NetApp를 사용한 벡터 데이터베이스 솔루션

NetApp Solutions

NetApp
May 03, 2024

목차

NetApp를 사용한 벡터 데이터베이스 솔루션

카르티키안 나가링감(Karthikean Nagalingam)과 로드리고 나시멘토(NetApp)

이 문서에서는 NetApp의 스토리지 솔루션을 사용하여 Milvus, pgvecto 오픈 소스 PostgreSQL 확장과 같은 벡터 데이터베이스의 구축과 관리에 대해 자세히 살펴봅니다. NetApp ONTAP 및 StorageGRID 오브젝트 스토리지를 사용하기 위한 인프라 지침을 자세히 살펴보고 AWS FSx for NetApp ONTAP에서 Milvus 데이터베이스 애플리케이션을 검증합니다. 이 문서는 NetApp의 파일 오브젝트 이중화와 벡터 임베드 기능을 지원하는 벡터 데이터베이스 및 애플리케이션에 대한 해당 유틸리티를 활용하고자 합니다. 또한 벡터 데이터베이스에 대한 백업 및 복원 기능을 제공하여 데이터 무결성과 가용성을 보장하는 NetApp의 엔터프라이즈 관리 제품인 SnapCenter의 기능을 강조합니다. 본 문서에서는 NetApp의 하이브리드 클라우드 솔루션에 대해 자세히 알아보고 온프레미스 및 클라우드 환경 전반에 걸쳐 데이터 복제 및 보호의 역할에 대해 설명합니다. NetApp ONTAP 기반 벡터 데이터베이스의 성능 검증에 대한 통찰력을 포함하고 있으며, 생성 AI에 대한 두 가지 실용적인 사용 사례인 LLM과 NetApp의 내부 ChatAI로 결론을 맺습니다. 이 문서는 벡터 데이터베이스 관리를 위한 NetApp의 스토리지 솔루션을 활용하기 위한 포괄적인 가이드입니다.

레퍼런스 아키텍처는 다음 사항에 중점을 둡니다.

소개

소개

벡터 데이터베이스는 대규모 언어 모델(LLM) 및 세대 인공 지능(AI)에서 의미 검색의 복잡성을 처리하도록 설계된 문제를 효과적으로 해결합니다. 기존의 데이터 관리 시스템과 달리 벡터 데이터베이스는 이미지, 비디오, 텍스트, 오디오, 비디오, 비디오, 오디오, 비디오 등 다양한 유형의 데이터를 처리 및 검색할 수 있습니다. 데이터 자체의 내용을 레이블이나 태그 대신 사용하여 다른 형태의 비정형 데이터를 만들 수 있습니다.

RDBMS(Relational Database Management Systems)의 한계는 쉽게 문서화되어 있으며, 특히 AI 애플리케이션에서 일반적으로 나타나는 구조화되지 않은 데이터를 다루는 데 어려움을 겪고 있습니다. RDBMS는 데이터를 관리 가능한 구조로 변환하는 데 시간이 많이 걸리고 오류가 발생하기 쉬운 프로세스를 필요로 하기 때문에 검색이 지연되고 비효율성이 초래되는 경우가 많습니다. 그러나 벡터 데이터베이스는 이러한 문제를 회피하도록 설계되었으므로 복잡하고 높은 차원 데이터를 관리하고 검색하여 AI 애플리케이션의 발전을 촉진할 수 있는 보다 효율적이고 정확한 솔루션을 제공합니다.

이 문서는 벡터 데이터베이스를 현재 사용 중이거나 사용할 계획이 있는 고객을 위한 포괄적인 가이드 역할을 하며 NetApp ONTAP, NetApp StorageGRID, Amazon FSxN for NetApp ONTAP 및 SnapCenter 등과 같은 플랫폼에서 벡터 데이터베이스를 활용하는 모범 사례를 자세히 설명합니다. 여기에 제공된 내용은 다음과 같은 다양한 주제를 다룹니다.

- NetApp ONTAP 및 StorageGRID 오브젝트 스토리지를 통해 NetApp 스토리지가 제공하는 Milvus와 같은 벡터 데이터베이스에 대한 인프라 지침
- 파일 및 오브젝트 저장소를 통해 AWS FSx for NetApp ONTAP의 Milvus 데이터베이스 검증
- NetApp의 파일 오브젝트 이중성을 자세히 살펴보고 벡터 데이터베이스와 기타 애플리케이션의 데이터에 대한 유틸리티를 보여줍니다.
- NetApp의 데이터 보호 관리 제품인 SnapCenter이 벡터 데이터베이스 데이터에 대한 백업 및 복원 기능을 제공하는 방법
- NetApp의 하이브리드 클라우드가 어떻게 사내 및 클라우드 환경 전반의 데이터 복제와 보호를 제공하는지 소개합니다.
- NetApp ONTAP 기반 Milvus 및 pgvector와 같은 벡터 데이터베이스의 성능 검증에 대한 통찰력을 제공합니다.
- 두 가지 구체적인 사용 사례로는 LLM(대형 언어 모델)을 사용한 수집 증강 생성(RAG)과 NetApp IT 팀의 ChatAI를 통해 간략히 설명된 개념과 관행의 실질적인 예를 제공합니다.

솔루션 개요

솔루션 개요

이 솔루션은 벡터 데이터베이스 고객이 직면한 과제를 해결하기 위해 NetApp 가 제공할 수 있는 고유한 이점과 기능을 보여줍니다. 고객은 NetApp ONTAP, NetApp의 클라우드 솔루션, StorageGRID 및 SnapCenter를 활용하여 비즈니스 운영에 상당한 가치를 더할 수 있습니다. 이러한 도구는 기존 문제를 해결할 뿐 아니라 효율성과 생산성을 향상시켜 전체 비즈니스 성장에 기여합니다.

NetApp를 선택해야 하는 이유

- ONTAP 및 StorageGRID와 같은 NetApp 제품을 사용하면 스토리지와 컴퓨팅을 분리함으로써 특정 요구사항에 따라 최적의 리소스 활용률을 달성할 수 있습니다. 이러한 유연성 덕분에 고객은 NetApp 스토리지 솔루션을 사용하여 스토리지를 독립적으로 확장할 수 있습니다.

- NetApp의 스토리지 컨트롤러를 활용하는 고객은 NFS 및 S3 프로토콜을 사용하여 벡터 데이터베이스에 데이터를 효율적으로 제공할 수 있습니다. 이러한 프로토콜은 고객 데이터 스토리지를 용이하게 하고 벡터 데이터베이스 인덱스를 관리하므로 파일 및 개체 방법을 통해 액세스되는 데이터의 여러 복사본이 필요하지 않습니다.
- NetApp ONTAP는 AWS, Azure, Google Cloud와 같은 주요 클라우드 서비스 공급자가 NAS 및 오브젝트 스토리지를 기본 지원합니다. 이러한 광범위한 호환성은 원활한 통합을 보장하므로 고객 데이터 이동성, 글로벌 액세스 가능성, 재해 복구, 동적 확장성 및 고성능을 지원합니다.
- NetApp의 강력한 데이터 관리 기능을 사용하면 데이터가 잠재적 위험과 위협으로부터 잘 보호되므로 안심하고 사용할 수 있습니다. NetApp은 데이터 보안의 우선순위를 정하고 고객이 중요한 정보의 안전성과 무결성에 대해 안심할 수 있도록 합니다.

벡터 데이터베이스

벡터 데이터베이스

벡터 데이터베이스는 머신 러닝 모델의 임베딩을 사용하여 비정형 데이터를 처리, 인덱싱 및 검색할 수 있도록 설계된 특수한 유형의 데이터베이스입니다. 데이터를 전통적인 표 형식으로 구성하는 대신 데이터를 벡터 임베딩이라고도 하는 고차원 벡터로 정렬합니다. 이 고유한 구조를 통해 데이터베이스에서 복잡한 다차원 데이터를 보다 효율적이고 정확하게 처리할 수 있습니다.

벡터 데이터베이스의 주요 기능 중 하나는 생성 AI를 사용하여 분석을 수행하는 것입니다. 여기에는 데이터베이스가 주어진 입력과 같은 데이터 포인트를 식별하는 유사성 검색, 규범과 크게 벗어나는 데이터 지점을 찾을 수 있는 이상 징후 감지 등이 포함됩니다.

또한 벡터 데이터베이스는 임시 데이터 또는 타임 스탬프 데이터를 처리하는 데 적합합니다. 이 유형의 데이터는 지정된 IT 시스템 내의 다른 모든 이벤트와 관련하여 '무슨' 및 언제 발생했는지에 대한 정보를 순서대로 제공합니다. 이러한 임시 데이터 처리 및 분석 기능으로 인해 벡터 데이터베이스는 시간에 따른 이벤트를 이해해야 하는 응용 프로그램에 특히 유용합니다.

ML 및 **AI**용 벡터 데이터베이스의 이점**:**

- 고차원 검색: 벡터 데이터베이스는 AI 및 ML 응용 프로그램에서 주로 생성되는 고차원 데이터를 관리하고 검색하는 데 탁월합니다.
- 확장성: 대량의 데이터를 처리할 수 있도록 효율적으로 확장하여 AI 및 ML 프로젝트의 성장과 확장을 지원합니다.
- 유연성: 벡터 데이터베이스는 높은 수준의 유연성을 제공하여 다양한 데이터 유형과 구조를 수용합니다.
- 성능: AI 및 ML 운영의 속도와 효율성을 높이는 데 매우 중요한 고성능 데이터 관리 및 검색을 제공합니다.
- 사용자 지정 가능한 인덱싱: Vector 데이터베이스는 사용자 지정 가능한 인덱싱 옵션을 제공하여 특정 요구 사항에 따라 최적화된 데이터 구성 및 검색을 지원합니다.

벡터 데이터베이스 및 사용 사례

이 섹션에서는 다양한 벡터 데이터베이스와 해당 사용 사례 세부 정보를 제공합니다.

Faiss와 **Scann**

벡터 검색의 영역에서 중요한 도구로 사용되는 라이브러리입니다. 이러한 라이브러리는 벡터 데이터를 관리하고 검색하는 데 중요한 기능을 제공하므로 데이터 관리의 전문 영역에 매우 유용한 리소스가 됩니다.

Elasticsearch(Elasticsearch)

널리 사용되는 검색 및 분석 엔진이며, 최근 벡터 검색 기능을 통합했습니다. 이 새로운 기능은 벡터 데이터를 보다 효과적으로 처리하고 검색할 수 있도록 기능을 향상시킵니다.

파인콘주식회사

고유한 기능 집합을 가진 강력한 벡터 데이터베이스입니다. 인덱싱 기능에서 고밀도 벡터와 희소 벡터를 모두 지원하므로 유연성과 적응성이 향상됩니다. 주요 장점 중 하나는 기존 검색 방법을 AI 기반 고밀도 벡터 검색과 결합하여 두 환경의 장점을 모두 활용하는 하이브리드 검색 방식을 만드는 능력에 있습니다.

주로 클라우드 기반의 Pinecone은 머신 러닝 애플리케이션용으로 설계되었으며 GCP, AWS, Open AI, GPT-3, GPT-3.5, GPT-4, catgut Plus, Elasticsearch, 하이스택, 설명합니다. Pinecone은 폐쇄형 소스 플랫폼이며 SaaS(Software as a Service) 오퍼링으로 사용할 수 있다는 점에 유의해야 합니다.

Pinecone은 고급 기능을 고려할 때 고도의 검색 및 하이브리드 검색 기능을 효과적으로 활용하여 위협을 감지하고 이에 대응할 수 있는 사이버 보안 산업에 특히 적합합니다.

채도

이 벡터 데이터베이스는 4가지 주요 기능을 가진 Core-API를 가지고 있으며, 그 중 하나는 메모리 내 문서 벡터 저장소를 포함합니다. 또한 Face Transformers 라이브러리를 활용하여 문서를 벡터화하여 기능과 다기능성을 향상시킵니다.
Chroma는 클라우드 및 온프레미스 모두에서 작동하도록 설계되어 사용자 요구에 따라 유연성을 제공합니다. 특히 오디오 관련 응용 프로그램에서 뛰어난 성능을 발휘하므로 오디오 기반 검색 엔진, 음악 추천 시스템 및 기타 오디오 관련 사용 사례에 적합합니다.

웨이비에이트

사용자가 내장 모듈이나 맞춤형 모듈을 사용하여 콘텐츠를 벡터화할 수 있는 다목적 벡터 데이터베이스로서, 특정 요구 사항에 따라 유연성을 제공합니다. 완전 관리형 솔루션과 자체 호스팅 솔루션을 모두 제공하여 다양한 구축 환경에 적합합니다.

Weaviate의 주요 특징 중 하나는 벡터와 객체를 모두 저장하여 데이터 처리 기능을 향상시키는 기능입니다. ERP 시스템의 의미론적 검색 및 데이터 분류 등 다양한 응용 프로그램에 널리 사용됩니다. 전자 상거래 부문에서는 검색 및 추천 엔진을 지원합니다. 또한 이미지 검색, 이상 징후 감지, 자동 데이터 조화 및 사이버 보안 위협 분석에 사용되어 여러 도메인에서 다양한 기능을 제공합니다.

레드입니다

Redis는 고속 인메모리 스토리지로 잘 알려진 고성능 벡터 데이터베이스로, 읽기-쓰기 작업에 대한 짧은 대기 시간을 제공합니다. 따라서 신속한 데이터 액세스가 필요한 추천 시스템, 검색 엔진 및 데이터 분석 애플리케이션에 매우 적합합니다.

Redis는 목록, 집합 및 정렬된 집합을 포함하여 벡터에 대한 다양한 데이터 구조를 지원합니다. 또한 벡터 간의 거리를 계산하거나 교차점과 결합을 찾는 등의 벡터 연산을 제공합니다. 이러한 기능은 유사성 검색, 클러스터링 및 콘텐츠 기반 추천 시스템에 특히 유용합니다.

확장성과 가용성 측면에서 Redis는 처리량이 많은 워크로드를 처리하는 데 탁월한 성능을 발휘하며 데이터 복제를 제공합니다. 또한 기존의 관계형 데이터베이스(RDBMS)를 비롯한 다른 데이터 유형과도 원활하게 통합됩니다. Redis에는 실시간 업데이트를 위한 게시/구독(Pub/Sub) 기능이 포함되어 있어 실시간 벡터 관리에 유용합니다. 또한 Redis는 가볍고 사용이 간편하여 벡터 데이터를 관리하기 위한 사용자 친화적인 솔루션입니다.

밀버스주식회사

MongoDB와 같이 문서 저장소와 같은 API를 제공하는 다기능 벡터 데이터베이스입니다. 이 제품은 다양한 데이터 유형을 지원하므로 데이터 과학 및 머신 러닝 분야에서 널리 사용되고 있습니다.

Milvus의 고유한 기능 중 하나는 다중 벡터화 기능으로, 사용자는 런타임에 검색에 사용할 벡터 유형을 지정할 수 있습니다. 또한 Faiss와 같은 다른 라이브러리 위에 있는 라이브러리인 Knowwhere를 활용하여 쿼리와 벡터 검색 알고리즘 간의 통신을 관리합니다.

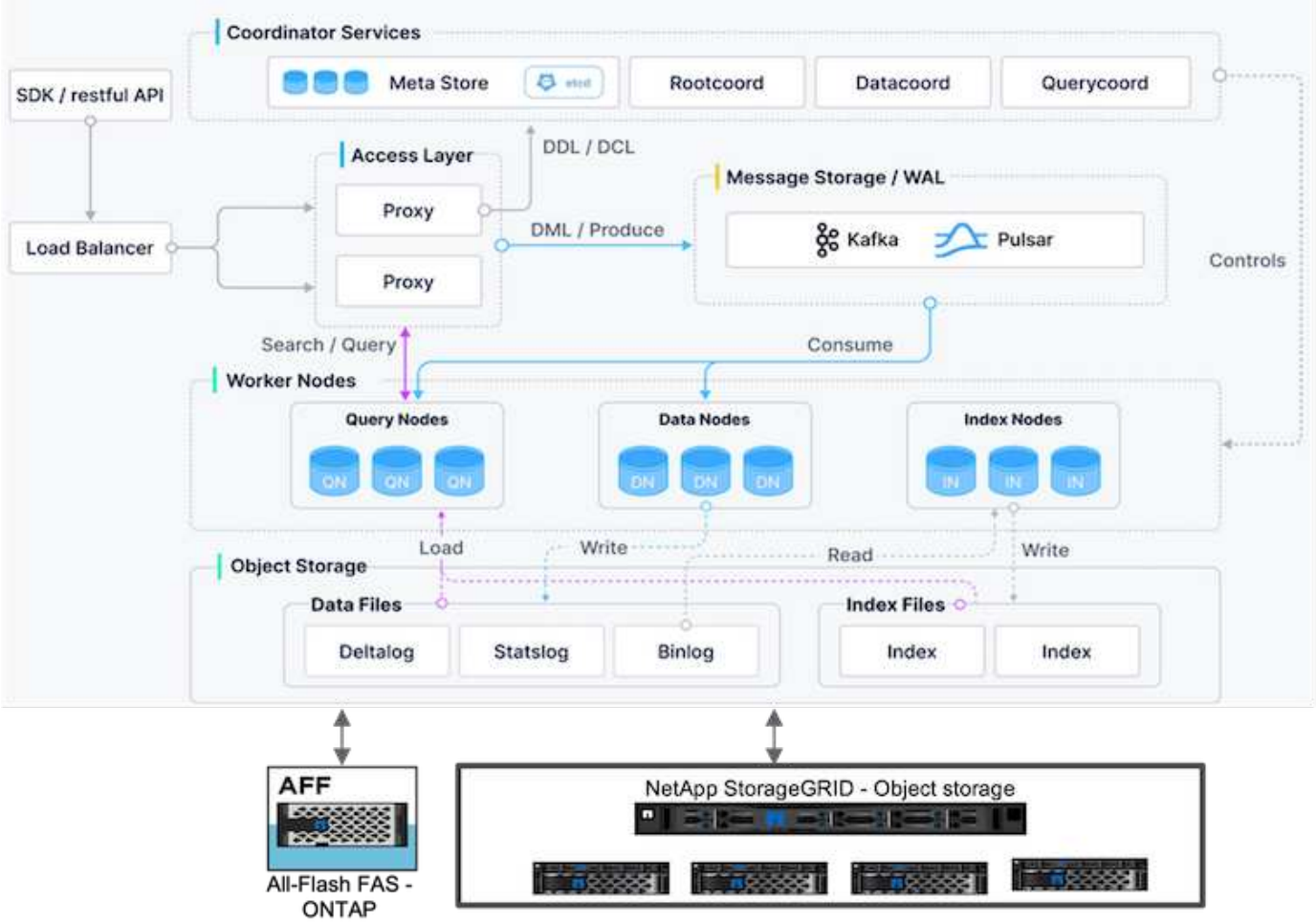
Milvus는 PyTorch 및 TensorFlow와의 호환성 덕분에 머신 러닝 워크플로우와 원활하게 통합됩니다. 따라서 전자 상거래, 이미지 및 비디오 분석, 개체 인식, 이미지 유사성 검색, 콘텐츠 기반 이미지 검색 등 다양한 응용 프로그램을 위한 탁월한 도구가 됩니다. 자연어 처리의 영역에서 Milvus는 문서 클러스터링, 의미론적 검색 및 질문 응답 시스템에 사용됩니다.

이 솔루션의 경우 솔루션 검증을 위해 milvus를 선택했습니다. 연주를 위해, 우리는 밀버스와 postgres (pgvecto.rs)를 모두 사용했습니다.

이 솔루션으로 **milvus**를 선택한 이유는 무엇입니까**?**

- 오픈 소스: Milvus는 오픈 소스 벡터 데이터베이스로, 커뮤니티 중심의 개발 및 개선을 촉진합니다.
- AI 통합: Similarity 검색 및 AI 애플리케이션을 내장하여 벡터 데이터베이스 기능을 개선합니다.
- 대용량 처리: Milvus는 DNN(Deep Neural Networks) 및 ML(Machine Learning) 모델에 의해 생성된 10억 개 이상의 임베디드 벡터를 저장, 인덱싱 및 관리할 수 있는 능력을 갖추고 있습니다.
- 사용자 친화적: 설치가 1분 이내에 완료되므로 사용이 간편합니다. Milvus는 또한 다양한 프로그래밍 언어에 대한 SDK를 제공합니다.
- 속도: 빠른 검색 속도를 제공하며 다른 제품보다 최대 10배 더 빠릅니다.
- 확장성 및 가용성: Milvus는 확장성이 뛰어나며 필요에 따라 스케일업 및 스케일아웃이 가능합니다.
- 다양한 기능: 다양한 데이터 유형, 속성 필터링, UDF(User-Defined Function) 지원, 구성 가능한 일관성 수준 및 이동 시간을 지원하므로 다양한 애플리케이션을 위한 다양한 도구가 될 수 있습니다.

Milvus 아키텍처 개요



이 섹션에서는 Milvus 아키텍처에 사용되는 더 높은 레버 구성 요소 및 서비스를 제공합니다.
* 액세스 계층 – 상태 비저장 프록시의 그룹으로 구성되어 있으며 시스템의 프런트 레이어와 사용자에게 엔드포인트의 역할을 합니다.
* 코디네이터 서비스 – 작업을 작업자 노드에 할당하고 시스템의 두뇌 역할을 수행합니다. 루트 코드, 데이터 코드 및 쿼리 코드의 세 가지 코디네이터 유형이 있습니다.
* 작업자 노드: 코디네이터 서비스의 지침을 따르고 사용자 트리거 DML/DDL commands.it 에는 쿼리 노드, 데이터 노드 및 인덱스 노드와 같은 세 가지 유형의 작업자 노드가 있습니다.
* 스토리지: 데이터 지속성을 책임집니다. 메타 스토리지, 로그 브로커 및 오브젝트 스토리지로 구성됩니다. ONTAP 및 StorageGRID와 같은 NetApp 스토리지는 Milvus에 고객 데이터와 벡터 데이터베이스 데이터를 위한 오브젝트 스토리지 및 파일 기반 스토리지를 제공합니다.

기술 요구 사항

기술 요구 사항

아래 설명된 하드웨어 및 소프트웨어 구성은 성능을 제외하고 본 문서에서 수행된 대부분의 검증에 사용되었습니다. 이러한 구성은 환경을 설정하는 데 도움이 되는 지침으로 사용됩니다. 그러나 특정 구성 요소는 개별 고객 요구 사항에 따라 다를 수 있습니다.

하드웨어 요구 사항

하드웨어	세부 정보
NetApp AFF 스토리지 어레이 HA 쌍	* A800 * ONTAP 9.14.1 * 48 x 3.49TB SSD-NVM * 두 개의 유연한 그룹 볼륨: 메타데이터 및 데이터. * 메타데이터 NFS 볼륨에는 250GB의 12개의 영구 볼륨이 있습니다. * 데이터는 ONTAP NAS S3 볼륨입니다
Fujitsu Primergy RX2540 M4 6개	* 64개의 CPU * 인텔® 제온® 골드 6142 CPU @ 2.60GHz * 256 GM 물리적 메모리 * 100GbE 네트워크 포트 1개
네트워킹	100GbE
StorageGRID	* SG100 1개, SGF6024 3개 * 3 x 24 x 7.68TB

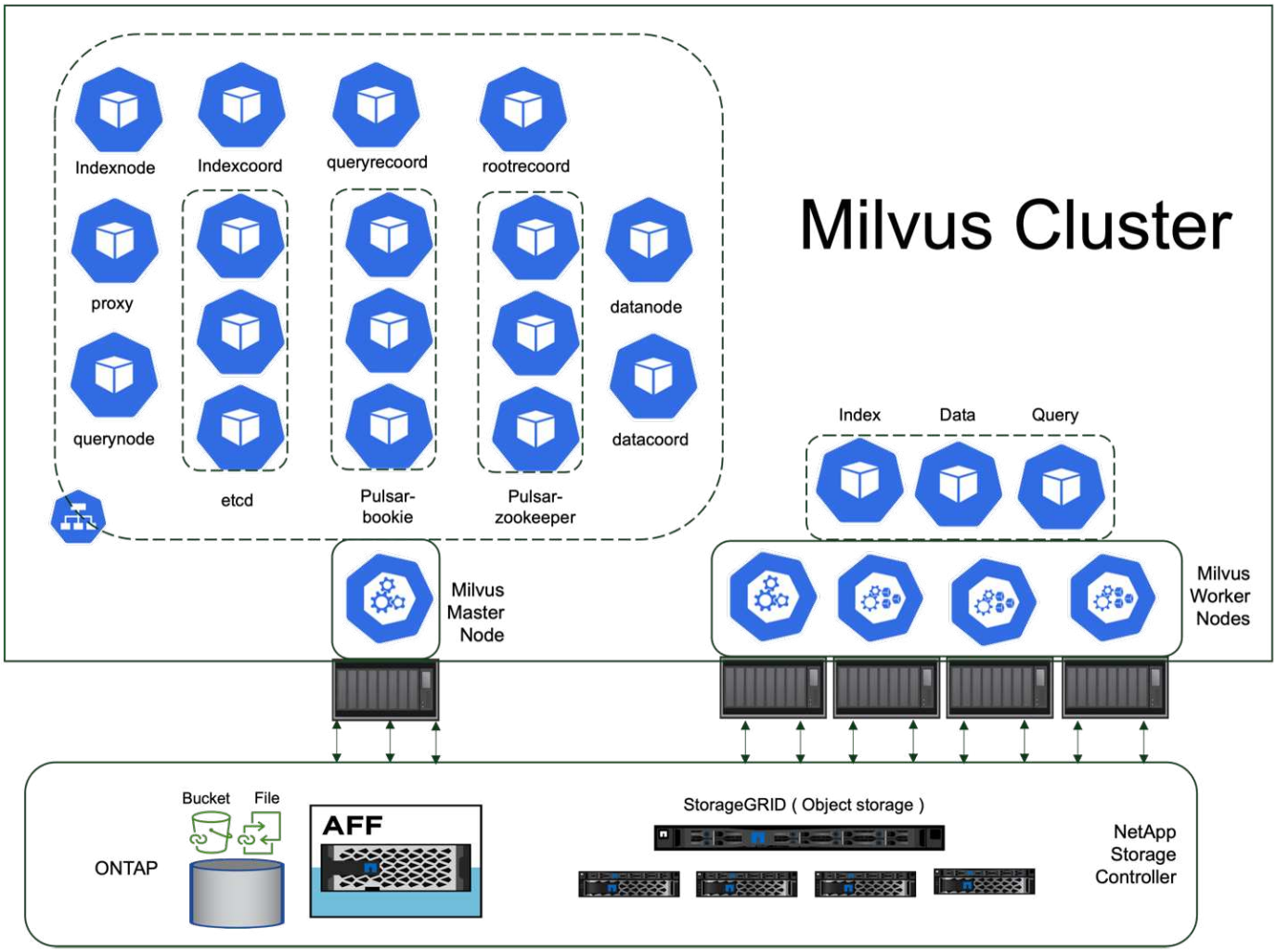
소프트웨어 요구 사항

소프트웨어	세부 정보
Milvus 클러스터	* 차트 - milvus - 4.1.11. * APP 버전 – 2.3.4 * 부킹기, zookeeper, Pulsar, etcd, proxy, querynode, 작업자
쿠버네티스	* 5노드 K8s 클러스터 *마스터 노드 1개 및 작업자 노드 4개 * 버전 – 1.7.2
파이썬	*3.10.12.

구현 절차

구현 절차

이 배포 섹션에서는 아래와 같이 랩 설정을 위해 Kubernetes와 함께 milvus 벡터 데이터베이스를 사용했습니다.



NetApp 스토리지는 클러스터에서 고객 데이터를 유지하고 클러스터 데이터를 Milvus로 유지할 수 있도록 스토리지를 제공합니다.

NetApp 스토리지 설정 **– ONTAP**

- 스토리지 시스템 초기화
- 스토리지 가상 시스템(SVM) 생성
- 논리적 네트워크 인터페이스 할당
- NFS, S3 구성 및 라이센스

NFS(네트워크 파일 시스템)에 대해 아래 단계를 따르십시오.

1. NFSv4용 FlexGroup 볼륨을 생성합니다. 이 검증을 위한 설정에서는 48개의 SSD, 컨트롤러의 루트 볼륨 전용 SSD 1개, NFSv4에 분산된 SSD 47개]]를 사용했습니다. FlexGroup 볼륨에 대한 NFS 엑스포트 정책이 Kubernetes(K8) 노드 네트워크에 대한 읽기/쓰기 권한이 있는지 확인하십시오. 이러한 권한이 없는 경우 K8s 노드 네트워크에 대한 읽기/쓰기(RW) 권한을 부여합니다.
2. 모든 K8 노드에서 폴더를 생성하고 각 K8 노드의 LIF(논리 인터페이스)를 통해 FlexGroup 볼륨을 이 폴더에 마운트합니다.

NAS S3(Network Attached Storage Simple Storage Service)에 대해 아래 단계를 따르십시오.

1. NFS에 대한 FlexGroup 볼륨을 생성합니다.
2. "vserver object-store-server create" 명령을 사용하여 HTTP가 활성화되고 admin 상태가 'up'으로 설정된 Object-store-server를 설정하십시오. HTTPS를 활성화하고 사용자 지정 수신기 포트를 설정할 수 있습니다.
3. "vserver object-store-server user create-user <username>" 명령을 사용하여 object-store-server 사용자를 생성합니다.
4. 액세스 키와 비밀 키를 얻으려면 "set diag; vserver object-store-server user show -user <username>" 명령을 실행할 수 있습니다. 그러나 앞으로 이러한 키는 사용자 생성 프로세스 중에 제공되거나 REST API 호출을 사용하여 검색할 수 있습니다.
5. 2단계에서 만든 사용자를 사용하여 object-store-server 그룹을 설정하고 액세스 권한을 부여합니다. 이 예에서는 "FullAccess"를 제공합니다.
6. 유형을 "nas"로 설정하고 NFSv3 볼륨에 대한 경로를 제공하여 NAS 버킷을 생성합니다. 또한 이 용도로 S3 버킷을 사용할 수도 있습니다.

NetApp 스토리지 설정 **– StorageGRID**

1. StorageGRID 소프트웨어를 설치합니다.
2. 테넌트와 버킷을 생성합니다.
3. 필요한 권한이 있는 사용자를 생성합니다.

자세한 내용은 에서 확인하십시오 https://docs.netapp.com/us-en/storagegrid-116/primer/index.html

솔루션 개요

당사는 5가지 주요 영역에 중점을 두고 포괄적인 솔루션 검증을 수행했으며, 자세한 내용은 아래에 나와 있습니다. 각 섹션에서는 고객이 직면한 과제, NetApp에서 제공하는 솔루션, 그리고 고객이 누릴 수 있는 이점에 대해 알아봅니다.

1. "온프레미스에서 Kubernetes를 사용하여 Milvus 클러스터 설정"
   스토리지와 컴퓨팅, 효과적인 인프라 관리 및 데이터 관리에 대해 독립적으로 확장해야 하는 고객의 당면 과제 이 섹션에서는 클러스터 데이터 및 고객 데이터 모두에 NetApp 스토리지 컨트롤러를 활용하여 Kubernetes에 Milvus 클러스터를 설치하는 프로세스를 자세히 설명합니다.
2. "NetApp ONTAP용 Amazon FSxN을 사용하는 Milvus – 파일 및 오브젝트 이중화"
   이 섹션에서는 클라우드에 벡터 데이터베이스를 배포해야 하는 이유와 Docker 컨테이너 내의 Amazon FSxN for NetApp ONTAP에 벡터 데이터베이스(milvus 독립형)를 배포하는 단계를 설명합니다.
3. "NetApp SnapCenter를 사용한 벡터 데이터베이스 보호."
   이 섹션에서는 SnapCenter가 ONTAP에 상주하는 벡터 데이터베이스 데이터와 Milvus 데이터를 보호하는 방법을 알아봅니다. 이 예에서는 고객 데이터에 NFS ONTAP 볼륨(vol1)에서 파생된 NAS 버킷(milvusdbvol1)을 활용하고 Milvus 클러스터 구성 데이터에 별도의 NFS 볼륨(vectordbpv)을 사용했습니다.
4. "NetApp SnapMirror를 사용한 재해 복구"
   이 섹션에서는 벡터 데이터베이스에 대한 DR(재해 복구)의 중요성과 NetApp 재해 복구 제품 SnapMirror가 벡터 데이터베이스에 DR 솔루션을 제공하는 방법에 대해 설명합니다.
5. "성능 검증"
   이 섹션에서는 Milvus 및 pgvecto.RS와 같은 벡터 데이터베이스의 성능 검증을 자세히 살펴보고, LLM 수명주기 내의 RAG 및 추론 워크로드를 지원하는 I/O 프로필 및 NetApp 스토리지 컨트롤러 같은 스토리지 성능 특성에 초점을 맞추고자 합니다. 이들 데이터베이스가 ONTAP 스토리지 솔루션과 결합될 때 성능 차별화 요소를 평가하고 식별합니다. 당사의 분석은 QPS(초당 처리된 쿼리 수)와 같은 주요 성능 지표를 기반으로 합니다.

온프레미스에서 **Kubernetes**를 사용한 **Milvus** 클러스터 설정

온프레미스에서 **Kubernetes**를 사용하여 **Milvus** 클러스터 설정

고객의 당면 과제: 스토리지와 컴퓨팅, 효과적인 인프라 관리 및 데이터 관리
Kubernetes와 벡터 데이터베이스는 함께 대규모 데이터 작업 관리를 위한 강력하고 확장 가능한 솔루션을 형성합니다. Kubernetes는 리소스를 최적화하고 컨테이너를 관리하는 한편, 벡터 데이터베이스는 고밀도 데이터 및 유사성 검색을 효율적으로 처리합니다. 이 결합을 사용하면 대규모 데이터 세트에서 복잡한 쿼리를 신속하게 처리할 수 있고 증가하는 데이터 볼륨으로 원활하게 확장할 수 있으므로 빅데이터 애플리케이션과 AI 워크로드에 이상적입니다.

1. 이 섹션에서는 클러스터 데이터 및 고객 데이터 모두에 NetApp 스토리지 컨트롤러를 활용하여 Kubernetes에 Milvus 클러스터를 설치하는 프로세스를 자세히 설명합니다.
2. Milvus 클러스터를 설치하려면 다양한 Milvus 클러스터 구성 요소의 데이터를 저장하기 위해 영구 볼륨(PVS)이 필요합니다. 이러한 구성 요소에는 etcd(3개 인스턴스), Pulsar-bookie-journal(3개 인스턴스), Pulsar-bookie-ledgers(3개 인스턴스), Pulsar-zookeeper-data(3개 인스턴스) 등이 있습니다.

> ⓘ Milvus 클러스터에서는 Milvus 클러스터의 안정적인 저장 및 메시지 스트림의 게시/구독을 지원하는 기본 엔진에 Pulsar 또는 Kafka를 사용할 수 있습니다. NFS를 사용하는 Kafka의 경우 NetApp은 ONTAP 9.12.1 이상에서 성능을 향상했으며, RHEL 8.7 또는 9.1 이상에 포함된 NFSv4.1 및 Linux 변경 사항과 함께 이러한 기능 향상을 통해 NFS에서 Kafka를 실행할 때 발생할 수 있는 "silly rename" 문제를 해결합니다. NetApp NFS 솔루션과 함께 Kafka 실행 주제에 관한 더 자세한 정보는 을 참조하십시오. https://docs.netapp.com/us-en/netapp-solutions/data-analytics/kafka-nfs-introduction.html.

3. 우리는 NetApp ONTAP에서 단일 NFS 볼륨을 생성하고 각각 250GB의 스토리지를 가진 12개의 영구 볼륨을 생성했습니다. 스토리지 크기는 클러스터 크기에 따라 다를 수 있습니다. 예를 들어 각 PV에 50GB가 있는 또 다른 클러스터가 있습니다. 자세한 내용은 아래 PV YAML 파일 중 하나를 참조하십시오. 총 12개의 파일이 있습니다. 각 파일에서 storageClassName은 'default'로 설정되고 스토리지 및 경로는 각 PV에 대해 고유합니다.

```
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. 각 PV YAML 파일에 대해 'kubectl apply' 명령을 실행하여 영구 볼륨을 생성한 다음 'kubectl get PV'를 사용하여 생성된 볼륨을 확인합니다

```
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. 고객 데이터 저장을 위해 Milvus는 MinIO, Azure Blob 및 S3와 같은 오브젝트 스토리지 솔루션을 지원합니다. 이 가이드에서는 S3를 활용합니다. 다음 단계는 ONTAP S3 및 StorageGRID 오브젝트 저장소 모두에 적용됩니다. 우리는 Helm을 사용하여 Milvus 클러스터를 구축합니다. Milvus 다운로드 위치에서 구성 파일 Values.yaml을 다운로드합니다. 이 문서에서 사용한 .yaml 값 파일은 부록을 참조하십시오.
6. 로그, etcd, zookeeper 및 bookkeeper를 포함하여 각 섹션에서 'storageClass'가 'default'로 설정되어 있는지 확인합니다.
7. MinIO 섹션에서 MinIO를 비활성화합니다.
8. ONTAP 또는 StorageGRID 오브젝트 스토리지에서 NAS 버킷을 생성하고 오브젝트 스토리지 자격 증명을 사용하여 외부 S3에 포함합니다.

```
###################################
# 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. Milvus 클러스터를 생성하기 전에 PVC(PersistentVolumeClaim)에 기존 리소스가 없는지 확인하십시오.

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

10. Helm 및 value.yaml 구성 파일을 사용하여 Milvus 클러스터를 설치하고 시작합니다.

```
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. PersistentVolumeClaims(PVC)의 상태를 확인합니다.

```
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. Pod의 상태를 확인합니다.

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

Pod 상태가 '실행 중'이고 예상대로 작동하는지 확인하십시오

13. Milvus 및 NetApp 오브젝트 스토리지에서 데이터 쓰기 및 읽기를 테스트합니다.
    - "prepare_data_netapp_new.py" Python 프로그램을 사용하여 데이터를 작성합니다.

```
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:~#
```

◦ "verify_data_netapp.py" Python 파일을 사용하여 데이터를 읽습니다.

```
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}}]}
```

NetApp 스토리지 컨트롤러를 사용하여 Kubernetes에 Milvus 클러스터를 구축하여 설명한 대로 위의 검증을 기반으로 Kubernetes와 벡터 데이터베이스를 통합하면 대규모 데이터 운영 관리를 위한 강력하고 확장 가능하며 효율적인 솔루션을 고객에게 제공합니다. 이 설정은 고객이 높은 차원 데이터를 처리하고 복잡한 쿼리를 신속하고 효율적으로 실행할 수 있도록 하여 빅 데이터 애플리케이션 및 AI 워크로드에 이상적인 솔루션입니다. 다양한 클러스터 구성 요소에 PVS(영구 볼륨)를 사용하고 NetApp ONTAP에서 단일 NFS 볼륨을 생성하면 최적의 리소스 활용도와 데이터 관리가 보장됩니다. PersistentVolumeClaims(PVC) 및 Pod의 상태를 확인하고 데이터 쓰기 및 읽기 테스트를 통해 고객은 안정적이고 일관된 데이터 작업을 보장할 수 있습니다. ONTAP 또는 StorageGRID 오브젝트 스토리지를 고객 데이터에 사용하면 데이터 접근성과 보안이

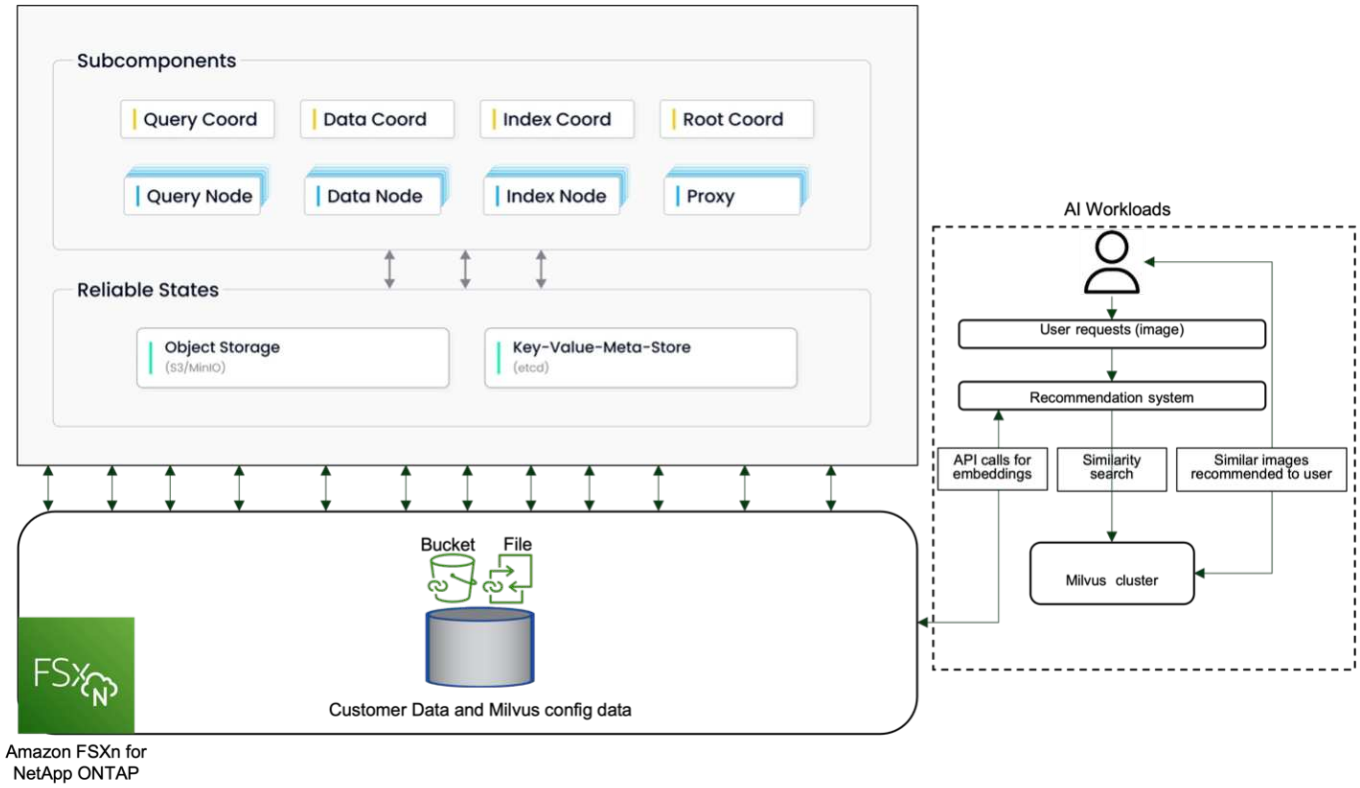
더욱 강화됩니다. 이 설정을 통해 고객은 증가하는 데이터 요구사항에 맞춰 원활하게 확장할 수 있는 복원력을 갖춘 고성능 데이터 관리 솔루션을 확보할 수 있습니다.

NetApp ONTAP 파일 및 객체 이중성을 위한 Amazon FSxN을 사용하는 Milvus

NetApp ONTAP용 Amazon FSxN을 사용하는 Milvus – 파일 및 오브젝트 이중화

이 섹션에서는 클라우드에 벡터 데이터베이스를 배포해야 하는 이유와 Docker 컨테이너 내의 Amazon FSxN for NetApp ONTAP에 벡터 데이터베이스(milvus 독립형)를 배포하는 단계를 설명합니다.

클라우드에 벡터 데이터베이스를 배포하면 특히 높은 차원 데이터를 처리하고 유사성 검색을 실행해야 하는 애플리케이션의 경우 몇 가지 중요한 이점을 얻을 수 있습니다. 첫째, 클라우드 기반 배포는 확장성을 제공하므로 증가하는 데이터 볼륨 및 쿼리 로드에 맞게 리소스를 쉽게 조정할 수 있습니다. 이를 통해 데이터베이스에서 고성능을 유지하면서 증가하는 수요를 효율적으로 처리할 수 있습니다. 둘째, 클라우드 배포는 데이터를 서로 다른 지리적 위치에 복제할 수 있어 고가용성 및 재해 복구를 제공하며, 데이터 손실 위험을 최소화하고 예기치 않은 이벤트에도 지속적인 서비스를 보장합니다. 셋째, 사용하는 리소스에 대해서만 비용을 지불하므로 비용 효율성이 향상되고 필요에 따라 확장 또는 축소할 수 있으므로 하드웨어에 대한 초기 투자 비용을 크게 줄일 필요가 없습니다. 마지막으로 클라우드에 벡터 데이터베이스를 배포하면 어디서나 데이터에 액세스하고 공유할 수 있으므로 팀 기반 작업과 데이터 기반 의사 결정을 용이하게 할 수 있으므로 협업을 개선할 수 있습니다.
이 검증에 사용된 NetApp ONTAP용 Amazon FSxN과 함께 Milvus 독립 실행형 아키텍처를 확인하십시오.



1. NetApp ONTAP용 Amazon FSxN 인스턴스를 생성하고 VPC, VPC 보안 그룹 및 서브넷의 세부 정보를 기록합니다. 이 정보는 EC2 인스턴스를 생성할 때 필요합니다. 자세한 내용은 여기 에서 확인할 수 있습니다. - https://us-east-1.console.aws.amazon.com/fsx/home?region=us-east-1#file-system-create
2. EC2 인스턴스를 생성하여 VPC, 보안 그룹 및 서브넷이 NetApp ONTAP용 Amazon FSxN 인스턴스의 인스턴스와 일치하는지 확인합니다.
3. 'apt-get install nfs-common' 명령을 사용하여 nfs-common을 설치하고 'SUDO apt-get update'를 사용하여 패키지 정보를 업데이트합니다.

4. 마운트 폴더를 생성하고 NetApp ONTAP용 Amazon FSxN을 마운트합니다.

```
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. 'apt-get install'을 사용하여 Docker 및 Docker Compose를 설치합니다.
6. Milvus 웹 사이트에서 다운로드할 수 있는 docker-composition.yaml 파일을 기반으로 Milvus 클러스터를 설정합니다.

```
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. docker-configuration.yml 파일의 'volumes' 섹션에서 NetApp NFS 마운트 지점을 해당 Milvus 컨테이너 경로, 특히 etcd, minio 및 standalone.Check에 매핑합니다 "부록 D: docker-composition.yml" YML의 변화에 대한 자세한 내용은
8. 마운트된 폴더 및 파일을 확인합니다.

```
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. docker-composition.yml 파일이 들어 있는 디렉토리에서 'docker-composure up-d'를 실행합니다.
10. Milvus 컨테이너의 상태를 확인합니다.

```
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. 벡터 데이터베이스의 읽기 및 쓰기 기능을 검증하기 위해 Amazon FSxN for NetApp ONTAP에서는 Python Milvus SDK와 PyMilvus의 샘플 프로그램을 사용했습니다. 'apt-get install python3-numpy python3-pip'을 사용하여 필요한 패키지를 설치하고 'pip3 install pymilvus'를 사용하여 PyMilvus를 설치합니다.
12. 벡터 데이터베이스에서 NetApp ONTAP용 Amazon FSxN의 데이터 쓰기 및 읽기 작업을 검증합니다.

```
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. verify_data_netapp.py 스크립트를 사용하여 읽기 작업을 확인합니다.

```
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. 고객이 AI 워크로드용 S3 프로토콜을 통해 벡터 데이터베이스에서 테스트된 NFS 데이터에 액세스하려는 경우 간단한 Python 프로그램을 사용하여 검증을 받을 수 있습니다. 예를 들어 이 섹션의 시작 부분에 있는 그림에서 언급했듯이 다른 응용 프로그램에서 이미지를 비슷한 방식으로 검색할 수 있습니다.

```
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#
```

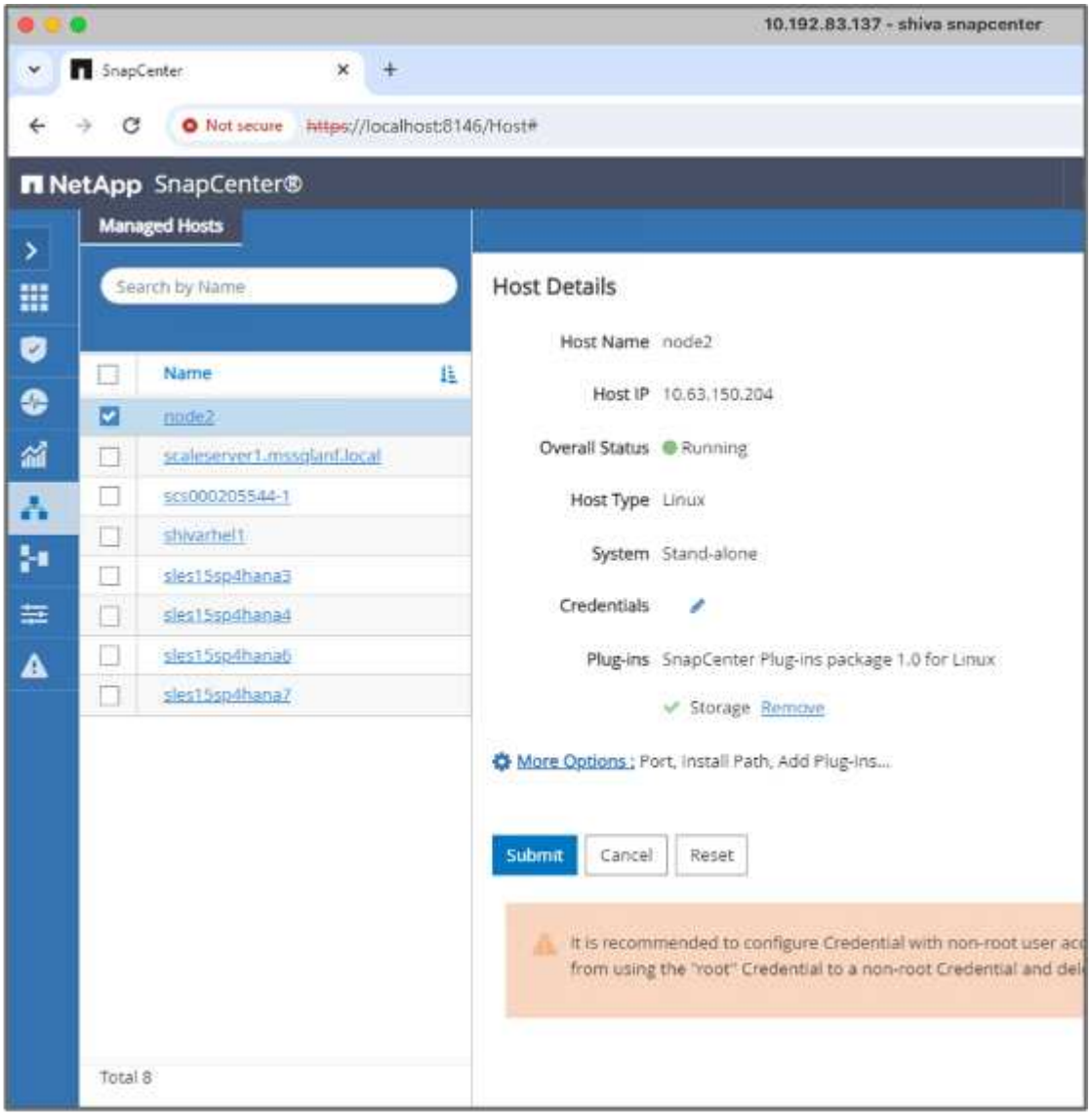
이 섹션에서는 NetApp ONTAP 데이터 스토리지용 Amazon의 NetApp FSxN을 활용하여 고객이 Docker 컨테이너 내에서 독립 실행형 Milvus 설정을 구축 및 운영하는 방법을 효과적으로 보여줍니다. 이 설치를 통해 고객은 Docker 컨테이너의 확장 가능하고 효율적인 환경 내에서 벡터 데이터베이스의 기능을 활용하여 차원 높은 데이터를 처리하고 복잡한 쿼리를 실행할 수 있습니다. NetApp ONTAP 인스턴스용 Amazon FSxN을 생성하고 EC2 인스턴스를 일치시킴으로써 최적의 리소스 활용도와 데이터 관리를 보장할 수 있습니다. 벡터 데이터베이스에서 FSxN의 데이터 쓰기 및 읽기 작업을 성공적으로 검증함으로써 고객은 안정적이고 일관된 데이터 작업을 보장할 수 있습니다. 또한, S3 프로토콜을 통해 AI 워크로드의 데이터를 나열(읽기) 기능은 향상된 데이터 접근성을 제공합니다. 따라서 이 포괄적인 프로세스는 고객에게 NetApp ONTAP용 Amazon의 FSxN의 기능을 활용하여 대규모 데이터 운영을 관리할 수 있는 강력하고 효율적인 솔루션을 제공합니다.

SnapCenter를 사용한 벡터 데이터베이스 보호

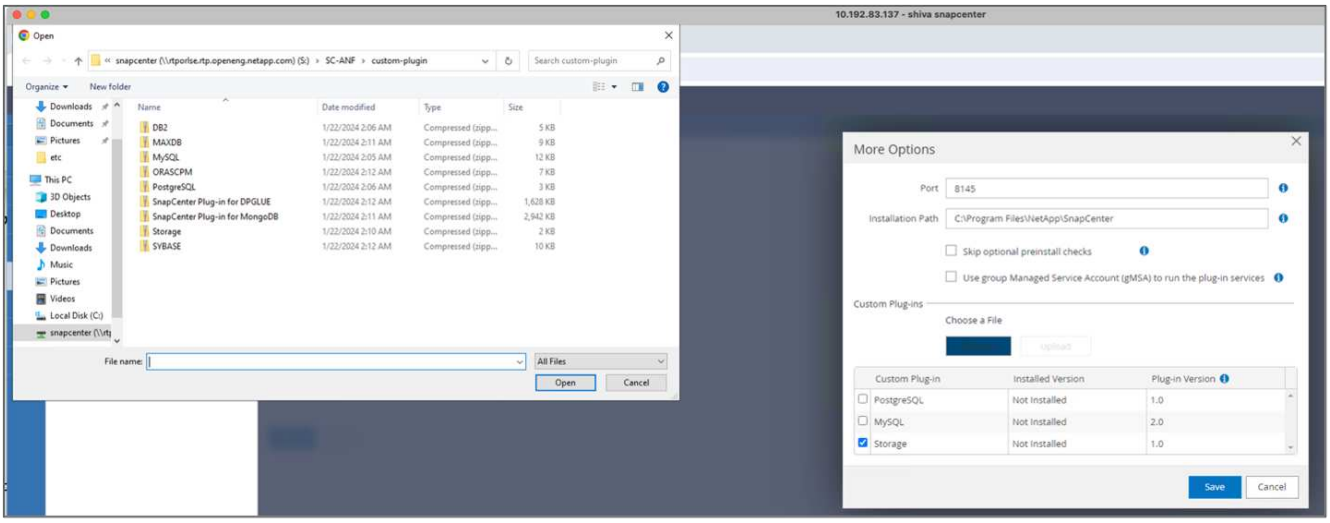
NetApp SnapCenter를 사용한 벡터 데이터베이스 보호**.**

예를 들어 영화 제작 업계에서 고객은 비디오 및 오디오 파일과 같은 중요한 내장 데이터를 보유하는 경우가 많습니다. 하드 드라이브 장애와 같은 문제로 인해 이 데이터가 손실되면 운영에 큰 영향을 미칠 수 있으며 수백만 달러 규모의 벤처 기업을 위험에 빠뜨릴 수 있습니다. 귀중한 콘텐츠가 유실되어 상당한 혼란과 재정적 손실이 발생하는 경우가 있었습니다. 따라서 이 업계에서는 필수 데이터의 보안과 무결성을 보장하는 것이 매우 중요합니다.
이 섹션에서는 SnapCenter가 ONTAP에 상주하는 벡터 데이터베이스 데이터와 Milvus 데이터를 보호하는 방법을 알아봅니다. 이 예에서는 고객 데이터에 NFS ONTAP 볼륨(vol1)에서 파생된 NAS 버킷(milvusdbvol1)을 활용하고 Milvus 클러스터 구성 데이터에 별도의 NFS 볼륨(vectordbpv)을 사용했습니다. 를 확인하십시오 "여기" SnapCenter 백업 워크플로우의 경우

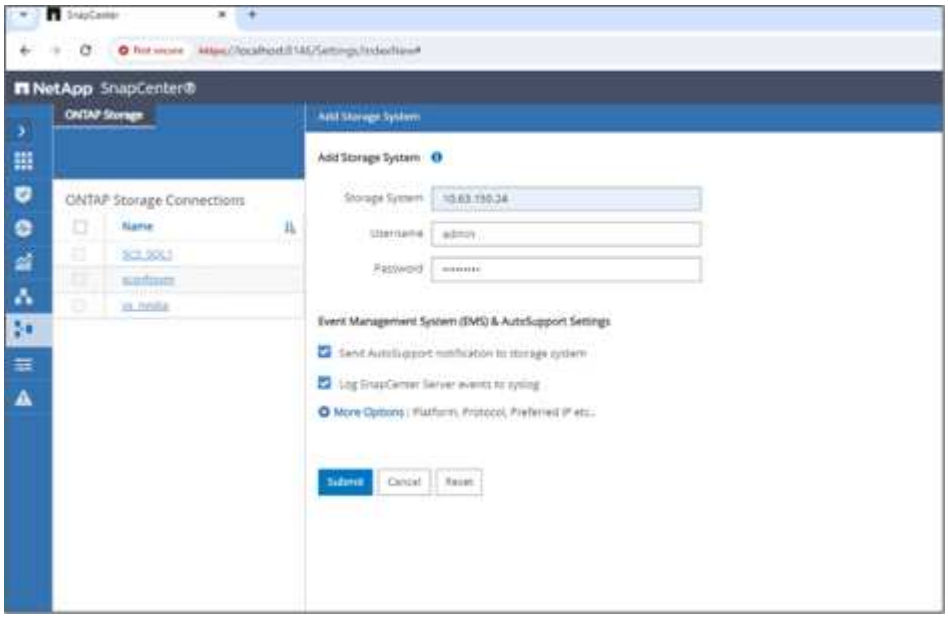
1. SnapCenter 명령을 실행하는 데 사용할 호스트를 설정합니다.



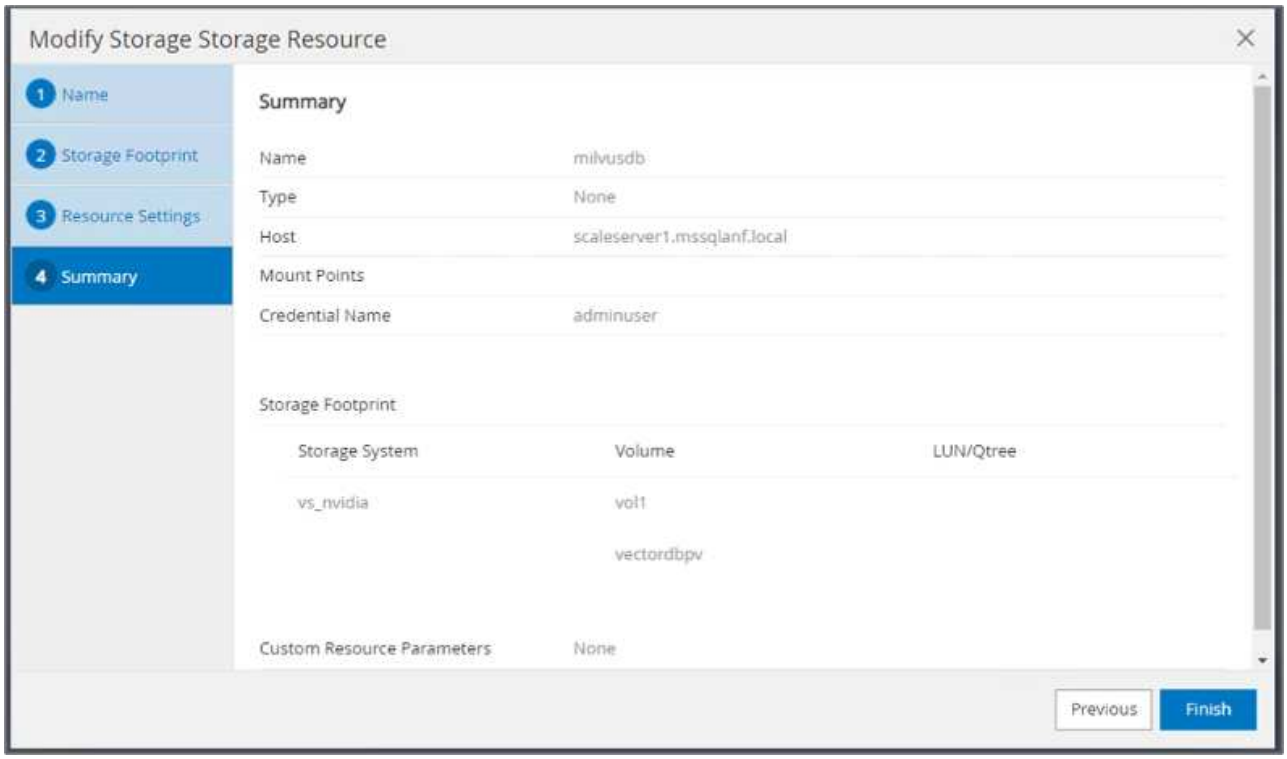
2. 스토리지 플러그인을 설치 및 구성합니다. 추가된 호스트에서 "추가 옵션"을 선택합니다. 에서 다운로드한 스토리지 플러그인을 찾아 선택합니다 "NetApp 자동화 스토어". 플러그인을 설치하고 구성을 저장합니다.



3. 스토리지 시스템 및 볼륨 설정: "Storage System"에 스토리지 시스템을 추가하고 SVM(Storage Virtual Machine)을 선택합니다. 이 예에서는 "vs_nvidia"를 선택했습니다.



4. 백업 정책 및 사용자 지정 스냅샷 이름을 통합하여 벡터 데이터베이스에 대한 리소스를 설정합니다.
   - 기본값으로 정합성 보장 그룹 백업을 설정하고 파일 시스템 정합성 보장 없이 SnapCenter를 설정합니다.
   - Storage Footprint 섹션에서 벡터 데이터베이스 고객 데이터 및 Milvus 클러스터 데이터와 연결된 볼륨을 선택합니다. 이 예에서는 "vol1"과 "vectordbpv"입니다.
   - 정책을 사용하여 벡터 데이터베이스 보호에 대한 정책을 만들고 벡터 데이터베이스 리소스를 보호합니다.



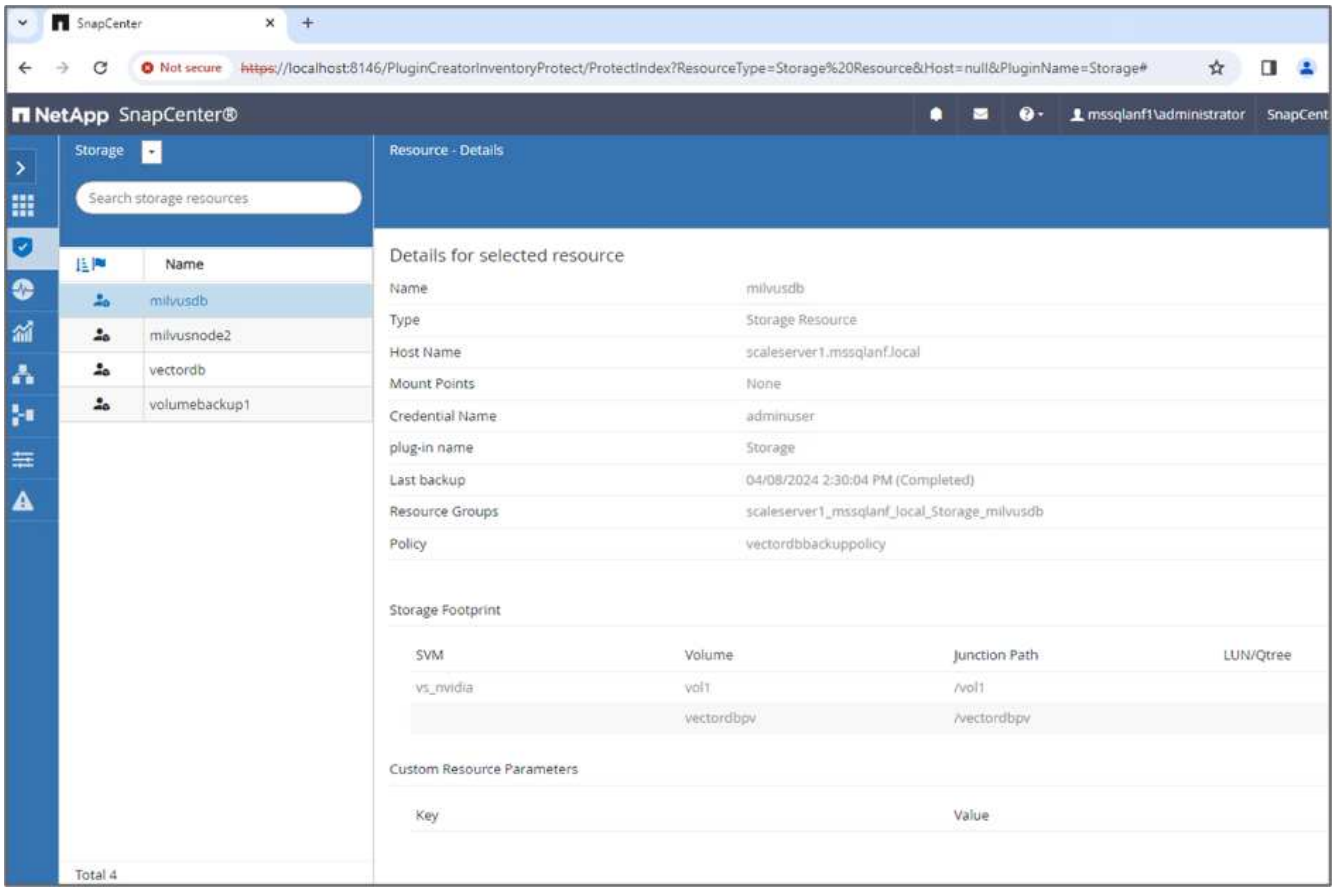
5. Python 스크립트를 사용하여 S3 NAS 버킷에 데이터를 삽입합니다. 이 경우 Milvus에서 제공하는 백업 스크립트, 즉 'prepare_data_netapp.py'를 수정하고 'sync' 명령을 실행하여 운영 체제에서 데이터를 플러시했습니다.

```
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. S3 NAS 버킷에서 데이터를 확인합니다. 이 예에서는 타임 스탬프 '2024-04-08 21:22'를 가진 파일이 'prepare_data_netapp.py' 스크립트에 의해 생성되었습니다.

```
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. 'milvusdb' 리소스의 CG(정합성 보장 그룹) 스냅샷을 사용하여 백업을 시작합니다

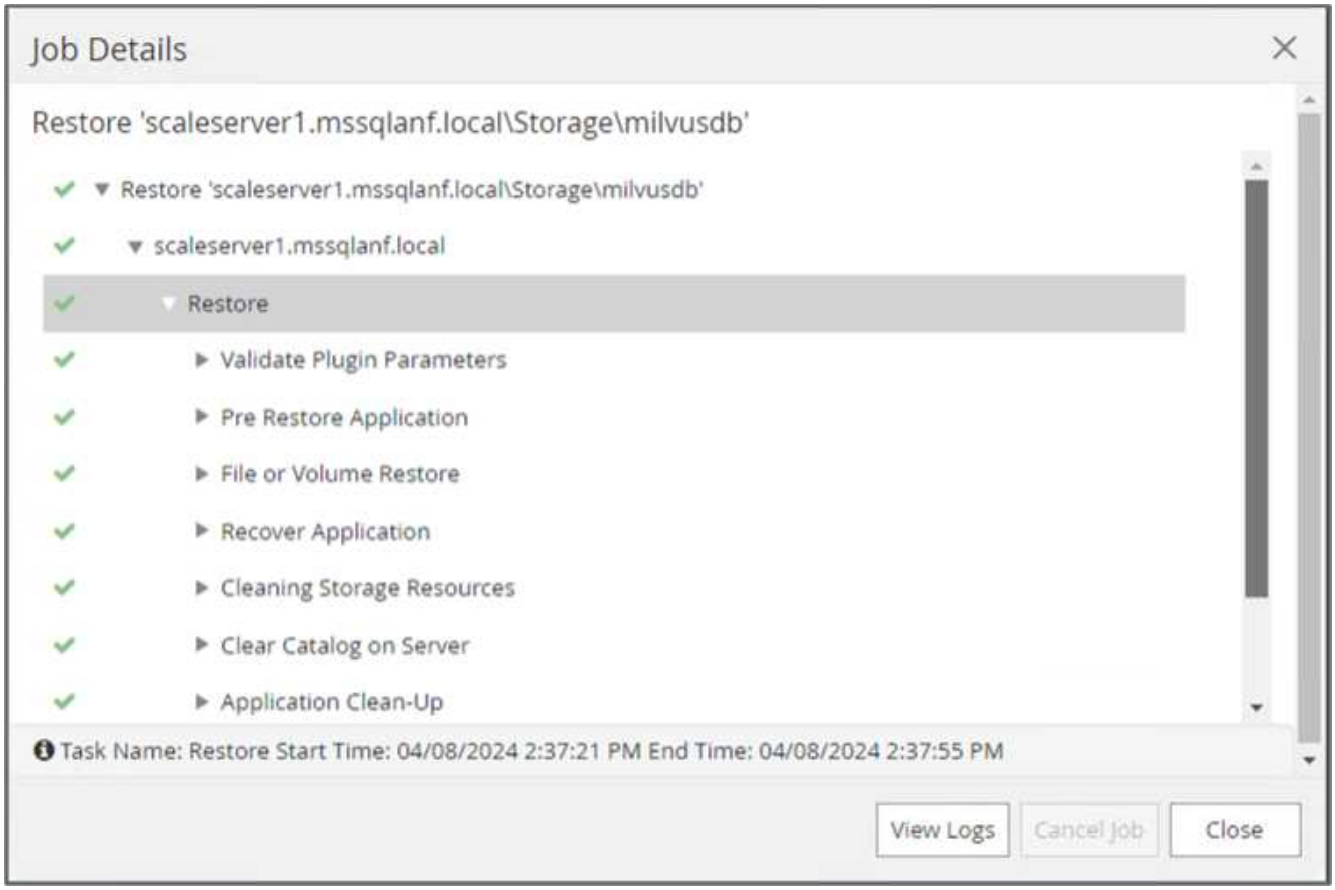


8. 백업 기능을 테스트하기 위해 백업 프로세스 후에 새 테이블을 추가하거나 NFS(S3 NAS 버킷)에서 일부 데이터를 제거했습니다.

   이 테스트를 위해 누군가가 백업 후 불필요한 새로운 수집 또는 부적절한 수집을 생성한 시나리오를 생각해 보십시오. 이 경우 새 컬렉션이 추가되기 전에 벡터 데이터베이스를 해당 상태로 되돌려야 합니다. 예를 들어, 'hello_milvus_netapp_sc_testnew', 'hello_milvus_netapp_sc_testnew2'와 같은 새 컬렉션이 삽입되었습니다.

```
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. 이전 스냅샷에서 S3 NAS 버킷의 전체 복구를 실행합니다.



10. Python 스크립트를 사용하여 'hello_milvus_netapp_SC_test' 및 'hello_milvus_netapp_SC_test2' 컬렉션에서 데이터를 확인하십시오.

```
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. 불필요하거나 부적절한 수집이 데이터베이스에 더 이상 존재하지 않는지 확인합니다.

```
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:~#
```

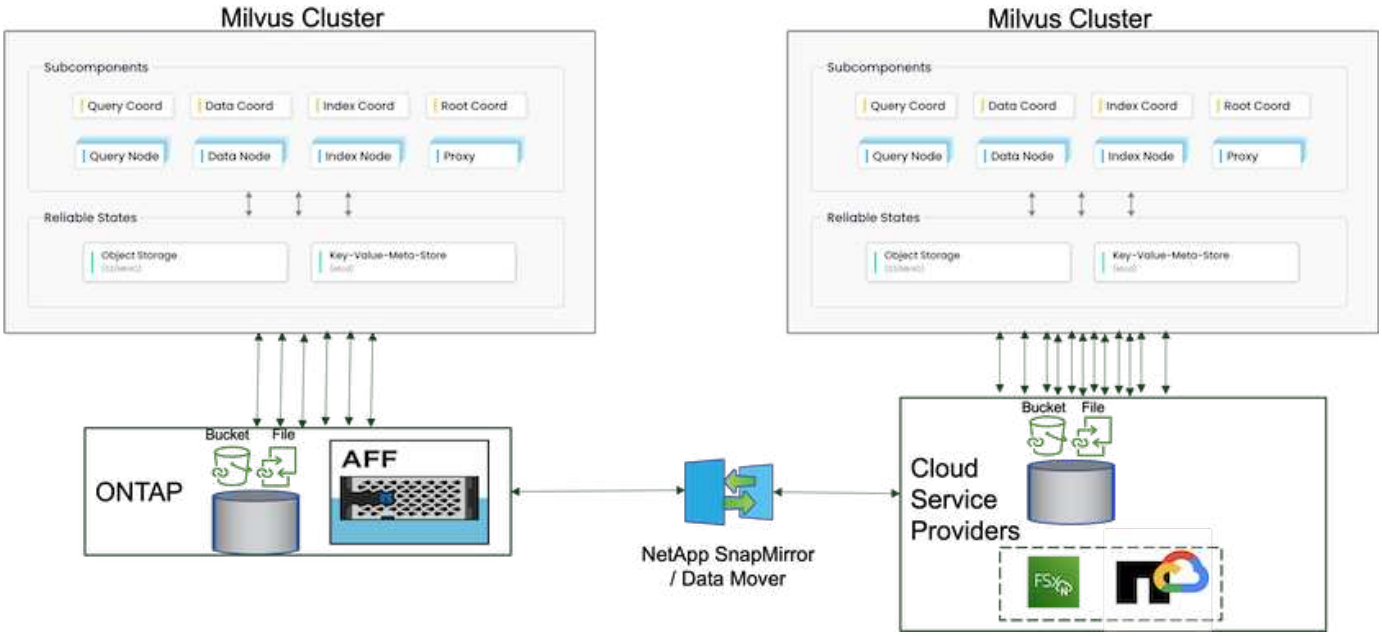
결론적으로, NetApp의 SnapCenter를 사용하여 벡터 데이터베이스 데이터와 ONTAP에 상주하는 Milvus 데이터를 보호하면 고객에게 상당한 혜택을 제공할 수 있으며, 특히 데이터 무결성이 영화 제작과 같이 중요한 산업에서 그 효과를 볼 수 있습니다. SnapCenter는 일관된 백업을 생성하고 전체 데이터 복원을 수행할 수 있으므로 내장 비디오 및 오디오 파일과 같은 중요한 데이터가 하드 드라이브 장애 또는 기타 문제로 인한 손실을 방지합니다. 이는 운영 중단을 방지할 뿐만 아니라 상당한 재정적 손실을 방지합니다.

이 섹션에서는 호스트 설정, 스토리지 플러그인 설치 및 구성, 사용자 지정 스냅샷 이름을 사용하여 벡터 데이터베이스에 대한 리소스 생성을 포함하여 ONTAP에 있는 데이터를 보호하도록 SnapCenter를 구성하는 방법을 살펴보았습니다. 또한 정합성 보장 그룹 스냅샷을 사용하여 백업을 수행하고 S3 NAS 버킷에서 데이터를 확인하는 방법에 대해서도 설명했습니다.

또한 백업 후 불필요하거나 부적절한 수집이 생성된 시나리오를 시뮬레이션했습니다. 이 경우 SnapCenter는 이전 스냅샷에서 전체 복원을 수행할 수 있으므로 새 컬렉션을 추가하기 전의 상태로 벡터 데이터베이스를 되돌릴 수 있으므로 데이터베이스의 무결성을 유지할 수 있습니다. 데이터를 특정 시점으로 복원하는 이러한 기능은 고객이 데이터를 안전하게 보호할 수 있을 뿐만 아니라 올바르게 유지 관리한다는 확신을 바탕으로 고객에게 매우 중요합니다. 따라서 NetApp의 SnapCenter 제품은 데이터 보호 및 관리를 위한 강력하고 안정적인 솔루션을 고객에게 제공합니다.

NetApp SnapMirror를 사용한 재해 복구

NetApp SnapMirror를 사용한 재해 복구



재해 복구는 벡터 데이터베이스의 무결성과 가용성을 유지하는 데 매우 중요합니다. 특히, 높은 차원 데이터를 관리하고 복잡한 유사성 검색을 실행하는 데 그 역할을 합니다. 재해 복구 전략을 잘 계획하고 구현하면 하드웨어 장애, 자연 재해 또는 사이버 공격과 같은 예상치 못한 사고가 발생할 경우 데이터가 손실되거나 손상되지 않습니다. 이는 특히 데이터 손실 또는 손상이 중대한 운영 중단 및 재정적 손실을 초래할 수 있는 벡터 데이터베이스를 사용하는 애플리케이션에 중요합니다. 또한 강력한 재해 복구 계획을 통해 다운타임을 최소화하고 신속한 서비스 복원을 가능하게 하여 비즈니스 연속성을 보장합니다. 이 기능은 서로 다른 지리적 위치, 정기적인 백업 및 장애 조치 메커니즘에 걸쳐 NetApp 데이터 복제 제품 SnapMirror를 통해 구현됩니다. 따라서 재해 복구는 단순한 보호 수단이 아니라 책임감 있고 효율적인 벡터 데이터베이스 관리의 중요한 구성 요소입니다.

NetApp의 SnapMirror는 한 NetApp ONTAP 스토리지 컨트롤러에서 다른 컨트롤러로 데이터 복제를 제공하고, 이는 DR(재해 복구) 및 하이브리드 솔루션에 주로 사용됩니다. 벡터 데이터베이스의 컨텍스트에서 이 툴을 사용하면 온프레미스와 클라우드 환경 간에 데이터를 원활하게 전환할 수 있습니다. 이러한 전환은 데이터 변환 또는 애플리케이션 리팩토링 없이 수행되므로 여러 플랫폼에서 데이터 관리의 효율성과 유연성을 향상할 수 있습니다.

벡터 데이터베이스 시나리오의 NetApp 하이브리드 솔루션은 다음과 같은 점에서 더 많은 이점을 제공합니다.

1. 확장성: NetApp의 하이브리드 클라우드 솔루션은 고객의 요구사항에 따라 리소스를 확장할 수 있는 기능을 제공합니다. 온프레미스 리소스를 NetApp ONTAP용 Amazon FSxN, Google Cloud NetApp Volume(GCNV)과 같은 정기적인 예측 가능한 워크로드 및 클라우드 리소스를 활용하여 사용량이 가장 많은 시간 또는 예상치 못한 부하에 대응할 수 있습니다.
2. 비용 효율성: NetApp의 하이브리드 클라우드 모델을 활용하면 일반 워크로드에 온프레미스 리소스를 사용하고 필요할 때만 클라우드 리소스에 대한 비용을 지불함으로써 비용을 최적화할 수 있습니다. NetApp instaclustr 서비스 오퍼링을 사용하면 이 용량제 모델을 비용 효율적으로 사용할 수 있습니다. instaclustr는 온프레미스 및 주요 클라우드 서비스 공급자를 위해 지원과 상담을 제공합니다.
3. 유연성: NetApp의 하이브리드 클라우드를 사용하면 데이터를 처리할 위치를 유연하게 선택할 수 있습니다. 예를 들어, 클라우드에서 보다 강력한 하드웨어와 덜 집중적인 작업을 수행하는 복잡한 벡터 작업을 사내에서 수행하도록 선택할 수 있습니다.
4. 비즈니스 연속성: 재해 발생 시 NetApp 하이브리드 클라우드에 데이터를 저장하면 비즈니스 연속성을 보장할 수 있습니다. 사내 리소스에 영향을 미치는 경우 클라우드로 빠르게 전환할 수 있습니다. NetApp SnapMirror를 활용하여 사내에서 클라우드로 또는 그 반대로 데이터를 이동할 수 있습니다.
5. 혁신: NetApp의 하이브리드 클라우드 솔루션은 최첨단 클라우드 서비스 및 기술에 대한 액세스를 제공하여 더 빠르게 혁신을 지원할 수 있습니다. NetApp ONTAP, Azure NetApp Files, Google Cloud NetApp Volumes용

Amazon FSxN과 같은 클라우드의 NetApp 혁신은 클라우드 서비스 공급자의 혁신적인 제품과 선호하는 NAS입니다.

Vector 데이터베이스 성능 검증

성능 검증

성능 검증은 벡터 데이터베이스와 스토리지 시스템 모두에서 중요한 역할을 하며 최적의 운영과 효율적인 리소스 활용도를 보장하는 핵심 요소로 작용합니다. 높은 차원 데이터를 처리하고 유사성 검색을 실행하는 것으로 알려진 벡터 데이터베이스는 복잡한 쿼리를 신속하고 정확하게 처리하기 위해 높은 성능 수준을 유지해야 합니다. 성능 검증은 병목 현상을 식별하고 구성을 세밀하게 조정하고 시스템이 서비스 성능 저하 없이 예상 로드를 처리할 수 있도록 하는 데 도움이 됩니다. 마찬가지로 스토리지 시스템에서 성능 검증은 전체 시스템 성능에 영향을 줄 수 있는 지연 시간 문제나 병목 현상 없이 데이터를 효율적으로 저장하고 검색하는 데 필수적입니다. 또한 정보에 근거하여 스토리지 인프라의 필요한 업그레이드 또는 변경 결정을 내릴 수 있도록 지원합니다. 따라서 성능 검증은 시스템 관리의 중요한 측면으로, 높은 서비스 품질, 운영 효율성 및 전반적인 시스템 안정성을 유지하는 데 크게 기여합니다.

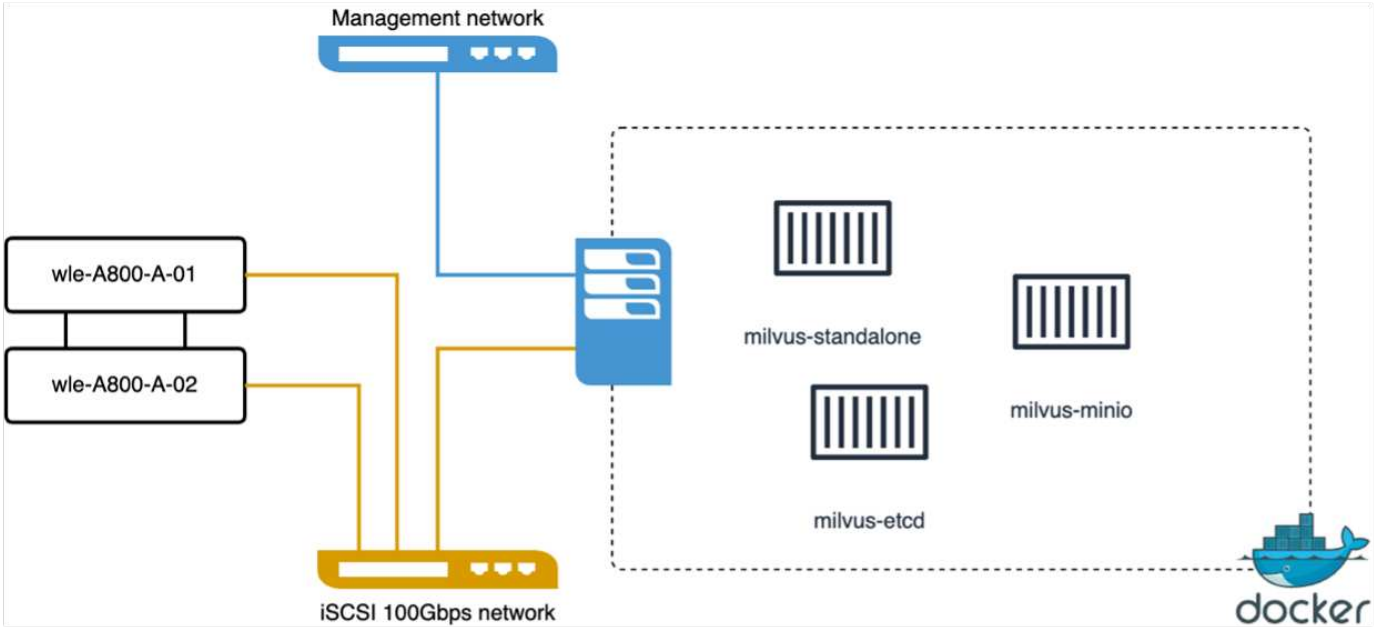
이 섹션에서는 Milvus 및 pgvecto.RS와 같은 벡터 데이터베이스의 성능 검증을 자세히 살펴보고, LLM 수명주기 내의 RAG 및 추론 워크로드를 지원하는 I/O 프로필 및 NetApp 스토리지 컨트롤러 같은 스토리지 성능 특성에 초점을 맞추고자 합니다. 이들 데이터베이스가 ONTAP 스토리지 솔루션과 결합될 때 성능 차별화 요소를 평가하고 식별합니다. 당사의 분석은 QPS(초당 처리된 쿼리 수)와 같은 주요 성능 지표를 기반으로 합니다.

아래 Milvus에 사용되는 방법론과 진행 상황을 확인하십시오.

세부 정보	**Milvus**(독립 실행형 및 클러스터)	**Postgres(pgvecto.rs)**
버전	2.3.2	0.2.0
파일 시스템	iSCSI LUN의 XFS	
워크로드 생성기	"벡터DB-벤치" –v0.0.5	
데이터 세트	LAION 데이터 세트 * 10,000,000개의 침구 * 768 치수 * ~ 300GB 데이터 세트 크기	

VectorDB - Milvus 독립 실행형 클러스터를 사용한 벤치

vectorDB-Bench를 사용하는 milvus 독립 실행형 클러스터에 대해 다음과 같은 성능 검증을 수행했습니다. milvus 독립 실행형 클러스터의 네트워크 및 서버 연결은 다음과 같습니다.



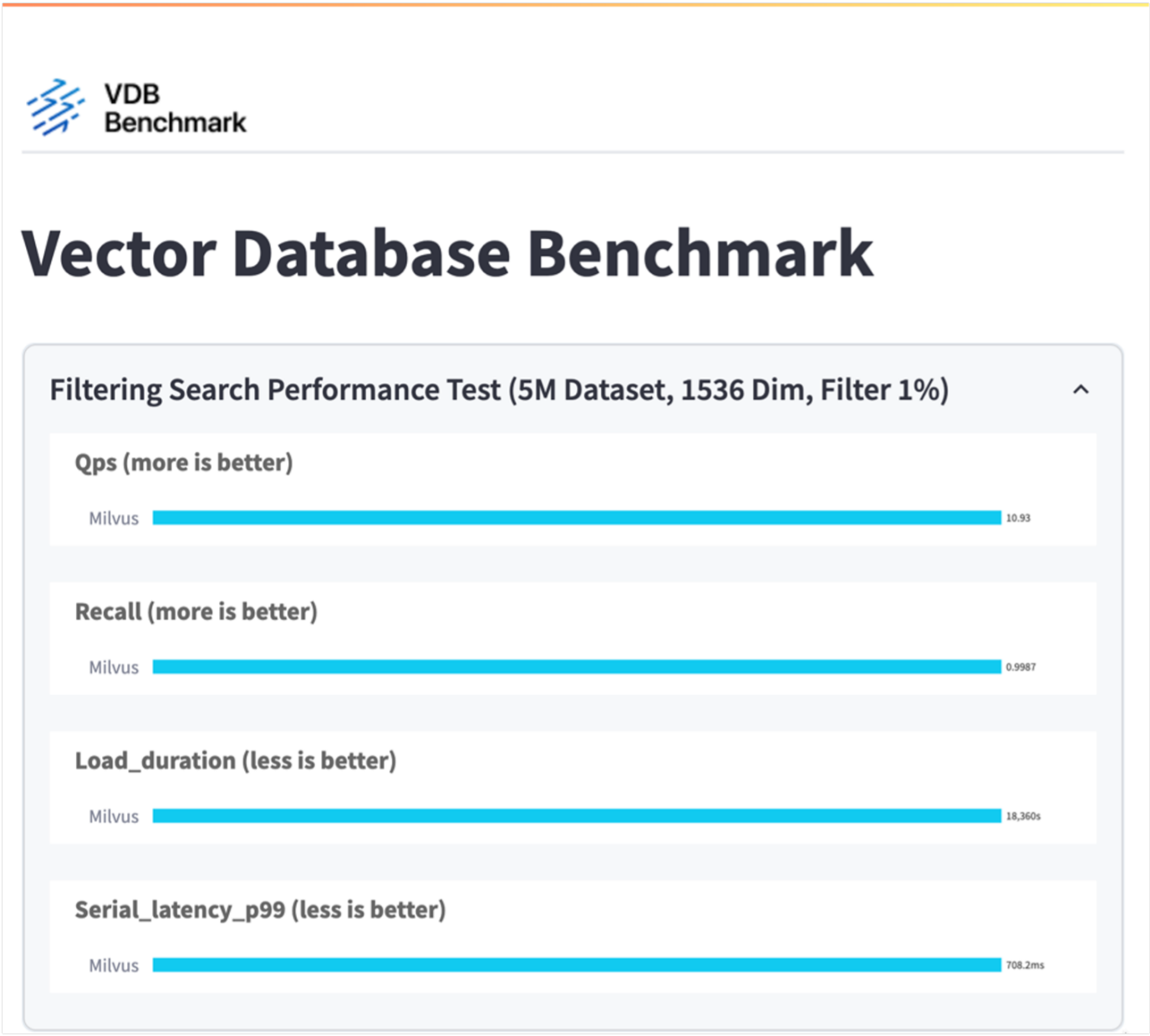
이 섹션에서는 Milvus 독립 실행형 데이터베이스 테스트의 관찰 결과 및 결과를 공유합니다.
. 이러한 테스트의 인덱스 유형으로 DiskANN을 선택했습니다.
. 약 100GB의 데이터 세트에 대한 인덱스를 수집, 최적화 및 생성하는 데 약 5시간이 걸렸습니다. 대부분의 기간 동안 20개의 코어(하이퍼 스레딩을 사용할 경우 40개의 vCPU와 동일)를 장착한 Milvus 서버는 최대 CPU 용량인 100%에서 작동하고 있었습니다. DiskANN은 특히 시스템 메모리 크기를 초과하는 대규모 데이터세트에 매우 중요합니다.
. 조회 단계에서 0.9987을 리콜한 상태에서 QPS(Queries per second)가 10.93인 것을 확인했습니다. 쿼리의 99번째 백분위수 지연 시간은 708.2밀리초로 측정되었습니다.

스토리지 관점에서 볼 때 데이터베이스는 수집, 삽입 후 최적화, 인덱스 생성 단계에서 약 1,000 ops/sec를 기록했습니다. 쿼리 단계에서는 32,000 ops/초가 필요했습니다

다음 섹션에서는 스토리지 성능 메트릭에 대해 설명합니다.

워크로드 단계	미터	값
데이터 수집 및 삽입 후 최적화	IOPS	1,000명 이하
	지연 시간	400개 이상의 활용
	워크로드	읽기/쓰기 조합, 주로 쓰기입니다
	IO 크기입니다	64KB
쿼리	IOPS	32,000에서 최고치
	지연 시간	400개 이상의 활용
	워크로드	100% 캐시된 읽기
	IO 크기입니다	주로 8KB

vectorDB-bench 결과는 다음과 같습니다.

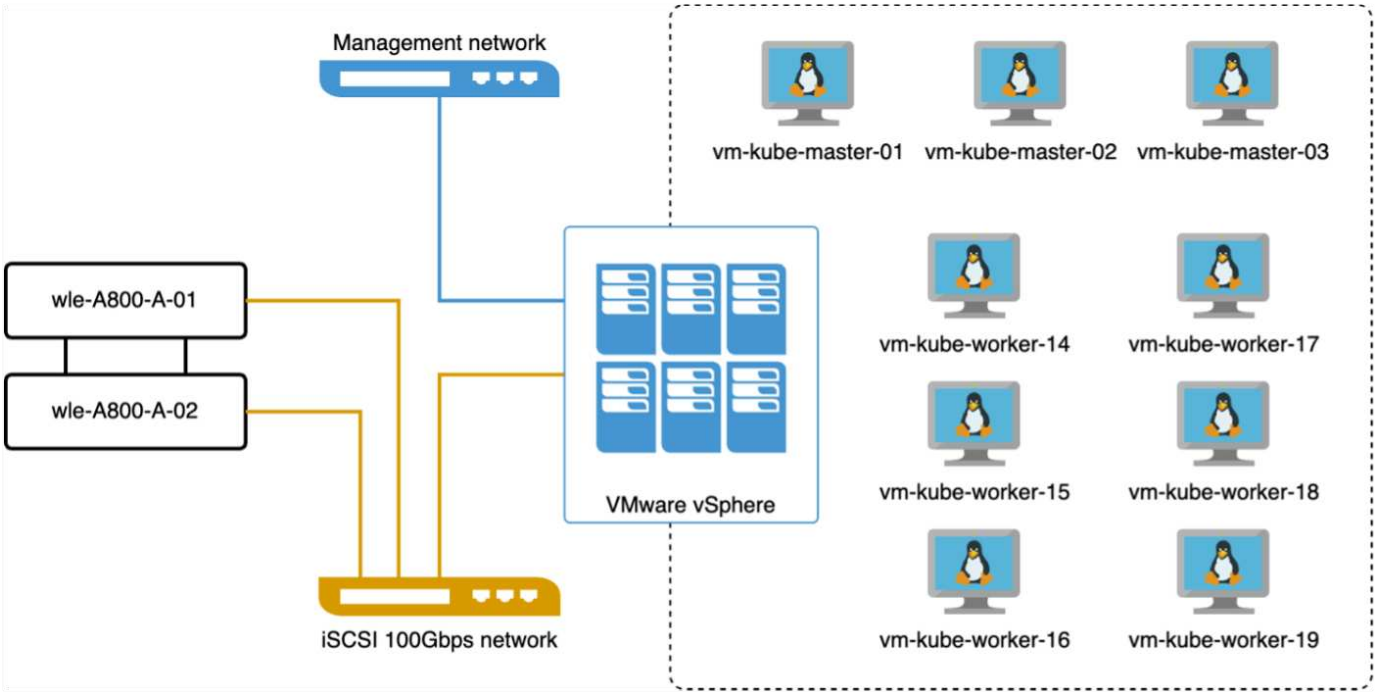


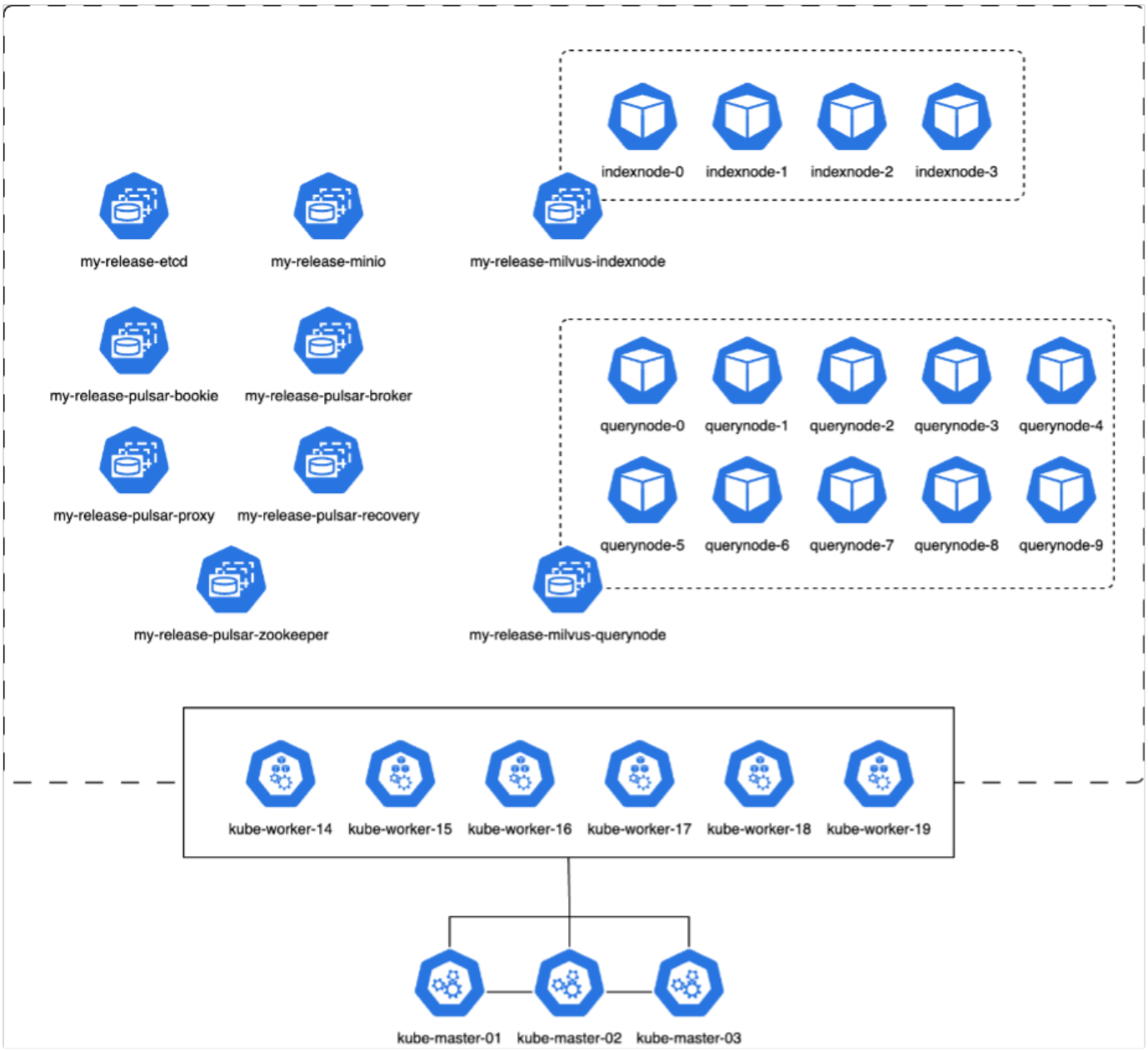
독립 실행형 Milvus 인스턴스의 성능 검증을 통해 현재 설정이 치수 1536의 5백만 벡터로 구성된 데이터 세트를 지원하기에는 불충분하다는 것을 알 수 있습니다. 스토리지에 적절한 리소스가 있으며 시스템에 병목 현상이 발생하지 않는 것으로 확인되었습니다.

VectorDB - 밀버스 클러스터를 사용한 벤치

이 섹션에서는 Kubernetes 환경 내에 Milvus 클러스터를 구축하는 방법에 대해 설명합니다. 이 Kubernetes 설정은 Kubernetes 마스터 및 작업자 노드를 호스팅하는 VMware vSphere 배포를 바탕으로 구성되었습니다.

VMware vSphere 및 Kubernetes 구축에 대한 자세한 내용은 다음 섹션에서 설명합니다.


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



이 섹션에서는 Milvus 데이터베이스 테스트에서 얻은 관찰 내용과 결과를 설명합니다.
*사용된 인덱스 유형은 DiskANN입니다.
*아래 표는 1536의 치수에서 5백만 벡터로 작업할 때 독립형 배포와 클러스터 배포를 비교한 것입니다. 우리는 데이터 수집 및 삽입 후 최적화에 걸리는 시간이 클러스터 구축에서 더 낮은 것으로 확인되었습니다. 독립 실행형 설정에 비해 클러스터 구축 시 쿼리의 99번째 백분위수 지연 시간이 6배 감소했습니다.
*클러스터 배포에서 QPS(Queries per Second) 비율이 높았지만 원하는 수준에서는 그렇지 않았습니다.

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%

아래 이미지는 스토리지 클러스터 지연 시간 및 총 IOPS(초당 입출력 작업 수)를 포함한 다양한 스토리지 메트릭을 보여줍니다.

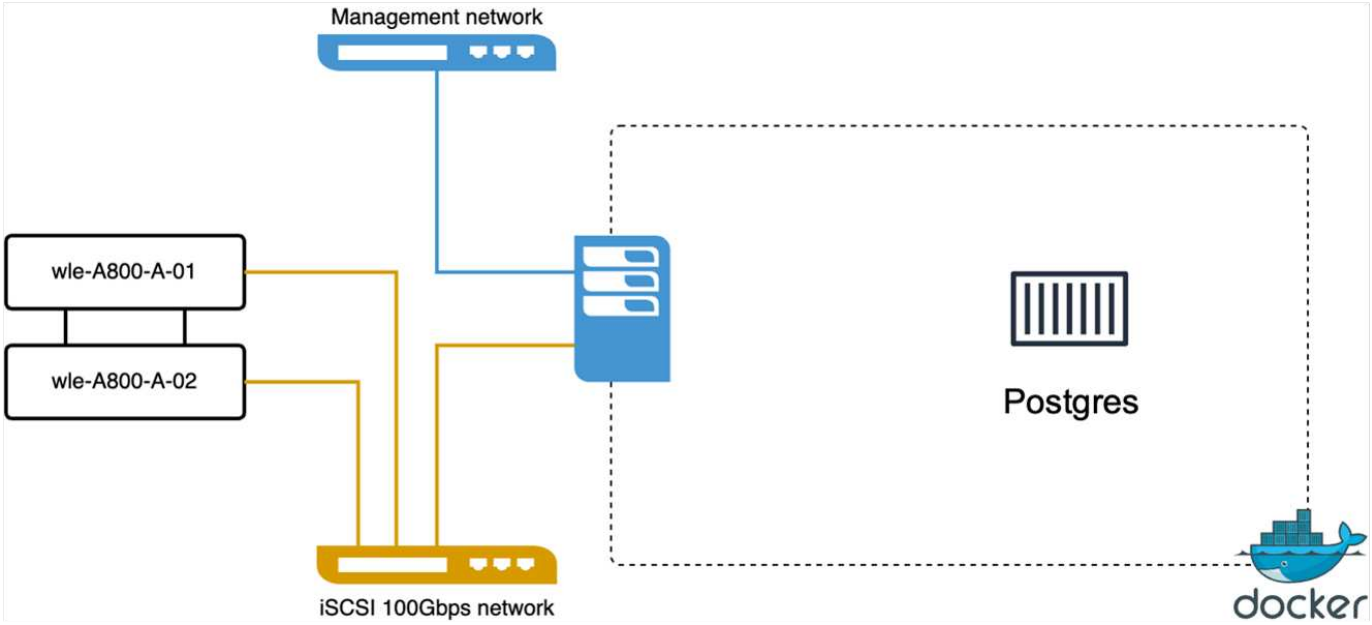
다음 섹션에서는 주요 스토리지 성능 메트릭에 대해 설명합니다.

워크로드 단계	미터	값
데이터 수집 및 삽입 후 최적화	IOPS	1,000명 이하
	지연 시간	400개 이상의 활용
	워크로드	읽기/쓰기 조합, 주로 쓰기입니다
	IO 크기입니다	64KB
쿼리	IOPS	147,000에서 최고점
	지연 시간	400개 이상의 활용
	워크로드	100% 캐시된 읽기
	IO 크기입니다	주로 8KB

독립 실행형 Milvus 및 Milvus 클러스터의 성능 검증을 토대로 스토리지 I/O 프로필에 대한 세부 정보를 제공합니다.
* 독립 실행형 및 클러스터 구축 모두에서 I/O 프로필이 일관되게 유지되는 것을 확인했습니다.
* 피크 IOPS에서 관찰된 차이는 클러스터 구축 시 클라이언트 수가 많기 때문입니다.

Postgres가 있는 **vectorDB-**벤치**(pgvecto.rs)**

VectorDB-Bench를 사용하여 PostgreSQL(pgvecto.rs)에 대해 다음 작업을 수행했습니다.
PostgreSQL(특히 pgvecto.rs)의 네트워크 및 서버 연결에 대한 세부 정보는 다음과 같습니다.

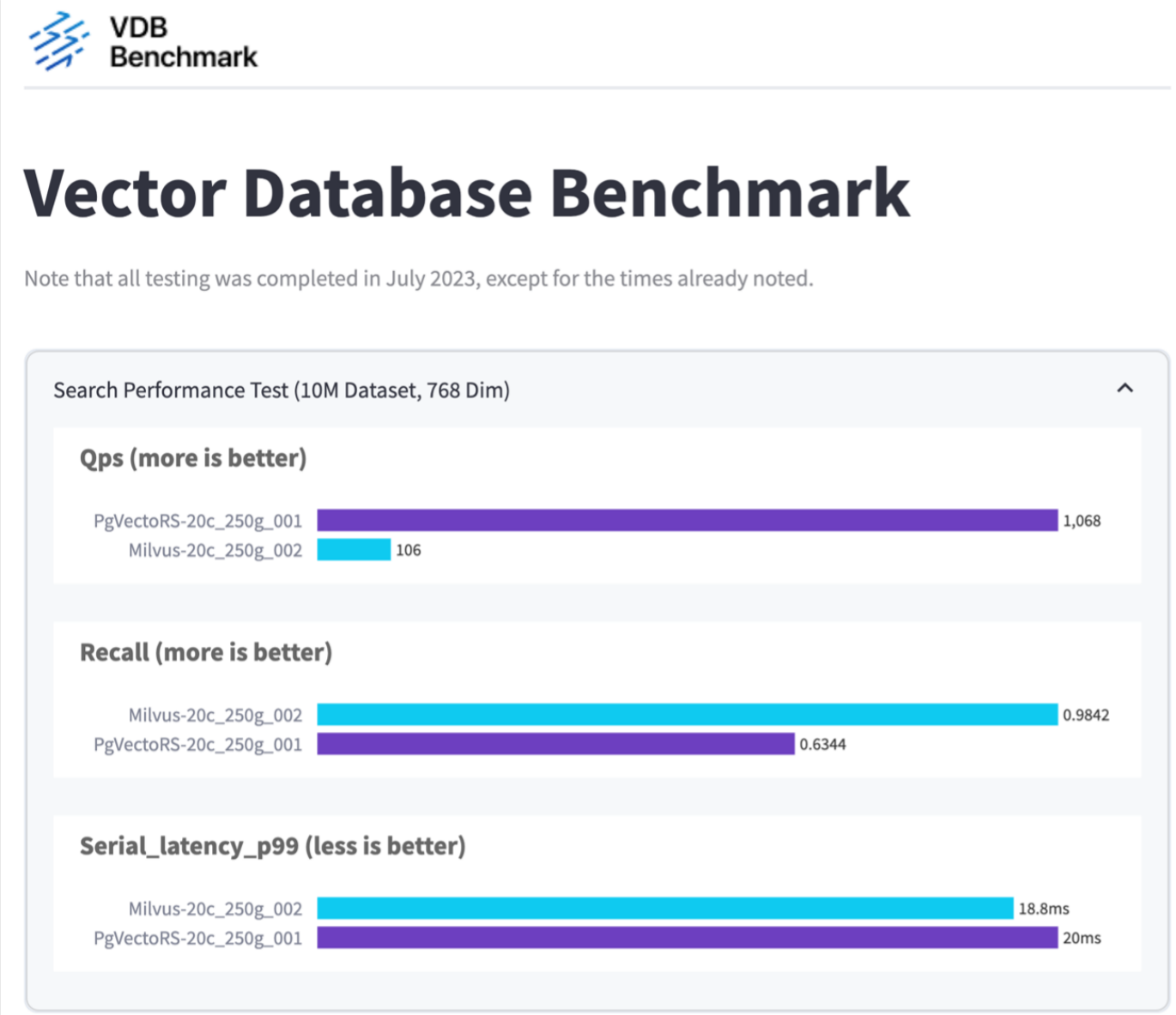


이 섹션에서는 특히 pgvecto.rs를 사용하여 PostgreSQL 데이터베이스를 테스트한 결과 및 관찰 결과를 공유합니다.
* 테스트 당시 DiskANN은 pgvecto.RS에 사용할 수 없었기 때문에 이러한 테스트의 인덱스 유형으로 HNSW를 선택했습니다.
* 데이터 수집 단계 동안, 우리는 768의 치수에서 천만 벡터로 구성된 COHERE 데이터세트를 로드했습니다. 이 과정은 약 4.5시간이 걸렸습니다.
* 쿼리 단계에서 0.6344를 리콜하여 1,068의 QPS(Queries per Second)를 확인했습니다. 쿼리의 99번째 백분위수 지연 시간은 20밀리초로 측정되었습니다. 대부분의 런타임 동안 클라이언트 CPU는 100% 용량으로 작동했습니다.

아래 이미지는 스토리지 클러스터 지연 시간 총 IOPS(초당 입출력 작업 수)를 포함한 다양한 스토리지 메트릭을 보여줍니다.


```
 The following section presents the key storage performance metrics.
image:pgvecto_storage_perf_metrics.png["오류: 그래픽 이미지가 없습니다"]
```

벡터 **DB** 벤치의 밀버스와 포스트그레스의 성능 비교



VectorDBBench를 사용한 Milvus 및 PostgreSQL의 성능 검증을 토대로 다음과 같은 점을 관찰했습니다.

- 인덱스 유형: HNSW
- 데이터 세트: 768차원으로 1,000만 벡터를 사용하는 COHERE

우리는 pgvecto.RS가 0.6344의 리콜로 1,068의 QPS(Queries per Second)를 달성했으며, Milvus는 0.9842의 리콜로 106의 QPS 속도를 달성했습니다.

쿼리의 높은 정밀도가 우선 순위인 경우 Milvus는 쿼리당 관련 항목의 비율이 더 높기 때문에 pgvecto.rs보다 성능이 뛰어납니다. 그러나 초당 쿼리 수가 더 중요한 요소인 경우 pgvecto.RS는 Milvus를 초과합니다. 하지만 pgvecto.rs를 통해 검색된 데이터의 품질이 낮고 검색 결과의 약 37%가 관련 없는 항목이라는 점을 유의해야 합니다.

성능 검증에 따른 관찰:

성능 검증을 토대로 다음과 같이 관찰했습니다.

Milvus의 I/O 프로필은 Oracle SLOB에서 볼 수 있는 OLTP 워크로드와 비슷합니다. 벤치마크는 데이터 수집, 사후 최적화 및 쿼리의 세 단계로 구성됩니다. 초기 단계는 주로 64KB 쓰기 작업이 특징이며, 쿼리 단계에는 대개 8KB

읽기가 포함됩니다. ONTAP는 Milvus I/O 로드를 능숙하게 처리할 것으로 기대하고 있습니다.

PostgreSQL 입출력 프로파일은 까다로운 스토리지 워크로드를 제공하지 않습니다. 현재 인메모리 구현이 진행 중이라는 점을 감안할 때 쿼리 단계에서 디스크 입출력을 관찰하지 못했습니다.

DiskANN은 스토리지 차별화를 위한 중요한 기술로 등장했습니다. 시스템 메모리 경계를 넘어 벡터 DB 검색의 효율적인 확장을 지원합니다. 그러나 HNSW와 같은 인메모리 벡터 DB 인덱스와 스토리지 성능 차이를 구별할 가능성은 거의 없습니다.

또한 인덱스 유형이 RAG 애플리케이션을 지원하는 벡터 데이터베이스의 가장 중요한 작동 단계인 HSNW인 경우 쿼리 단계에서 스토리지가 중요한 역할을 수행하지 않는다는 점도 주목할 필요가 있습니다. 여기서 중요한 점은 스토리지 성능이 이러한 애플리케이션의 전체 성능에 크게 영향을 미치지 않는다는 것입니다.

PostgreSQL:pgvector를 사용한 Instaclustr 벡터 데이터베이스

PostgreSQL:pgvector를 사용한 Instaclustr 벡터 데이터베이스

이 섹션에서는 instaclustr 제품이 pgvector 기능에 PostgreSQL과 통합되는 방법에 대해 자세히 설명합니다. "PGVector 및 PostgreSQL ® 을 사용하여 LLM 정확도와 성능을 개선하는 방법: Embeddings 및 PGVector의 역할 소개"의 예가 있습니다. 를 확인하십시오 "블로그" 자세한 내용을 보려면

Vector Database 사용 사례

Vector Database 사용 사례

이 섹션에서는 대규모 언어 모델을 사용한 검색 증강 세대 및 NetApp IT 챗봇과 같은 두 가지 사용 사례에 대해 설명합니다.

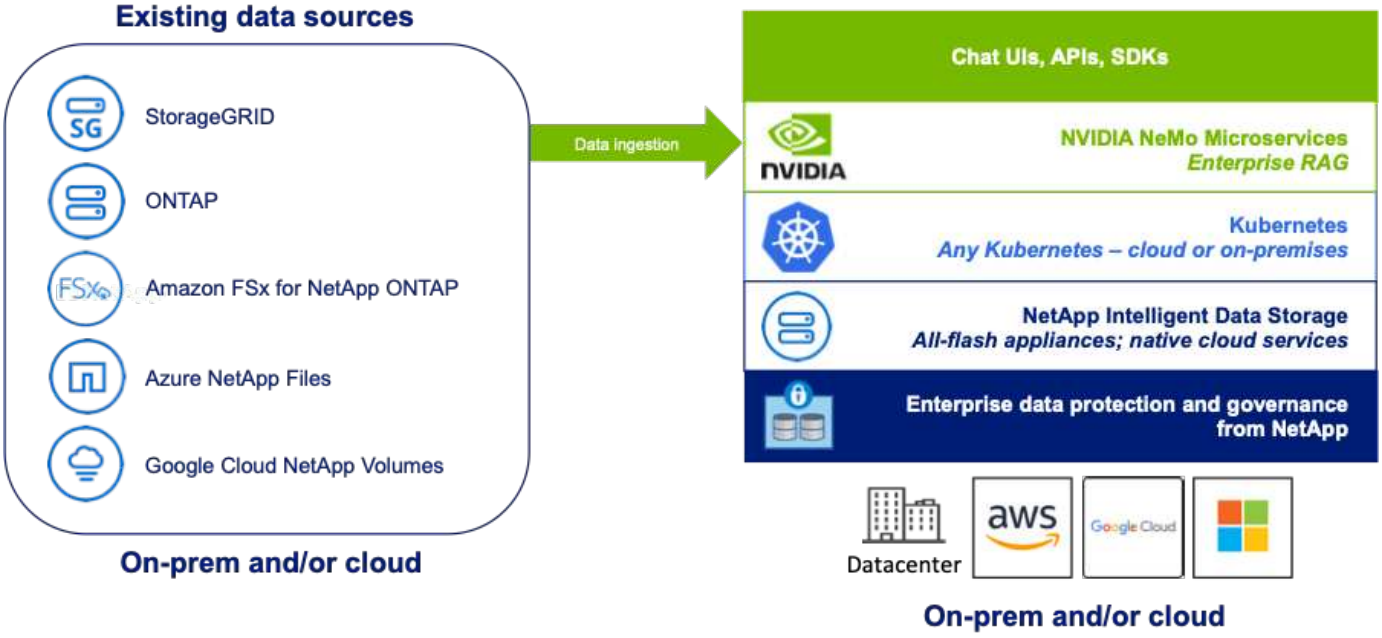
대형 언어 모델**(LLM)**을 사용한 검색 증강 생성**(RAG)**

```
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 Operator는 기업에서 RAG를 구현하는 데 유용한 도구입니다. 이 작업자는 전체 래그 파이프라인을 배포하는 데 사용할 수 있습니다. RAG 파이프라인은 Milvus 또는 pgvecto를 지식 기반 임베딩을 저장하기 위한 벡터 데이터베이스로 사용하도록 사용자 정의할 수 있습니다. 자세한 내용은 설명서를 참조하십시오.

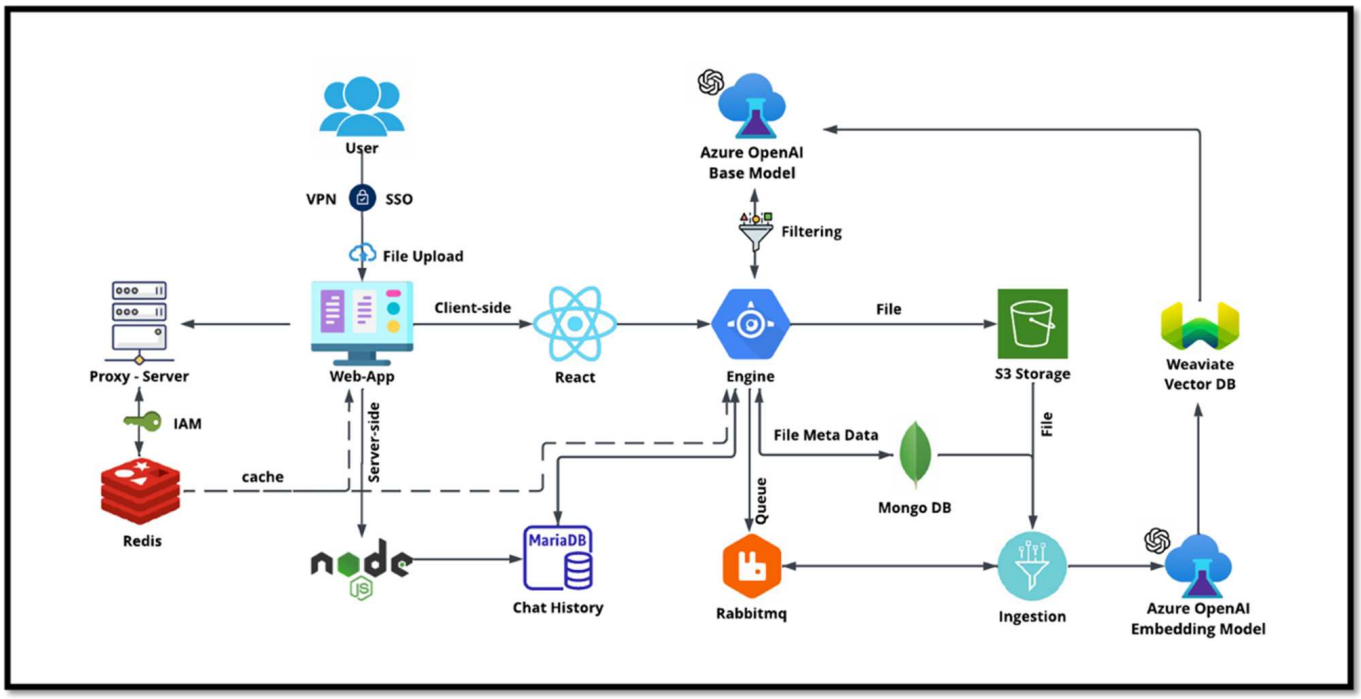
NetApp has validated an enterprise RAG architecture powered by the NVIDIA Enterprise RAG LLM Operator alongside NetApp storage. Refer to our blog post for more information and to see a demo. Figure 1 provides an overview of this architecture.

그림 1) NVIDIA Nemo 마이크로서비스 및 NetApp 기반 엔터프라이즈 RAG



NetApp IT 챗봇 사용 사례

NetApp의 챗봇은 벡터 데이터베이스를 위한 또 다른 실시간 사용 사례 역할을 합니다. 이 경우 NetApp Private OpenAI Sandbox는 NetApp 내부 사용자의 쿼리를 관리하는 효과적이고 안전하며 효율적인 플랫폼을 제공합니다. 엄격한 보안 프로토콜, 효율적인 데이터 관리 시스템 및 정교한 AI 처리 기능을 통합하여 SSO 인증을 통해 조직 내 역할과 책임에 따라 사용자에 대한 고품질의 정확한 응답을 보장합니다. 이 아키텍처는 고급 기술을 병합하여 사용자 중심의 지능형 시스템을 만들 수 있는 가능성을 강조합니다.



활용 사례는 네 가지 주요 섹션으로 나눌 수 있습니다.

사용자 인증 및 확인:

- 사용자 쿼리는 먼저 NetApp SSO(Single Sign-On) 프로세스를 통해 사용자의 신원을 확인합니다.
- 인증에 성공하면 시스템이 VPN 연결을 확인하여 안전한 데이터 전송을 보장합니다.

데이터 전송 및 처리:

- VPN이 검증되면 NetAIChat 또는 NetAICreate 웹 애플리케이션을 통해 MariaDB로 데이터가 전송됩니다. MariaDB는 사용자 데이터를 관리하고 저장하는 데 사용되는 빠르고 효율적인 데이터베이스 시스템입니다.
- 그런 다음 MariaDB는 NetApp Azure 인스턴스로 정보를 전송하여 사용자 데이터를 AI 처리 유닛에 연결합니다.

OpenAI 및 콘텐츠 필터링과의 상호 작용:

- Azure 인스턴스는 사용자의 질문을 콘텐츠 필터링 시스템으로 보냅니다. 이 시스템은 쿼리를 정리하고 처리할 수 있도록 준비합니다.
- 그런 다음 정리된 입력이 Azure OpenAI 기본 모델로 전송되고, 이 모델에서는 입력을 기반으로 응답을 생성합니다.

응답 생성 및 조정:

- 기본 모델의 응답을 먼저 점검하여 정확하고 콘텐츠 표준을 충족하는지 확인합니다.
- 확인을 통과하면 사용자에게 응답이 다시 전송됩니다. 이 프로세스를 통해 사용자는 자신의 쿼리에 대해 명확하고 정확하며 적절한 답변을 받을 수 있습니다.

결론

결론

결론적으로 이 문서에서는 NetApp 스토리지 솔루션에 Milvus 및 pgvector 같은 벡터 데이터베이스의 구축 및 관리에 대한 포괄적인 개요를 제공합니다. 또한 NetApp ONTAP 및 StorageGRID 오브젝트 스토리지를 활용하기 위한 인프라 지침을 논의하고 파일 및 오브젝트 저장소를 통해 AWS FSx for NetApp ONTAP의 Milvus 데이터베이스를 검증했습니다.

NetApp의 파일 오브젝트 이중성을 살펴보면서 벡터 데이터베이스의 데이터뿐만 아니라 다른 애플리케이션에도 유용성을 입증했습니다. 또한 NetApp의 엔터프라이즈 관리 제품인 SnapCenter이 벡터 데이터베이스 데이터에 대한 백업, 복원 및 클론 복제 기능을 제공하여 데이터 무결성과 가용성을 보장하는 방법에 대해서도 설명했습니다.

이 문서에서는 NetApp의 하이브리드 클라우드 솔루션이 사내 및 클라우드 환경 전반에서 데이터 복제 및 보호 기능을 제공하여 원활하고 안전한 데이터 관리 경험을 제공하는 방법에 대해서도 설명합니다. NetApp ONTAP에서 Milvus 및 pgvecto와 같은 벡터 데이터베이스의 성능 검증에 대한 통찰력을 제공하여 효율성과 확장성에 대한 중요한 정보를 제공했습니다.

마지막으로, LLM을 사용한 RAG와 NetApp의 내부 ChatAI라는 두 가지 세대 AI 사용 사례에 대해 논의했습니다. 이러한 실제 사례는 실제 적용 사례 및 이 문서에 설명된 개념과 관행의 이점을 강조합니다. 본 문서는 벡터 데이터베이스 관리를 위한 NetApp의 강력한 스토리지 솔루션을 활용하려는 모든 사용자에게 포괄적인 가이드입니다.

감사의 말

저자는 이 문서를 NetApp 고객 및 NetApp 분야에서 유용하게 활용할 수 있도록 아래 기고자와 의견을 제공해 주신 분들께 진심으로 감사드립니다.

1. Sathish Thyagarajan, NetApp, ONTAP AI & Analytics 기술 마케팅 엔지니어
2. NetApp 기술 마케팅 엔지니어 Mike Olesby
3. AJ Mahajan, 선임 이사, NetApp
4. Joe Scott, NetApp 워크로드 성능 엔지니어링 담당 매니저
5. Puneet Dhawan, 제품 관리 FSX, NetApp 수석 이사
6. Yuval Kalderon, NetApp FSx 제품 팀 선임 제품 매니저

추가 정보를 찾을 수 있는 위치

이 문서에 설명된 정보에 대해 자세히 알아보려면 다음 문서 및/또는 웹 사이트를 검토하십시오.

- Milvus 설명서 - https://milvus.io/docs/overview.md
- Milvus 독립형 설명서 - https://milvus.io/docs/v2.0.x/install_standalone-docker.md
- NetApp 제품 설명서 https://www.netapp.com/support-and-training/documentation/
- Instaclustr - "instalclustr 설명서"

버전 기록

버전	날짜	문서 버전 기록
버전 1.0	2024년 4월	최초 릴리스

부록 A: Values.YAML

부록 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:~#
```

부록 B: prepare_data_netapp_new.py

부록 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:~#
```

부록 C: verify_data_netapp.py

부록 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:~#
```

부록 D: docker-composition.yml

부록 D: docker-composition.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
```