRID

1. 安装 storageGRID 软件。
2. 创建租户和存储桶。
3. 创建具有所需权限的用户。

请查看更多详细信息 https://docs.netapp.com/us-en/storagegrid-116/primer/index.html

解决方案验证

解决方案概述

我们针对五个关键领域进行了全面的解决方案验证，具体细节概述如下。每个部分都深入探讨了客户面临的挑战、 NetApp提供的解决方案以及随后给客户带来的好处。

1. "在本地使用 Kubernetes 设置 Milvus 集群"客户面临的挑战是独立扩展存储和计算、有效的基础设施管理和数据管理。在本节中，我们详细介绍了在 Kubernetes 上安装 Milvus 集群的过程，并利用NetApp存储控制器存储集群数据和客户数据。
2. milvus 与 Amazon FSx ONTAP for NetApp ONTAP – 文件和对象二元性 在本节中，我们将介绍为什么ONTAP在云中部署矢量数据库，以及在NetApp Amazon FSxAmazon FSxONTAPNetAppONTAP（milvus独立版）的步骤。
3. "使用NetApp SnapCenter进行矢量数据库保护。"在本节中，我们将深入探讨SnapCenter如何保护驻留在ONTAP中的矢量数据库数据和 Milvus 数据。在此示例中，我们利用源自 NFS ONTAP卷（vol1）的 NAS 存储桶（milvusdbvol1）来存储客户数据，并使用单独的 NFS 卷（vectordbpv）来存储 Milvus 集群配置数据。
4. "使用NetApp SnapMirror进行灾难恢复"在本节中，我们讨论灾难恢复（DR）对于矢量数据库的重要性以及NetApp灾难恢复产品SnapMirror如何为矢量数据库提供DR解决方案。
5. "性能验证"在本节中，我们旨在深入研究矢量数据库（例如 Milvus 和 pgvecto.rs）的性能验证，重点关注它们的存储性能特征，例如 I/O 配置文件和 NetApp 存储控制器在 LLM 生命周期内支持 RAG 和推理工作负载的行为。当这些数据库与ONTAP存储解决方案结合时，我们将评估并识别任何性能差异因素。我们的分析将基于关键性能指标，例如每秒处理的查询数（QPS）。

在本地使用 **Kubernetes** 设置 **Milvus** 集群

本节讨论针对NetApp矢量数据库解决方案的 milvus 集群设置。

在本地使用 **Kubernetes** 设置 **Milvus** 集群

客户面临的挑战是在存储和计算上独立扩展、有效的基础设施管理和数据管理，Kubernetes 和矢量数据库共同构成了管理大数据操作的强大、可扩展的解决方案。 Kubernetes 优化资源并管理容器，而矢量数据库则高效处理高维数据和相似性搜索。这种组合能够快速处理大型数据集上的复杂查询，并随着数据量的增加而无缝扩展，使其成为大数据应用程序和人工智能工作负载的理想选择。

1. 在本节中，我们详细介绍了在 Kubernetes 上安装 Milvus 集群的过程，并利用NetApp存储控制器存储集群数据和客户数据。
2. 要安装 Milvus 集群，需要持久卷 (PV) 来存储来自各个 Milvus 集群组件的数据。这些组件包括 etcd（三个实例）、pulsar-bookie-journal（三个实例）、pulsar-bookie-ledgers（三个实例）和 pulsar-zookeeper-data（三个实例）。

> 在 milvus 集群中，我们可以使用 pulsar 或者 kafka 作为支撑 Milvus 集群可靠存储以及消息流发布/订阅的底层引擎。对于使用 NFS 的 Kafka， NetApp在ONTAP 9.12.1 及更高版本中做出了改进，这些增强功能以及 RHEL 8.7 或 9.1 及更高版本中包含的 NFSv4.1 和 Linux 更改解决了在 NFS 上运行 Kafka 时可能出现的“愚蠢重命名”问题。如果您对使用 NetApp NFS 解决方案运行 Kafka 主题的更多深入信息感兴趣，请查看 -"此链接" 。

3. 我们从NetApp ONTAP创建了一个 NFS 卷，并建立了 12 个持久卷，每个卷具有 250GB 的存储空间。存储大小可能因集群大小而异；例如，我们有另一个集群，其中每个 PV 有 50GB。请参阅下面的 PV YAML 文件之一以了解更多详细信息；我们总共有 12 个这样的文件。在每个文件中，storageClassName 设置为“default”，并且存储和路径对于每个 PV 都是唯一的。

```
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. 对每个 PV YAML 文件执行“kubectl apply”命令来创建持久卷，然后使用“kubectl get pv”验证其创建

```
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. 为了存储客户数据，Milvus 支持对象存储解决方案，例如 MinIO、Azure Blob 和 S3。在本指南中，我们使用 S3。以下步骤适用于ONTAP S3 和StorageGRID对象存储。我们使用 Helm 来部署 Milvus 集群。从 Milvus 下载位置下载配置文件 values.yaml。有关我们在本文档中使用的 values.yaml 文件，请参阅附录。
6. 确保每个部分中的“storageClass”设置为“default”，包括日志、etcd、zookeeper 和 bookkeeper。
7. 在 MinIO 部分，禁用 MinIO。
8. 从ONTAP或StorageGRID对象存储创建 NAS 存储桶，并使用对象存储凭据将其包含在外部 S3 中。

```
###################################
# 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. 在创建 Milvus 集群之前，请确保 PersistentVolumeClaim（PVC）没有任何预先存在的资源。

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

10. 利用 Helm 和 values.yaml 配置文件安装并启动 Milvus 集群。

```
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. 验证 PersistentVolumeClaims (PVC) 的状态。

```
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. 检查 pod 的状态。

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

请确保 Pod 状态为“正在运行”且按预期工作

13. 测试在 Milvus 和NetApp对象存储中写入和读取数据。
    - 使用“prepare_data_netapp_new.py”Python 程序写入数据。

```
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:~#
```

    - 使用“verify_data_netapp.py”Python 文件读取数据。

```
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}}]}
```

基于以上验证，Kubernetes 与矢量数据库的集成，通过使用NetApp存储控制器在 Kubernetes 上部署 Milvus 集群，为客户提供了强大、可扩展且高效的大规模数据操作管理解决方案。此设置为客户提供了处理高维数据和快速高效地执行复杂查询的能力，使其成为大数据应用和人工智能工作负载的理想解决方案。对各种集群组件使用持久卷 (PV)，以及从NetApp ONTAP创建单个 NFS 卷，可确保最佳资源利用率和数据管理。验证 PersistentVolumeClaims (PVC) 和 pod 的状态以及测试数据写入和读取的过程为客户提供了可靠且一致的数据操作的保证。使用ONTAP或StorageGRID对象存储客户数据进一步增强了数据的可访问性和安全性。总体而言，这种设置为客户提供了一种有弹性且高性能的数据管理解决方案，可以随着客户不断增长的数据需求而无缝扩展。

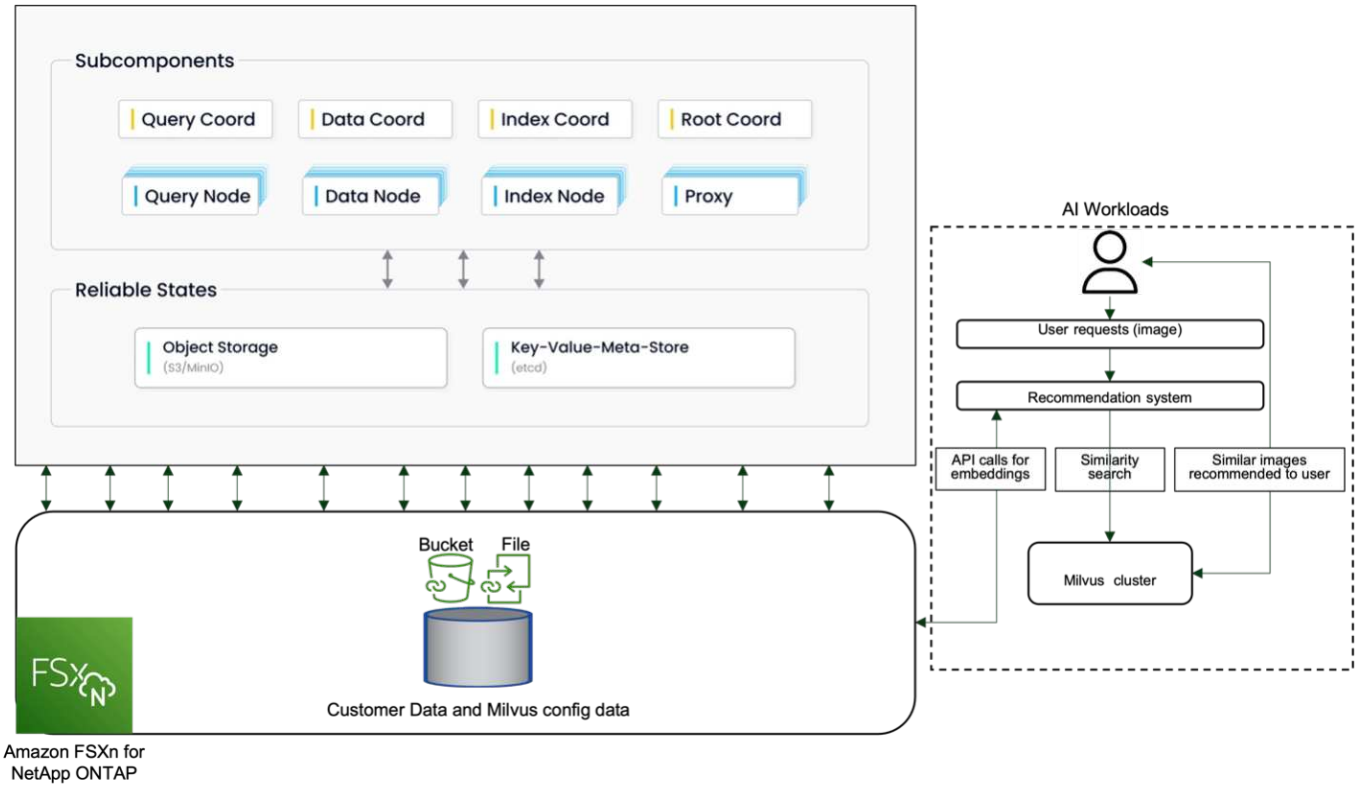
Milvus 与Amazon FSx ONTAP for NetApp ONTAP - 文件和对象二元性

本节讨论使用Amazon FSx ONTAP为NetApp提供矢量数据库解决方案的 milvus 集群设置。

Milvus 与Amazon FSx ONTAP for NetApp ONTAP – 文件和对象二元性

在本节中，我们将介绍为什么需要在云中部署矢量数据库，以及在 Docker 容器中的Amazon FSx ONTAP for NetApp ONTAP中部署矢量数据库（milvus 独立版）的步骤。

在云中部署矢量数据库有几个显著的好处，特别是对于需要处理高维数据和执行相似性搜索的应用程序。首先，基于云的部署提供了可扩展性，允许轻松调整资源以适应不断增长的数据量和查询负载。这确保数据库能够有效地处理增加的需求，同时保持高性能。其次，云部署提供了高可用性和灾难恢复，因为数据可以在不同的地理位置复制，最大限度地降低数据丢失的风险，并确保即使在意外事件期间也能持续提供服务。第三，它具有成本效益，因为您只需为您使用的资源付费，并且可以根据需求扩大或缩小规模，从而无需在硬件上进行大量的前期投资。最后，在云中部署矢量数据库可以增强协作，因为可以从任何地方访问和共享数据，从而促进基于团队的工作和数据驱动的决策。请使用Amazon FSx ONTAP for NetApp ONTAP检查此验证中使用的 milvus 独立架构。



1. 为NetApp ONTAP实例创建Amazon FSx ONTAP ，并记下 VPC、VPC 安全组和子网的详细信息。创建 EC2 实例时需要此信息。您可以在此处找到更多详细信息 - https://us-east-1.console.aws.amazon.com/fsx/home?region=us-east-1#file-system-create
2. 创建一个 EC2 实例，确保 VPC、安全组和子网与Amazon FSx ONTAP for NetApp ONTAP实例的 VPC、安全组和子网匹配。
3. 使用命令“apt-get install nfs-common”安装 nfs-common，并使用“sudo apt-get update”更新包信息。
4. 创建一个挂载文件夹并在其上挂载适用于NetApp ONTAP 的Amazon FSx ONTAP 。

```
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. 使用“apt-get install”安装 Docker 和 Docker Compose。
6. 根据 docker-compose.yaml 文件搭建 Milvus 集群，该文件可以从 Milvus 网站下载。

```
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. 在 docker-compose.yml 文件的“volumes”部分中，将NetApp NFS 挂载点映射到相应的 Milvus 容器路径，具体在 etcd、minio 和 standalone 中。检查"附录 D：docker-compose.yml"有关 yml 更改的详细信息
8. 验证已安装的文件夹和文件。

```
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. 从包含 docker-compose.yml 文件的目录运行“docker-compose up -d”。
10. 检查 Milvus 容器的状态。

```
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. 为了验证Amazon FSx ONTAP for NetApp ONTAP中矢量数据库及其数据的读写功能，我们使用了 Python Milvus SDK 和来自 PyMilvus 的示例程序。使用“apt-get install python3-numpy python3-pip”安装必要的软件包，并使用“pip3 install pymilvus”安装 PyMilvus。
12. 验证向量数据库中Amazon FSx ONTAP for NetApp ONTAP的数据写入和读取操作。

```
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
/448789845791611912/448789845791611913/448789845791611939/103/4487898457
```

```
91411923/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. 使用verify_data_netapp.py脚本检查读取操作。

```
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. 如果客户想要通过 S3 协议访问（读取）矢量数据库中测试的 NFS 数据以用于 AI 工作负载，则可以使用简单的 Python 程序进行验证。一个例子可以是来自另一个应用程序的图像的相似性搜索，如本节开头的图片中提到的那样。

```
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#
```

本节有效地演示了客户如何在 Docker 容器中部署和操作独立的 Milvus 设置，并利用 Amazon 的NetApp FSx ONTAP进行NetApp ONTAP数据存储。此设置允许客户利用矢量数据库的强大功能来处理高维数据和执行复杂查询，所有这些都可以在可扩展且高效的 Docker 容器环境中完成。通过创建适用于NetApp ONTAP实例和匹配的 EC2 实例的Amazon FSx ONTAP ，客户可以确保最佳的资源利用率和数据管理。FSx ONTAP在矢量数据库中数据写入和读取操作的成功验证为客户提供了可靠、一致的数据操作的保证。此外，通过 S3 协议列出（读取）来自 AI 工作负载的数据的能力增强了数据可访问性。因此，这一全面的流程为客户提供了一个强大而高效的解决方案，用于管理他们的大规模数据操作，并利用了 Amazon FSx ONTAP for NetApp ONTAP的功能。

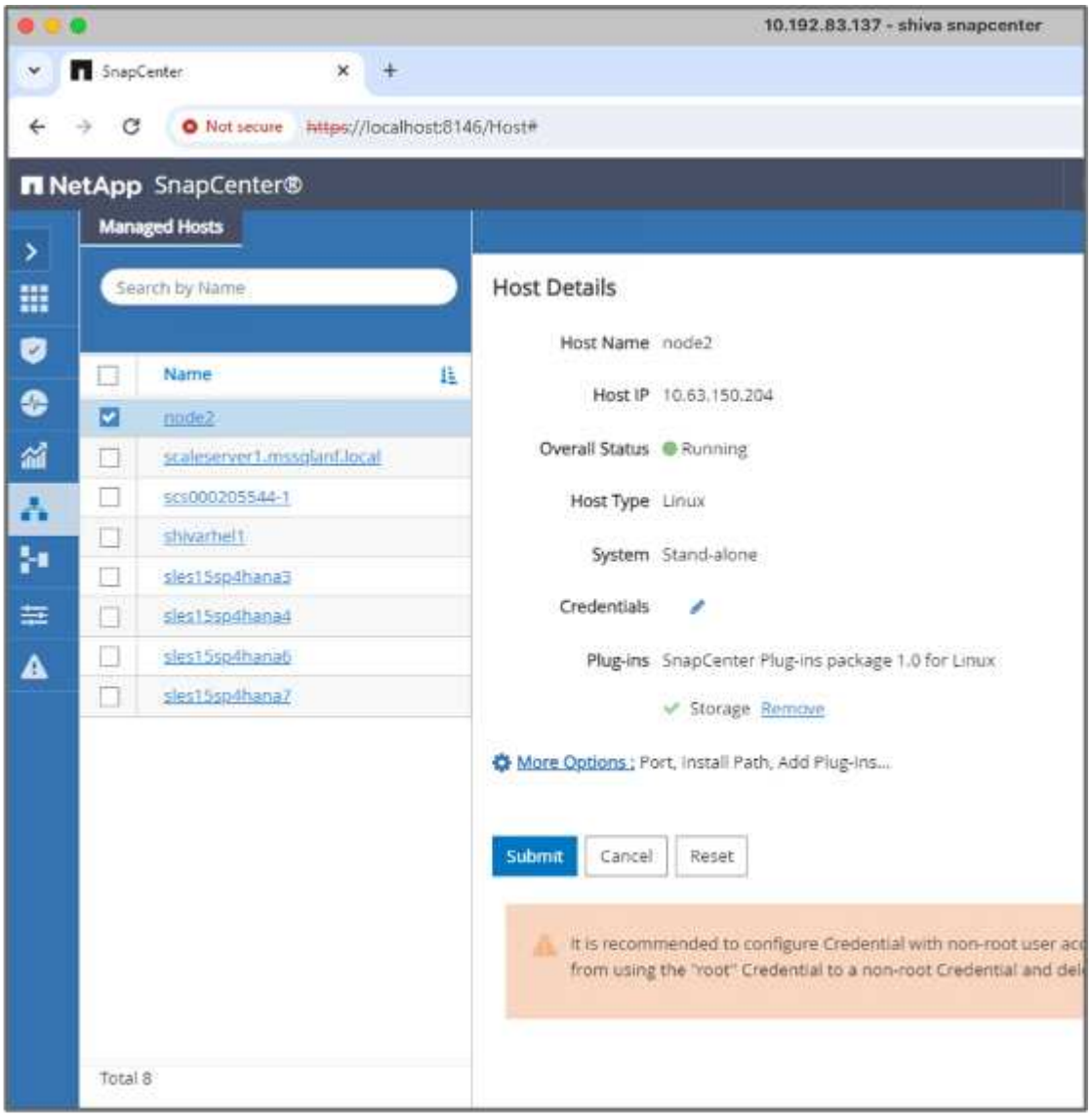
使用SnapCenter进行矢量数据库保护

本节介绍如何使用NetApp SnapCenter为矢量数据库提供数据保护。

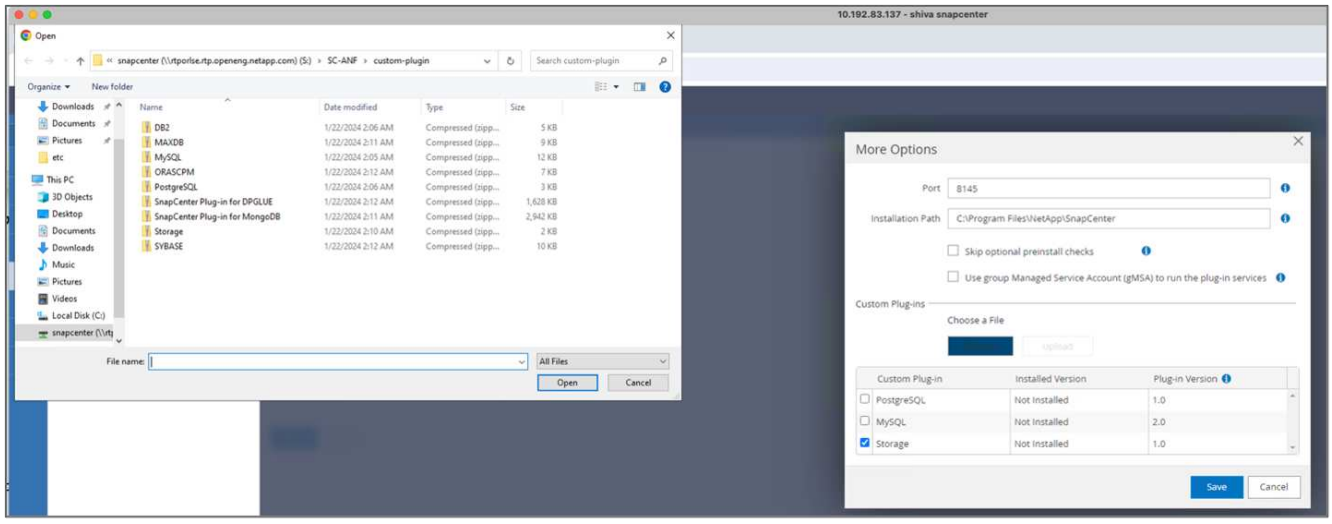
使用**NetApp SnapCenter**进行矢量数据库保护。

例如，在电影制作行业，客户通常拥有关键的嵌入式数据，如视频和音频文件。由于硬盘故障等问题而导致的数据丢失可能会对其运营产生重大影响，甚至可能危及价值数百万美元的企业。我们曾遇到过宝贵内容丢失的情况，造成严重的混乱和经济损失。因此，确保这些重要数据的安全性和完整性对该行业至关重要。在本节中，我们将深入探讨SnapCenter如何保护驻留在ONTAP中的矢量数据库数据和 Milvus 数据。在此示例中，我们使用了从 NFS ONTAP卷 (vol1) 派生的 NAS 存储桶 (milvusdbvol1) 来存储客户数据，并使用了单独的 NFS 卷 (vectordbpv) 来存储 Milvus 集群配置数据。请查看"此处"Snapcenter 备份工作流程

1. 设置将用于执行SnapCenter命令的主机。

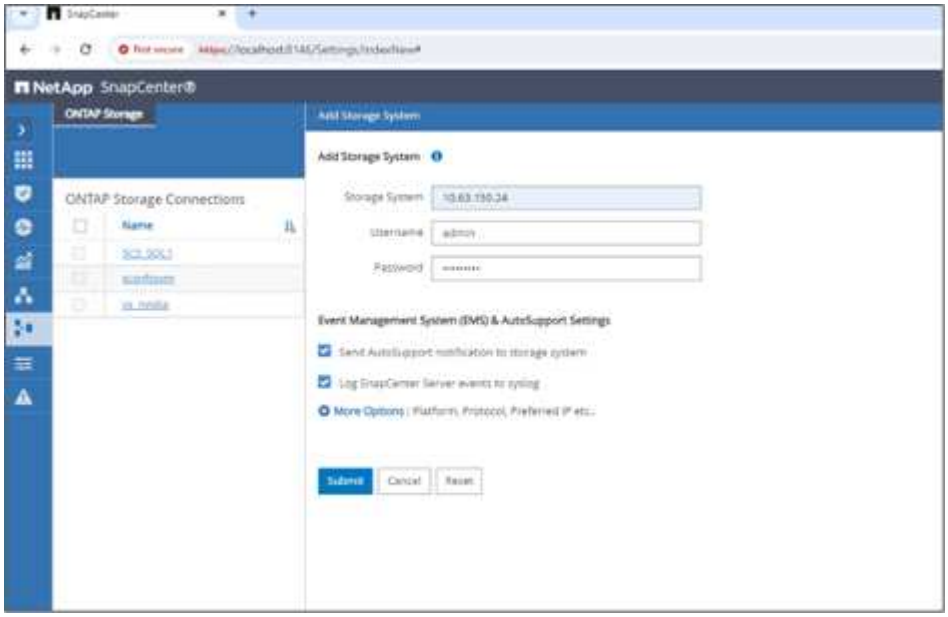


2. 安装并配置存储插件。从添加的主机中，选择“更多选项”。导航到并选择下载的存储插件"NetApp自动化商店"。安装插件并保存配置。

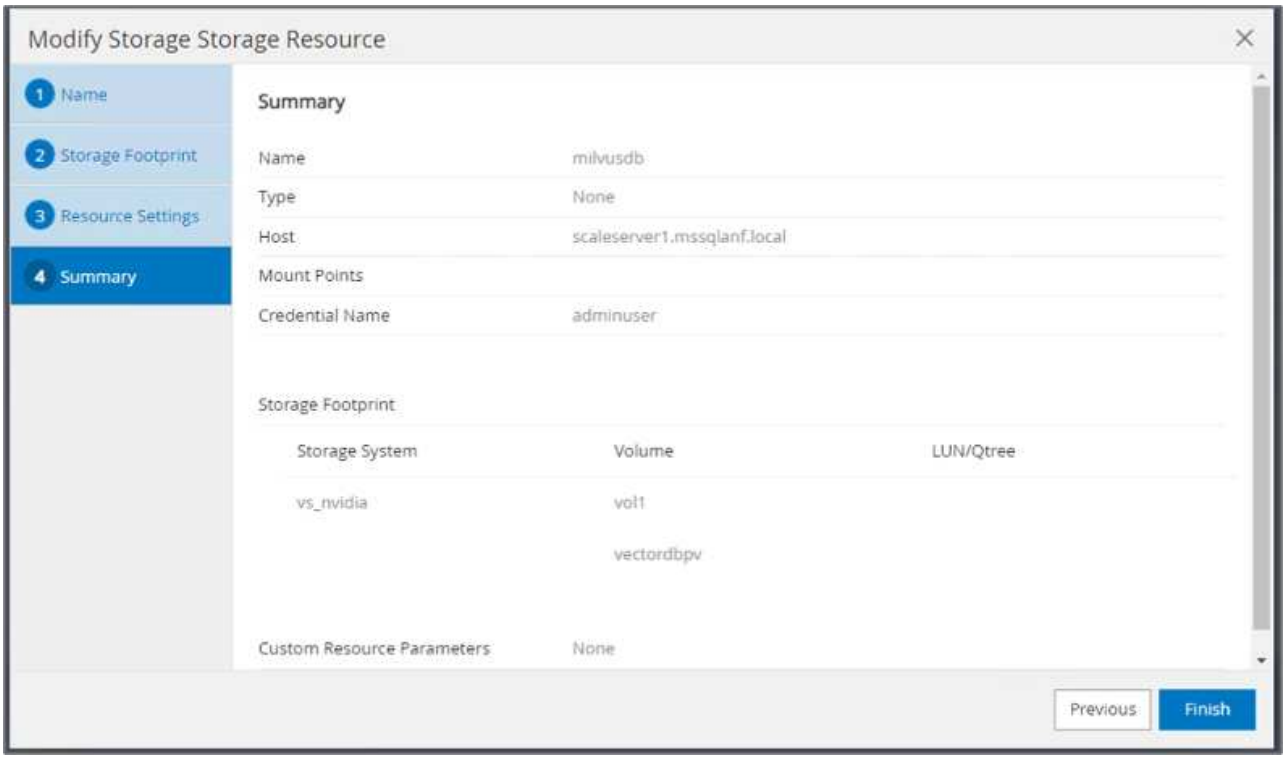


3. 设置存储系统和卷：在“存储系统”下添加存储系统，并选择SVM（存储虚拟机）。在这个例子中，我们选择

了“vs_nvidia”。



4. 为矢量数据库建立资源，包含备份策略和自定义快照名称。
   - 使用默认值启用一致性组备份，并启用不具有文件系统一致性的SnapCenter 。
   - 在存储占用空间部分，选择与矢量数据库客户数据和 Milvus 集群数据关联的卷。在我们的示例中，这些是“vol1”和“vectordbpv”。
   - 创建矢量数据库保护策略，并利用该策略保护矢量数据库资源。



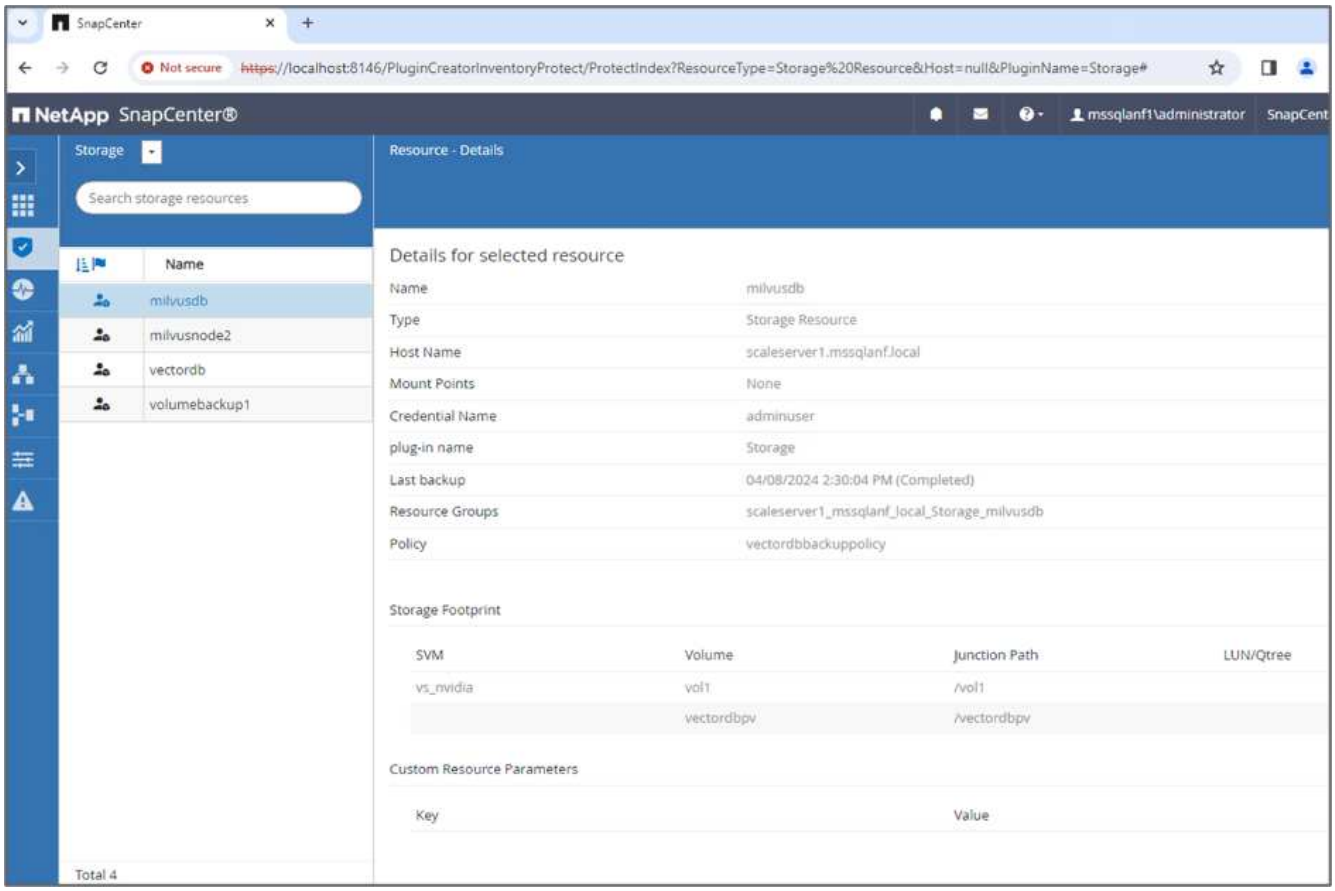
5. 使用 Python 脚本将数据插入 S3 NAS 存储桶。在我们的案例中，我们修改了 Milvus 提供的备份脚本，即“prepare_data_netapp.py”，并执行“sync”命令从操作系统中刷新数据。

```
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. 验证 S3 NAS 存储桶中的数据。在我们的示例中，带有时间戳“2024-04-08 21:22”的文件是由“prepare_data_netapp.py”脚本创建的。

```
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. 使用“milvusdb”资源中的一致性组 (CG) 快照启动备份

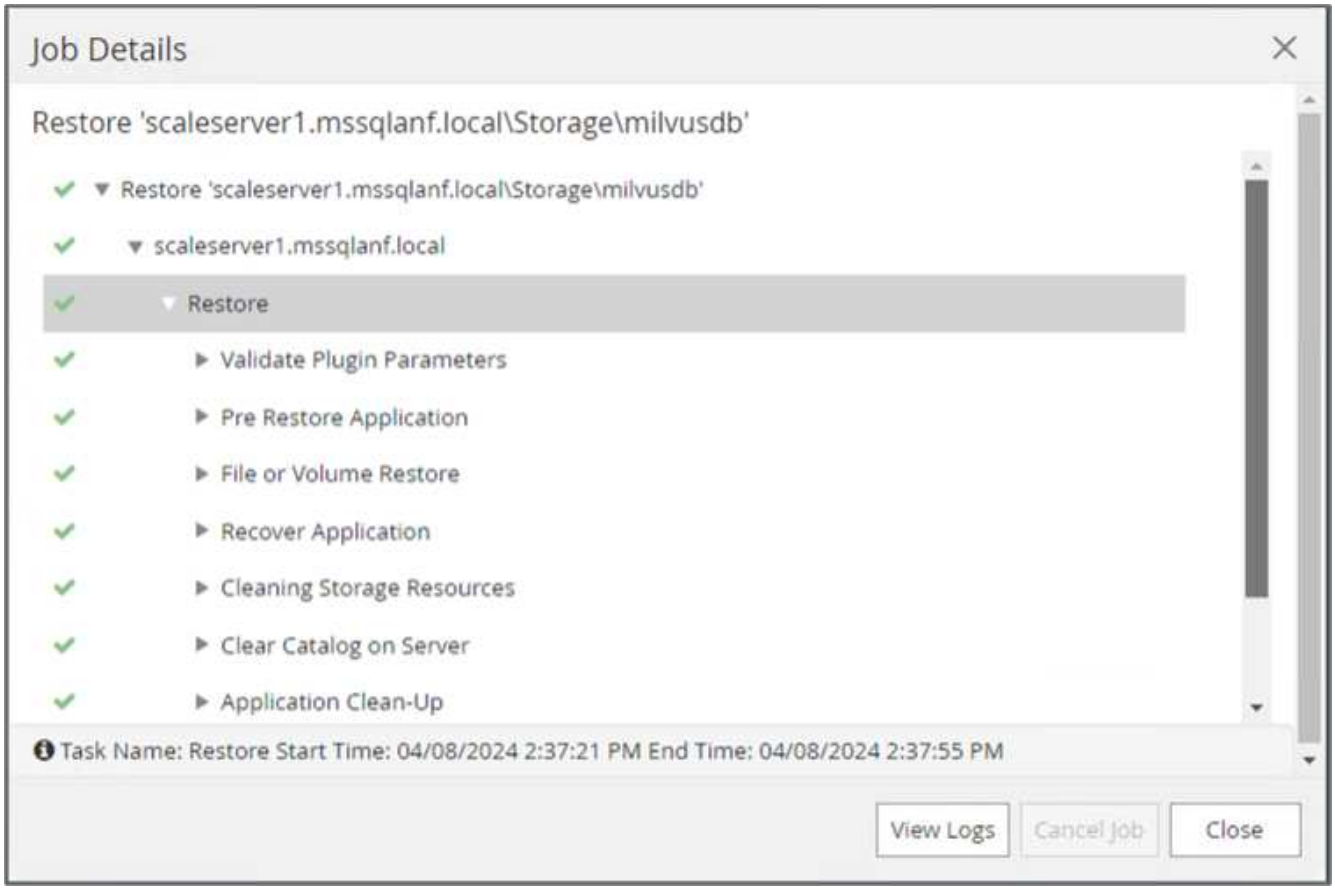


8. 为了测试备份功能，我们在备份过程后添加了一个新表，或者从 NFS（S3 NAS 存储桶）中删除了一些数据。

   对于此测试，想象一下有人在备份后创建了新的、不必要的或不适当的集合的场景。在这种情况下，我们需要将矢量数据库恢复到添加新集合之前的状态。例如，已插入“hello_milvus_netapp_sc_testnew”和“hello_milvus_netapp_sc_testnew2”等新集合。

```
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. 从上一个快照执行 S3 NAS 存储桶的完整恢复。



10. 使用 Python 脚本验证来自“hello_milvus_netapp_sc_test”和“hello_milvus_netapp_sc_test2”集合的数据。

```
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. 验证数据库中不再存在不必要或不适当的集合。

```
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:~#
```

总之，使用 NetApp 的SnapCenter来保护驻留在ONTAP中的矢量数据库数据和 Milvus 数据可以为客户带来显著的优势，特别是在数据完整性至关重要的行业，例如电影制作。 SnapCenter 能够创建一致的备份并执行完整的数据恢复，确保关键数据（例如嵌入式视频和音频文件）不会因硬盘故障或其他问题而丢失。这不仅可以防止运营中断，还可以防止重大财务损失。

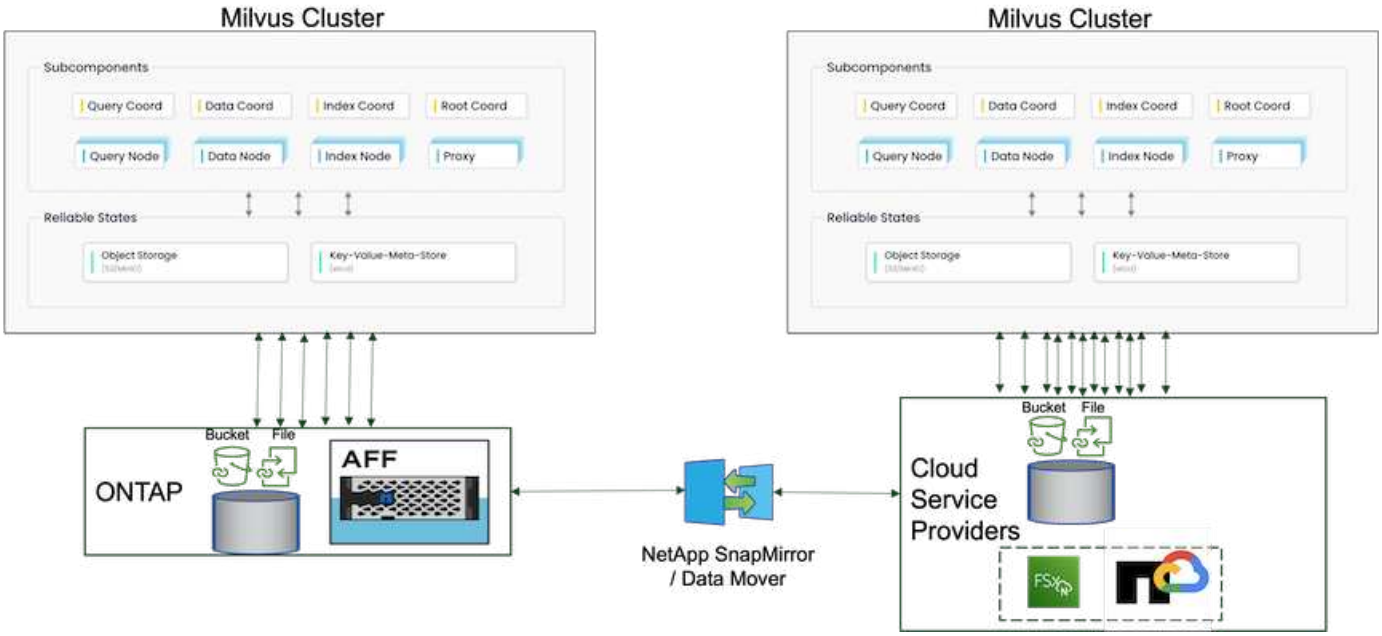
在本节中，我们演示了如何配置SnapCenter来保护驻留在ONTAP中的数据，包括主机的设置、存储插件的安装和配置，以及使用自定义快照名称为矢量数据库创建资源。我们还展示了如何使用一致性组快照执行备份并验证 S3 NAS 存储桶中的数据。

此外，我们模拟了备份后创建不必要或不适当的集合的情况。在这种情况下，SnapCenter 从以前的快照执行完整恢复的能力可确保矢量数据库可以恢复到添加新集合之前的状态，从而保持数据库的完整性。这种将数据恢复到特定时间点的功能对于客户来说非常宝贵，它为他们提供了保证，确保他们的数据不仅安全，而且得到正确的维护。因此，NetApp 的SnapCenter产品为客户提供了强大而可靠的数据保护和管理解决方案。

使用NetApp SnapMirror进行灾难恢复

本节讨论使用SnapMirror为NetApp实现矢量数据库解决方案的 DR（灾难恢复）。

使用NetApp SnapMirror进行灾难恢复



灾难恢复对于维护矢量数据库的完整性和可用性至关重要，尤其是考虑到其在管理高维数据和执行复杂相似性搜索中的作用。精心规划和实施的灾难恢复策略可确保在发生硬件故障、自然灾害或网络攻击等不可预见的事件时数据不会丢失或受到损害。这对于依赖矢量数据库的应用程序尤其重要，因为数据的丢失或损坏可能导致严重的运营中断和财务损失。此外，强大的灾难恢复计划还可以最大限度地减少停机时间并允许快速恢复服务，从而确保业务连续性。这是通过NetApp数据复制产品 SnapMirror 跨不同地理位置、定期备份和故障转移机制实现的。因此，灾难恢复不仅仅是一种保护措施，而且是负责任、高效的矢量数据库管理的重要组成部分。

NetApp 的SnapMirror提供从一个NetApp ONTAP存储控制器到另一个存储控制器的数据复制，主要用于灾难恢复 (DR) 和混合解决方案。在矢量数据库的背景下，该工具有助于实现数据在本地和云环境之间的平稳过渡。这种转变无需任何数据转换或应用程序重构即可实现，从而提高了跨多个平台数据管理的效率和灵活性。

NetApp Hybrid解决方案在矢量数据库场景下可以带来更多优势：

1. 可扩展性：NetApp 的混合云解决方案能够根据您的需求扩展您的资源。您可以利用本地资源来处理常规、可预测的工作负载，并利用云资源（例如Amazon FSx ONTAP for NetApp ONTAP和 Google Cloud NetApp Volume（NetApp Volumes））来应对高峰时段或意外负载。
2. 成本效益：NetApp 的混合云模型允许您通过使用内部资源来处理常规工作负载并仅在需要时支付云资源费用，从而优化成本。这种按需付费模式通过NetApp instaclustr 服务产品可以实现相当高的成本效益。对于本地和主要云服务提供商，instaclustr 提供支持和咨询。
3. 灵活性：NetApp 的混合云让您可以灵活地选择在何处处理数据。例如，您可能选择在拥有更强大硬件的本地执行复杂的矢量操作，而在云中执行不太密集的操作。
4. 业务连续性：如果发生灾难，将数据保存在NetApp混合云中可以确保业务连续性。如果您的本地资源受到影响，您可以快速切换到云端。我们可以利用NetApp SnapMirror将数据从本地移动到云端，反之亦然。
5. 创新：NetApp 的混合云解决方案还可以通过提供对尖端云服务和技术的访问来实现更快的创新。 NetApp在云领域的创新，例如Amazon FSx ONTAP for NetApp ONTAP、 Azure NetApp Files和Google Cloud NetApp Volumes都是云服务提供商的创新产品和首选 NAS。

矢量数据库性能验证

本节重点介绍在矢量数据库上执行的性能验证。

性能验证

性能验证在矢量数据库和存储系统中都起着至关重要的作用，是确保最佳运行和高效资源利用的关键因素。矢量数据库以处理高维数据和执行相似性搜索而闻名，需要保持高性能水平才能快速准确地处理复杂查询。性能验证有助于识别瓶颈、微调配置并确保系统能够处理预期负载而不会降低服务质量。同样，在存储系统中，性能验证对于确保数据高效存储和检索至关重要，不会出现可能影响整体系统性能的延迟问题或瓶颈。它还有助于对存储基础设施的必要升级或变更做出明智的决策。因此，性能验证是系统管理的一个重要方面，对维持高服务质量、运行效率和整体系统可靠性有重要贡献。

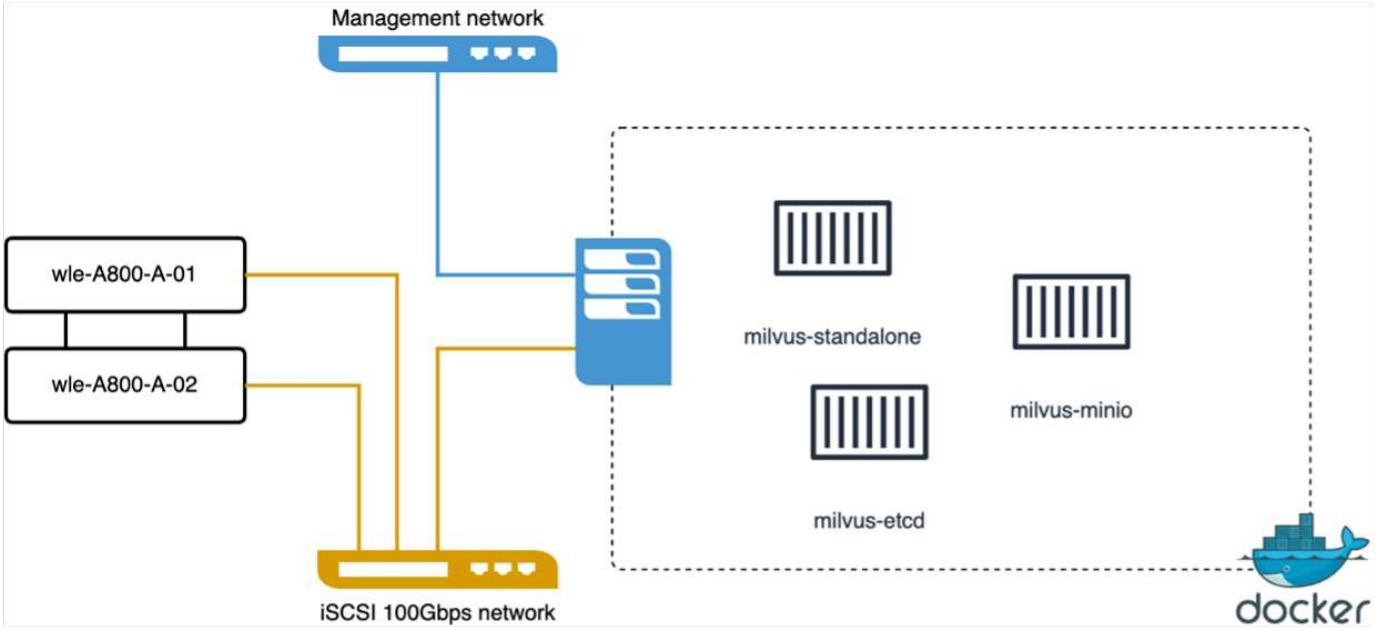
在本节中，我们旨在深入研究矢量数据库（例如 Milvus 和 pgvecto.rs）的性能验证，重点关注它们的存储性能特征，例如 I/O 配置文件和 NetApp 存储控制器在 LLM 生命周期内支持 RAG 和推理工作负载的行为。当这些数据库与ONTAP存储解决方案结合时，我们将评估并识别任何性能差异因素。我们的分析将基于关键性能指标，例如每秒处理的查询数（QPS）。

请检查下面用于 milvus 和进度的方法。

详细信息	Milvus（单机和集群）	Postgres（pgvecto.rs）#
version	2.3.2	0.2.0
Filesystem	iSCSI LUN 上的 XFS	
工作负载生成器	"VectorDB-Bench"– v0.0.5	
数据集	LAION 数据集 * 1000 万个嵌入 * 768 个维度 * 数据集大小约为 300GB	
存储控制器	AFF 800 * 版本 — 9.14.1 * 4 x 100GbE — 用于 milvus，2x 100GbE 用于 postgres * iscsi	

带有 **Milvus** 独立集群的 **VectorDB-Bench**

我们使用vectorDB-Bench在milvus独立集群上进行了以下性能验证。 milvus 独立集群的网络和服务器连接如下。



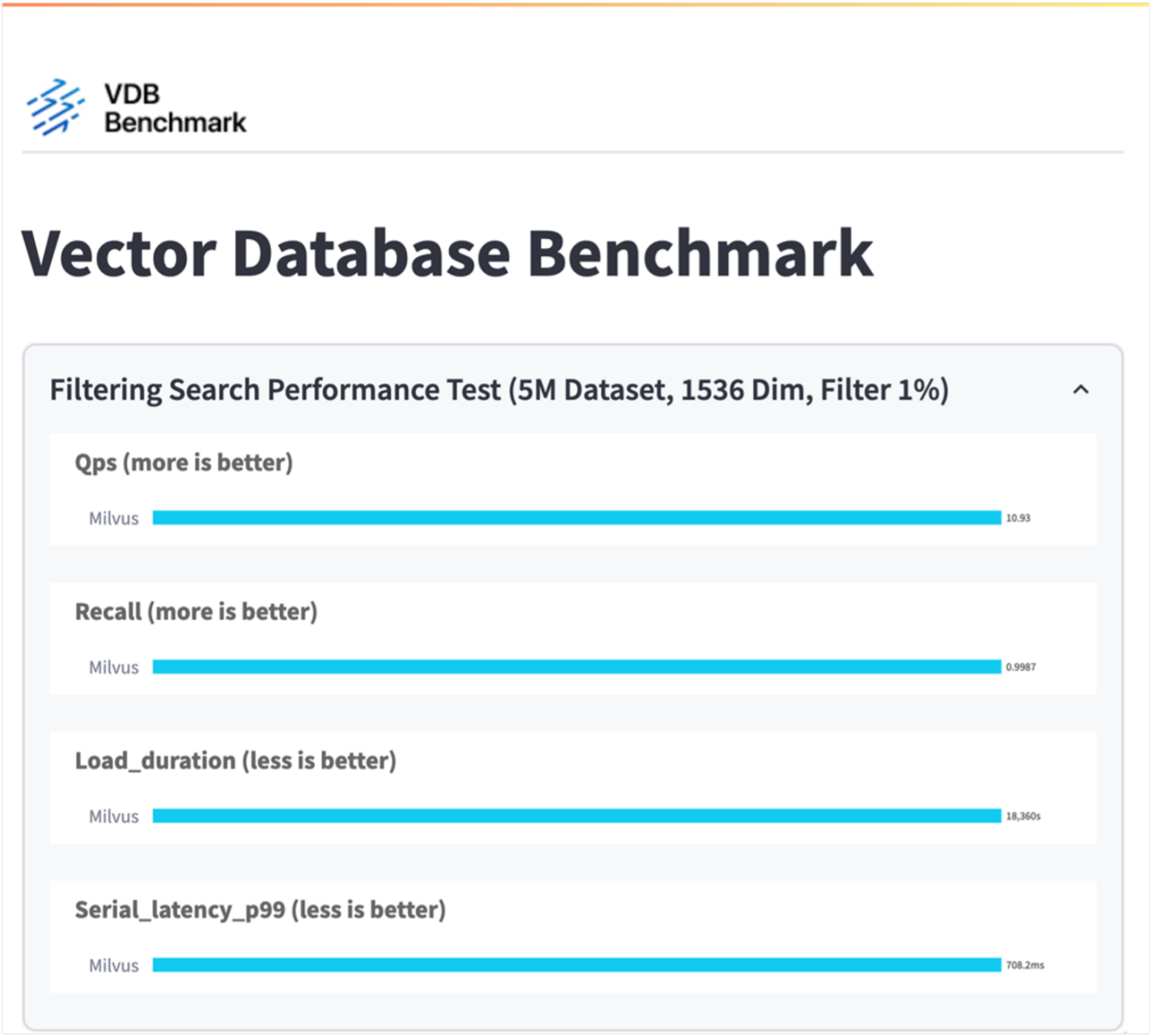
在本节中，我们分享测试 Milvus 独立数据库的观察和结果。。我们选择 DiskANN 作为这些测试的索引类型。。提取、优化和创建大约 100GB 数据集的索引大约需要 5 个小时。在此持续时间的大部分时间里，配备 20 个内核（启用超线程时相当于 40 个 vCPU）的 Milvus 服务器都以其最大 CPU 容量 100% 运行。我们发现 DiskANN 对于超过系统内存大小的大型数据集尤为重要。。在查询阶段，我们观察到每秒查询次数 (QPS) 为 10.93，召回率为 0.9987。查询的第 99 个百分位延迟测量为 708.2 毫秒。

从存储角度来看，数据库在摄取、插入后优化和索引创建阶段发出大约 1,000 个操作/秒。在查询阶段，它要求每秒 32,000 次操作。

以下部分介绍存储性能指标。

工作负载阶段	指标	值
数据提取和插入后优化	IOPS	< 1,000
	延迟	< 400 微秒
	工作量	读/写混合，主要是写入
	IO 大小	64 KB
查询	IOPS	峰值为32,000
	延迟	< 400 微秒
	工作量	100% 缓存读取
	IO 大小	主要为8KB

VectorDB-bench 结果如下。

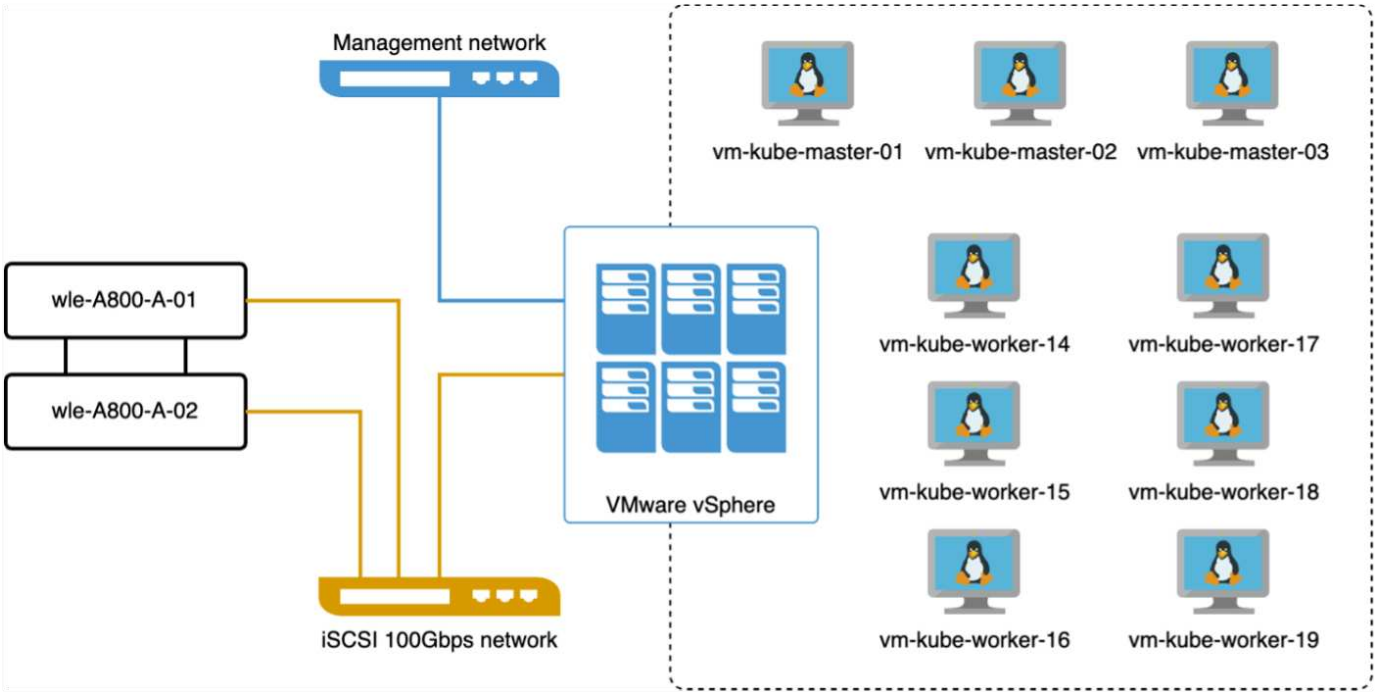


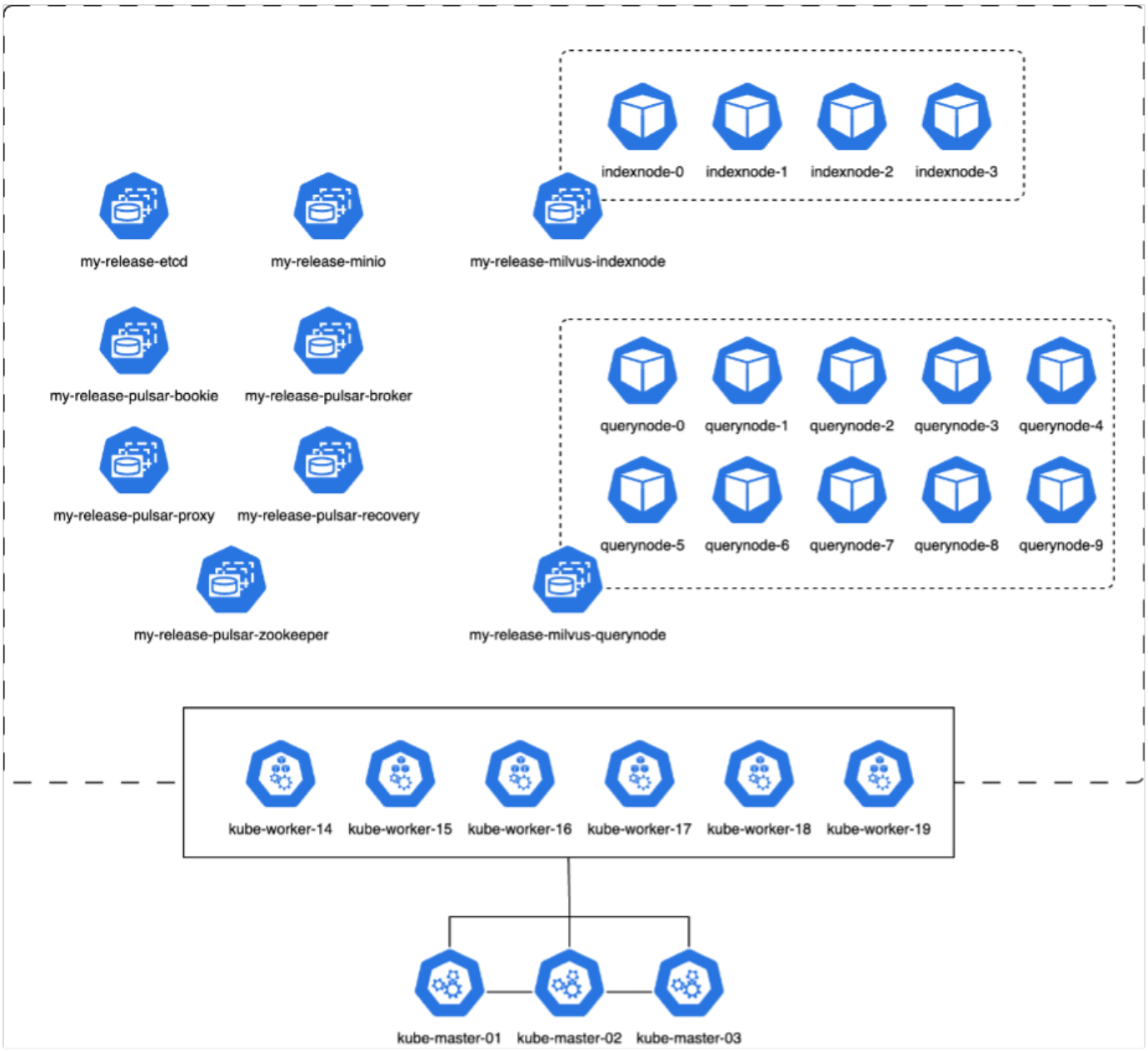
从独立 Milvus 实例的性能验证来看，当前的设置不足以支持 500 万个向量、维度为 1536 的数据集。我们已确定存储拥有足够的资源，不会构成系统的瓶颈。

带有 **milvus** 集群的 **VectorDB-Bench**

在本节中，我们讨论在 Kubernetes 环境中部署 Milvus 集群。此 Kubernetes 设置构建于 VMware vSphere 部署之上，该部署托管 Kubernetes 主节点和工作节点。

以下部分介绍 VMware vSphere 和 Kubernetes 部署的详细信息。


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



在本节中，我们介绍了测试 Milvus 数据库的观察结果和结果。 * 使用的索引类型是 DiskANN。 * 下表比较了在处理 500 万个向量（维度为 1536）时独立部署和集群部署的差异。我们观察到，在集群部署中，数据提取和插入后优化所需的时间较短。与独立设置相比，集群部署中查询的第 99 个百分位延迟减少了六倍。 * 尽管集群部署中的每秒查询数 (QPS) 率较高，但并未达到预期水平。

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%

下图提供了各种存储指标的视图，包括存储集群延迟和总 IOPS（每秒输入/输出操作）。

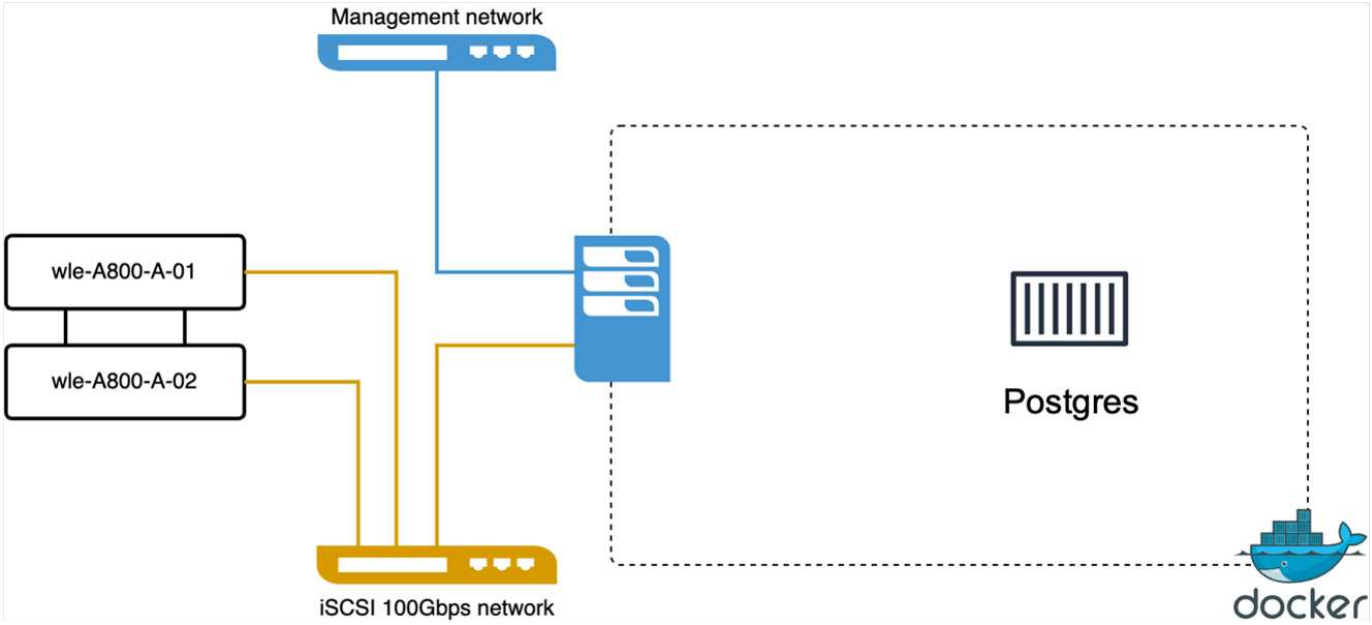
以下部分介绍关键的存储性能指标。

工作负载阶段	指标	值
数据提取和插入后优化	IOPS	< 1,000
	延迟	< 400 微秒
	工作量	读/写混合，主要是写入
	IO 大小	64 KB
查询	IOPS	峰值为147,000
	延迟	< 400 微秒
	工作量	100% 缓存读取
	IO 大小	主要为8KB

基于独立 Milvus 和 Milvus 集群的性能验证，我们展示了存储 I/O 配置文件的详细信息。 * 我们观察到 I/O 配置文件在独立部署和集群部署中保持一致。 * 峰值 IOPS 的观察到的差异可以归因于集群部署中的客户端数量较多。

带有 **Postgres** 的**vectorDB-Bench**（**pgvecto.rs**）

我们使用 VectorDB-Bench 对 PostgreSQL（pgvecto.rs）进行了如下操作：PostgreSQL（具体来说，pgvecto.rs）的网络和服务器连接详情如下：

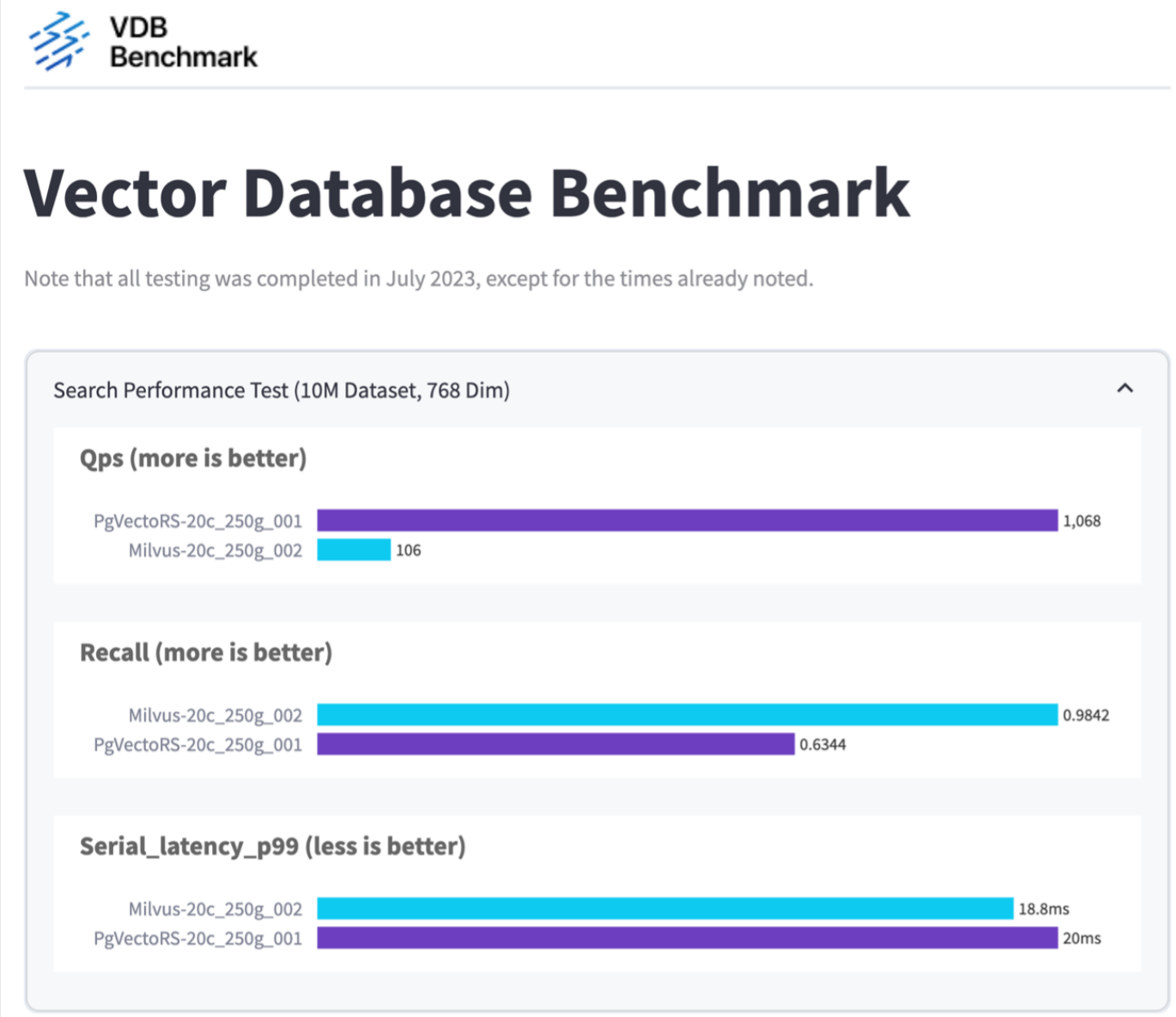


在本节中，我们分享测试 PostgreSQL 数据库（特别是使用 pgvecto.rs）的观察和结果。 * 我们选择 HNSW 作为这些测试的索引类型，因为在测试时，DiskANN 不适用于 pgvecto.rs。 * 在数据提取阶段，我们加载了 Cohere 数据集，该数据集包含 1000 万个向量，维度为 768。该过程大约耗时 4.5 小时。 * 在查询阶段，我们观察到每秒查询次数 (QPS) 为 1,068，召回率为 0.6344。查询的第 99 个百分位延迟测量为 20 毫秒。在大部分运行时间内，客户端 CPU 都以 100% 的容量运行。

下图提供了各种存储指标的视图，包括存储集群延迟总 IOPS（每秒输入/输出操作）。


```
 The following section presents the key storage performance metrics.
image:pgvecto-storage-perf-metrics.png["该图显示输入/输出对话框或表示书面内容"]
```

milvus 与 **postgres** 在 **Vector DB Bench** 上的性能对比



根据我们使用 VectorDBBench 对 Milvus 和 PostgreSQL 进行的性能验证，我们观察到以下情况：

- 索引类型：HNSW
- 数据集：包含 768 个维度的 1000 万个向量

我们发现 pgvecto.rs 的每秒查询数 (QPS) 达到 1,068，召回率为 0.6344，而 Milvus 的每秒查询数 (QPS) 达到 106，召回率为 0.9842。

如果您优先考虑查询的高精度，那么 Milvus 的性能优于 pgvecto.rs，因为它在每个查询中检索到更高比例的相关项目。但是，如果每秒查询次数是一个更关键的因素，那么 pgvecto.rs 就超过了 Milvus。但值得注意的是，通过 pgvecto.rs 检索的数据质量较低，大约 37% 的搜索结果是不相关的项目。

根据我们的性能验证得出的观察结果：

根据我们的性能验证，我们做出了以下观察：

在 Milvus 中，I/O 配置文件与 OLTP 工作负载非常相似，例如 Oracle SLOB 中的工作负载。基准测试包括三个阶段：数据提取、后优化和查询。初始阶段主要以 64KB 写入操作为特征，而查询阶段主要涉及 8KB 读取。我

们希望ONTAP能够熟练地处理 Milvus I/O 负载。

PostgreSQL I/O 配置文件不会带来具有挑战性的存储工作负载。鉴于目前正在进行的内存实现，我们在查询阶段没有观察到任何磁盘 I/O。

DiskANN 成为存储区分的关键技术。它使得向量数据库搜索能够超越系统内存边界进行有效扩展。然而，不太可能通过内存中的向量数据库索引（例如 HNSW）建立存储性能差异。

还值得注意的是，当索引类型为 HSNW 时，存储在查询阶段并不起关键作用，而查询阶段是支持 RAG 应用的矢量数据库最重要的操作阶段。这里的含义是存储性能不会显著影响这些应用程序的整体性能。

使用 PostgreSQL 的 Instaclustr 矢量数据库：pgvector

本节讨论 instaclustr 产品如何与NetApp矢量数据库解决方案中的 postgreSQL 的 pgvector 功能集成的具体细节。

使用 PostgreSQL 的 Instaclustr 矢量数据库：pgvector

在本节中，我们将深入探讨 instaclustr 产品如何在 pgvector 功能上与 postgreSQL 集成的具体细节。我们有一个例子“如何使用 PGVector 和 PostgreSQL 提高 LLM 准确性和性能：嵌入简介和 PGVector 的作用”。请检查"博客"以获取更多信息。

矢量数据库用例

本节概述了NetApp矢量数据库解决方案的用例。

矢量数据库用例

在本节中，我们讨论两个用例，例如使用大型语言模型的检索增强生成和NetApp IT 聊天机器人。

使用大型语言模型 (LLM) 进行检索增强生成 (RAG)

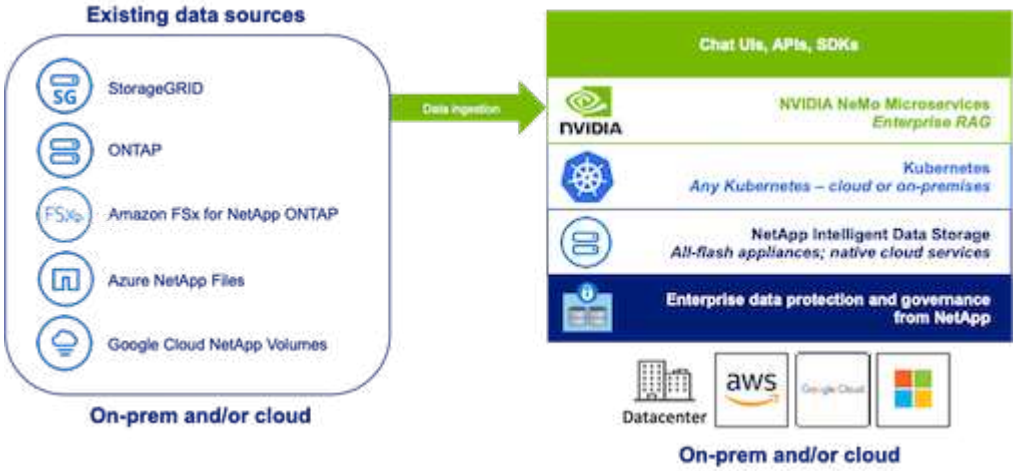
```
Retrieval-augmented generation, or RAG, is a technique for enhancing the
accuracy and reliability of Large Language Models, or LLMs, by augmenting
prompts with facts fetched from external sources. In a traditional RAG
deployment, vector embeddings are generated from an existing dataset and
then stored 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.
```

NVIDIA Enterprise RAG LLM Operator 是在企业中实施 RAG 的有用工具。该操作员可用于部署完整的 RAG 管

道。 RAG 管道可以定制为使用 Milvus 或 pgvecto 作为存储知识库嵌入的向量数据库。有关详细信息，请参阅文档。

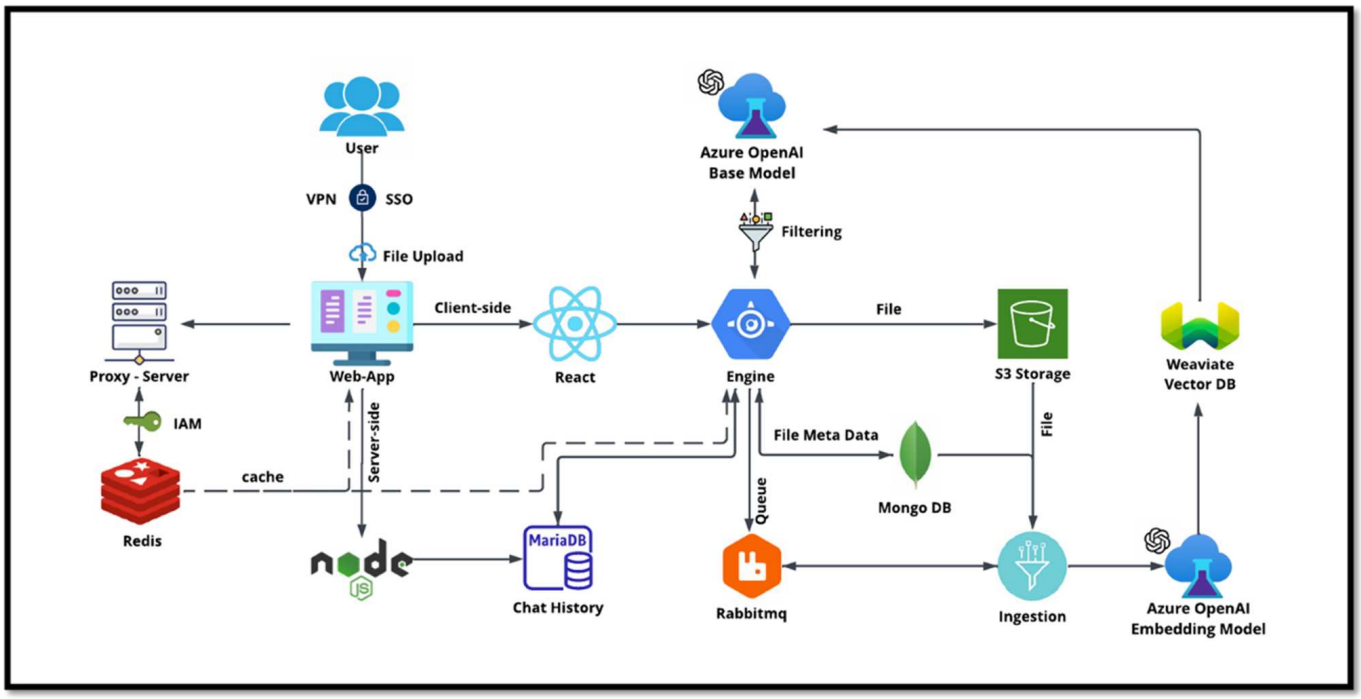
```
NetApp has validated an enterprise RAG architecture powered by the NVIDIA
Enterprise RAG LLM Operator alongside NetApp storage. Refer to our blog
post for more information and to see a demo. Figure 1 provides an overview
of this architecture.
```

图 1) 由NVIDIA NeMo 微服务和NetApp提供支持的企业 RAG



NetApp IT 聊天机器人用例

NetApp 的聊天机器人是矢量数据库的另一个实时用例。在这种情况下， NetApp Private OpenAI Sandbox 为管理来自 NetApp 内部用户的查询提供了一个有效、安全且高效的平台。通过结合严格的安全协议、高效的数据管理系统和复杂的人工智能处理能力，它保证通过 SSO 身份验证根据组织中用户的角色和职责为他们提供高质量、精确的响应。这种架构凸显了融合先进技术以创建以用户为中心的智能系统的潜力。



用例可以分为四个主要部分。

用户身份验证和验证：

- 用户查询首先经过NetApp单点登录 (SSO) 流程来确认用户的身份。
- 身份验证成功后，系统会检查VPN连接以确保数据传输的安全。

数据传输和处理：

- 一旦 VPN 验证通过，数据就会通过 NetAIChat 或 NetAICreate Web 应用程序发送到 MariaDB。 MariaDB 是一个快速高效的数据库系统，用于管理和存储用户数据。
- 然后，MariaDB 将信息发送到NetApp Azure 实例，该实例将用户数据连接到 AI 处理单元。

与 **OpenAI** 和内容过滤的交互：

- Azure 实例将用户的问题发送到内容过滤系统。该系统清理查询并准备进行处理。
- 清理后的输入随后被发送到 Azure OpenAI 基础模型，该模型根据输入生成响应。

响应生成和审核：

- 首先检查基础模型的响应，以确保其准确性并符合内容标准。
- 检查通过后，将响应发送回用户。此过程可确保用户收到对其查询的清晰、准确和适当的答案。

结束语

本节总结了NetApp的矢量数据库解决方案。

结束语

总而言之，本文档全面概述了在NetApp存储解决方案上部署和管理矢量数据库（例如 Milvus 和 pgvector）。我们讨论了利用NetApp ONTAP和StorageGRID对象存储的基础设施指南，并通过文件和对象存储验证了 AWS FSx ONTAP中的 Milvus 数据库。

我们探索了 NetApp 的文件对象二元性，证明了它不仅适用于矢量数据库中的数据，也适用于其他应用程序。我们还重点介绍了 NetApp 的企业管理产品SnapCenter如何为矢量数据库数据提供备份、恢复和克隆功能，确保数据的完整性和可用性。

该文档还深入探讨了 NetApp 的混合云解决方案如何在本地和云环境中提供数据复制和保护，从而提供无缝、安全的数据管理体验。我们对NetApp ONTAP上 Milvus 和 pgvecto 等矢量数据库的性能验证提供了见解，并提供了有关其效率和可扩展性的宝贵信息。

最后，我们讨论了两个生成式 AI 用例：带有 LLM 的 RAG 和 NetApp 的内部 ChatAI。这些实际示例强调了本文档中概述的概念和实践的实际应用和好处。总的来说，对于任何希望利用 NetApp 强大的存储解决方案来管理矢量数据库的人来说，本文档都是一份全面的指南。

声明

作者衷心感谢以下贡献者以及其他提供反馈和评论的人，使本文对NetApp客户和NetApp领域具有价值。

1. Sathish Thyagarajan， NetApp ONTAP AI 与分析技术营销工程师
2. NetApp技术营销工程师 Mike Oglesby
3. NetApp高级总监 AJ Mahajan
4. NetApp工作负载性能工程经理 Joe Scott
5. NetApp Fsx 产品管理高级总监 Puneet Dhawan
6. NetApp FSx 产品团队高级产品经理 Yuval Kalderon

在哪里可以找到更多信息

要了解有关本文档中描述的信息的更多信息，请查看以下文档和/或网站：

- Milvus 文档 - https://milvus.io/docs/overview.md
- Milvus 独立文档 - https://milvus.io/docs/v2.0.x/install_standalone-docker.md
- NetApp产品文档https://www.netapp.com/support-and-training/documentation/[]
- instaclustr -"installclustr 文档"

版本历史

版本	日期	文档版本历史
1.0 版	2024年4月	初始版本

附录 A：Values.yaml

本节提供NetApp矢量数据库解决方案中使用的值的示例 YAML 代码。

附录 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:~#
```

附录 B：prepare_data_netapp_new.py

本节提供用于准备矢量数据库数据的Python脚本示例。

附录 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:~#
```

附录C：verify_data_netapp.py

本节包含一个示例 Python 脚本，可用于验证NetApp矢量数据库解决方案中的矢量数据库。

附录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:~#
```

附录 D：docker-compose.yml

本节包含NetApp矢量数据库解决方案的示例 YAML 代码。

附录 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
```