Red Hat Enterprise Linux 8

스토리지 중복 제거 및 압축

VDO를 사용하여 RHEL 8의 스토리지 용량 최적화

Last Updated: 2022-07-31

Red Hat Enterprise Linux 8 스토리지 중복 제거 및 압축

VDO를 사용하여 RHEL 8의 스토리지 용량 최적화

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

법적 공지

초록

이 문서 컬렉션에서는 VDO(Virtual Data Optimizer)를 사용하여 Red Hat Enterprise Linux 8의 중복 제거 및 압축 스토리지 풀을 관리하는 방법에 대한 지침을 제공합니다.

차례

보다 포괄적 수용을 위한 오픈 소스 용어 교체

Red Hat은 코드, 문서, 웹 속성에서 문제가 있는 용어를 교체하기 위해 최선을 다하고 있습니다. 먼저 마스터(master), 슬레이브(slave), 블랙리스트(blacklist), 화이트리스트(whitelist) 등 네 가지 용어를 교체하고 있습니다. 이러한 변경 작업은 작업 범위가 크므로 향후 여러 릴리스에 걸쳐 점차 구현할 예정입니다. 자세한 내용은 CTO Chris Wright의 메시지 를 참조하십시오.

RED HAT 문서에 관한 피드백 제공

문서 개선을 위한 의견을 보내 주십시오. 문서를 개선하는 방법을 알려주십시오.

- 특정 문구에 대한 간단한 의견 작성 방법은 다음과 같습니다.
  1. 문서가 *Multi-page HTML* 형식으로 표시되는지 확인합니다. 또한 문서 오른쪽 상단에 **피드백** 버튼이 있는지 확인합니다.
  2. 마우스 커서를 사용하여 주석 처리하려는 텍스트 부분을 강조 표시합니다.
  3. 강조 표시된 텍스트 아래에 표시되는 **피드백 추가** 팝업을 클릭합니다.
  4. 표시된 지침을 따릅니다.
- Bugzilla를 통해 피드백을 제출하려면 새 티켓을 생성합니다.
  1. Bugzilla 웹 사이트로 이동하십시오.
  2. Component로 **Documentation**을 선택하십시오.
  3. **Description** 필드에 문서 개선을 위한 제안 사항을 기입하십시오. 관련된 문서의 해당 부분 링크를 알려주십시오.
  4. **Submit Bug**를 클릭하십시오.

1장. VDO 배포

시스템 관리자는 VDO를 사용하여 중복 제거 및 압축 스토리지 풀을 만들 수 있습니다.

1.1. VDO 소개

VDO(가상 데이터 최적화 도구)는 중복 제거, 압축 및 씬 프로비저닝의 형태로 Linux에 대한 인라인 데이터 감소를 제공합니다. VDO 볼륨을 설정할 때 VDO 볼륨을 구성할 블록 장치를 지정하고 현재 보유한 논리 스토리지의 양을 지정합니다.

- 활성 VM 또는 컨테이너를 호스팅할 때 Red Hat은 10:1 논리적 및 물리적 비율을 사용하여 스토리지를 프로비저닝하는 것이 좋습니다. 즉, 1TB의 물리적 스토리지를 사용하는 경우 10TB의 논리적 스토리지가 됩니다.

- Ceph에서 제공하는 유형과 같은 개체 스토리지의 경우, 3:1 논리적 및 물리적 비율을 사용하는 것이 좋습니다. 즉, 물리적 스토리지의 1TB가 3TB 논리 스토리지로 존재할 것을 권장합니다.

두 경우 모두 VDO에서 제공하는 논리적 장치 상단에 파일 시스템을 배치한 다음 직접 또는 분산 클라우드 스토리지 아키텍처의 일부로 사용할 수 있습니다.

VDO는 씬 프로비저닝되므로 파일 시스템과 애플리케이션은 사용 중인 논리 공간만 표시하며 실제 사용 가능한 실제 공간을 인식하지 못합니다. 스크립팅을 사용하여 실제 사용 가능한 공간을 모니터링하고 임계값을 초과하는 경우 경고를 생성합니다(예: VDO 볼륨이 80% 가득 차는 경우).

추가 리소스

- 물리적 공간 모니터링에 대한 자세한 내용은 2.1절. "VDO 볼륨에서 여유 공간 관리" 을 참조하십시오.

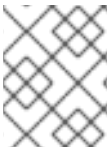
1.2. VDO 배포 시나리오

다음과 같은 목적으로 중복 제거 스토리지를 제공하는 다양한 방법으로 VDO를 배포할 수 있습니다.

- 블록 및 파일 액세스

- 로컬 및 원격 스토리지 둘 다

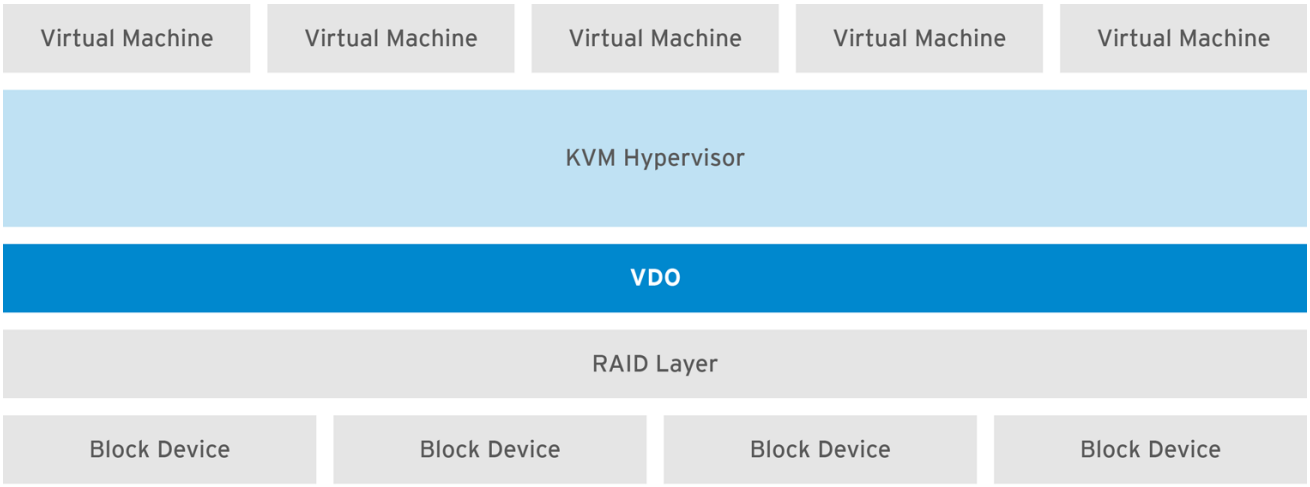
VDO는 중복 제거 스토리지를 표준 Linux 블록 장치로 노출하므로 표준 파일 시스템, iSCSI 및 FC 대상 드라이버 또는 통합 스토리지와 함께 사용할 수 있습니다.



참고

현재 Ceph RADOS Block Device(RBD) 상단에 VDO 볼륨 배포가 지원됩니다. 그러나 VDO 볼륨 상단에 Red Hat Ceph Storage 클러스터 구성 요소의 배포는 현재 지원되지 않습니다.

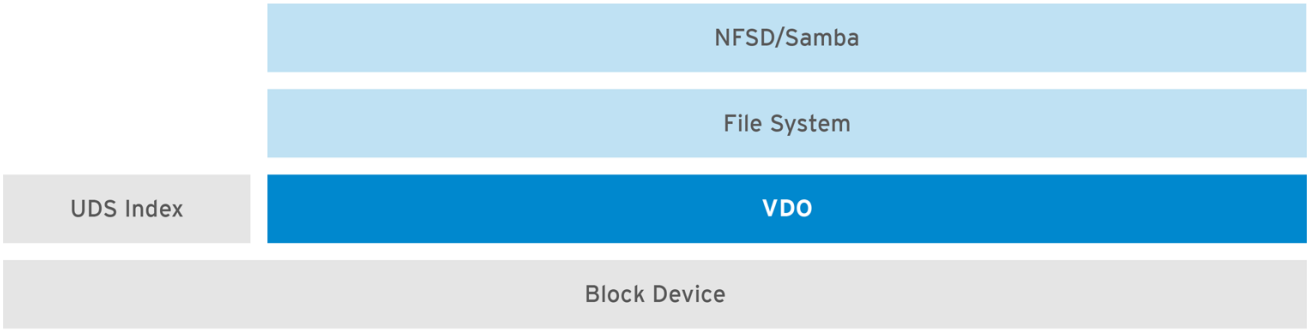
KVM
직접 연결된 스토리지로 구성된 KVM 서버에 VDO를 배포할 수 있습니다.



RHEL_462492_1117

파일 시스템

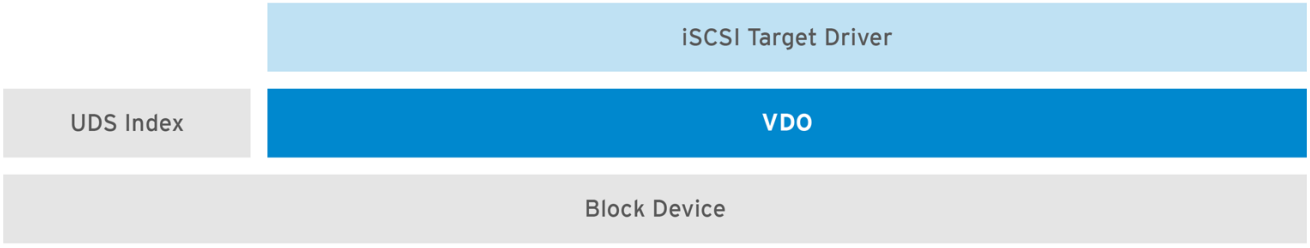
VDO 상단에 파일 시스템을 생성하고 NFS 서버 또는 Samba를 사용하여 NFS 또는 CIFS 사용자에게 노출할 수 있습니다.



RHEL_466924_0218

iSCSI에 VDO 배치

VDO 스토리지 대상 전체를 iSCSI 대상으로 원격 iSCSI 이니시에이터로 내보낼 수 있습니다.



RHEL_466924_0218

iSCSI에서 VDO 볼륨을 생성할 때 iSCSI 계층 위에 또는 아래에 VDO 볼륨을 배치할 수 있습니다. 고려해야 할 사항은 여러 가지가 있지만 환경에 가장 적합한 방법을 선택할 수 있도록 몇 가지 지침이 제공됩니다.

iSCSI 계층 아래의 iSCSI 서버(대상)에 VDO 볼륨을 배치하는 경우:

- VDO 볼륨은 다른 iSCSI LUN과 유사하게 이니시에이터에 투명합니다. 클라이언트의 씬 프로비저닝 및 공간 절약을 숨기면 LUN이 표시되고 모니터링 및 유지 관리가 쉬워집니다.

- VDO 메타데이터 읽기 또는 쓰기가 없고 dedupe 권고에 대한 읽기 확인이 네트워크에서 수행되지 않으므로 네트워크 트래픽이 줄어듭니다.
- iSCSI 대상에서 사용 중인 메모리 및 CPU 리소스는 성능이 향상될 수 있습니다. 예를 들어, 볼륨 감소가 iSCSI 대상에서 발생하기 때문에 하이퍼바이저의 수가 늘어나는 것을 호스팅할 수 있습니다.
- 클라이언트에서 이니시에이터에서 암호화를 구현하고 대상 아래에 VDO 볼륨이 있는 경우 공간 절약이 실현되지 않습니다.

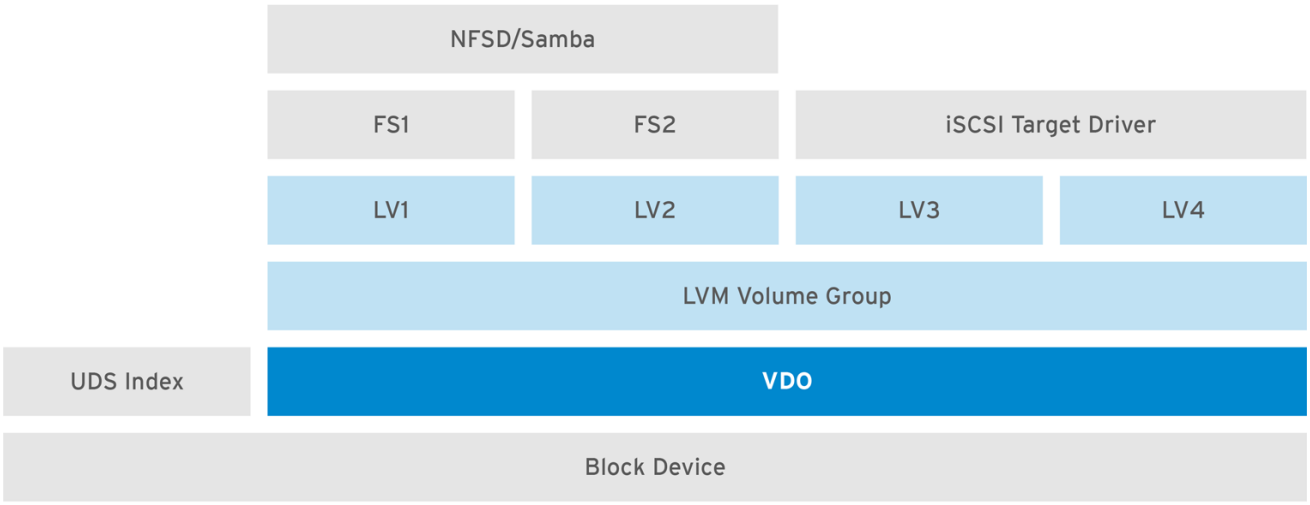
iSCSI 계층 위의 iSCSI 클라이언트(이니시에이터)에 VDO 볼륨을 배치하는 경우:

- 공간 절약률이 높은 경우 ASYNC 모드에서 네트워크 전반에 낮은 네트워크 트래픽이 발생할 가능성이 있습니다.
- 공간 절약을 직접 보고 제어하고 사용량을 모니터링할 수 있습니다.
- 예를 들어 **dm-crypt** 를 사용하여 데이터를 암호화하려는 경우 crypt에 VDO를 구현하고 공간 효율성을 활용할 수 있습니다.

LVM

기능이 풍부한 시스템에서 LVM을 사용하여 동일한 중복 제거 스토리지 풀에서 모두 지원하는 여러 개의 논리 장치 번호(LUN)를 제공할 수 있습니다.

다음 다이어그램에서 VDO 대상은 LVM에서 관리할 수 있도록 물리 볼륨으로 등록됩니다. 여러 논리 볼륨(*LV1 ~ LV4*)*이 중복 스토리지 풀에서 생성됩니다*. 이러한 방식으로 VDO는 다중 프로토콜 통합 블록 또는 기본 중복 제거 스토리지 풀에 대한 파일 액세스를 지원할 수 있습니다.

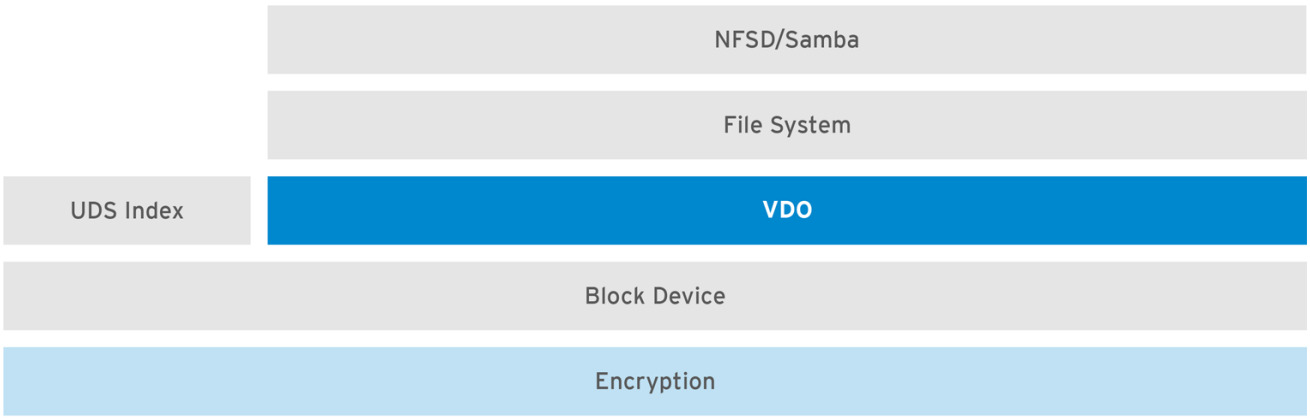


RHEL_466924_0218

중복화된 통합 스토리지 설계를 사용하면 여러 파일 시스템에서 LVM 도구를 통해 동일한 중복 제거 도메인을 전체적으로 사용할 수 있습니다. 또한 파일 시스템은 모두 VDO 위에 LVM 스냅샷, copy-on-write, 축소 또는 확장 기능을 활용할 수 있습니다.

Encryption

DM Crypt와 같은 DM(Device Mapper) 메커니즘은 VDO와 호환됩니다. VDO 볼륨을 암호화하면 데이터 보안을 보장할 수 있으며 VDO 위의 모든 파일 시스템은 여전히 중복 제거됩니다.



중요

VDO 위의 암호화 계층을 적용하면 데이터 중복 제거가 거의 발생하지 않습니다. VDO에서 중복을 제거하기 전에 암호화를 사용하면 중복 블록이 서로 다릅니다.

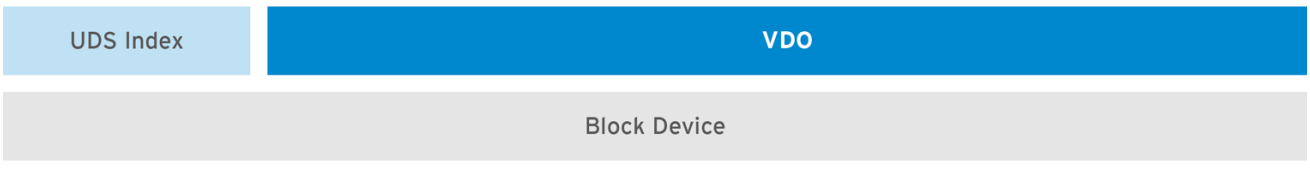
항상 VDO 아래에 암호화 계층을 배치합니다.

iSCSI에서 VDO 볼륨을 생성할 때 iSCSI 계층 위에 또는 아래에 VDO 볼륨을 배치할 수 있습니다. 고려해야 할 사항은 여러 가지가 있지만 환경에 가장 적합한 방법을 선택할 수 있도록 몇 가지 지침이 제공됩니다.

1.3. VDO 볼륨의 구성 요소

VDO는 블록 장치를 백업 저장소로 사용하며, 이 저장소에는 하나 이상의 디스크, 파티션 또는 플랫 파일로 구성된 물리적 스토리지 집계가 포함될 수 있습니다. 스토리지 관리 툴에서 VDO 볼륨을 생성할 때 VDO는 UDS 인덱스 및 VDO 볼륨을 위한 볼륨 공간을 예약합니다. UDS 인덱스와 VDO 볼륨은 중복 제거 블록 스토리지를 제공하기 위해 상호 작용합니다.

그림 1.1. VDO 디스크 조직



VDO 솔루션은 다음 구성 요소로 구성됩니다.

kvdo

Linux 장치 매퍼 계층으로 로드되는 커널 모듈은 중복 제거, 압축 및 씬 프로비저닝된 블록 스토리지 볼륨을 제공합니다.
kvdo 모듈은 블록 장치를 노출합니다. 블록 스토리지에 대해 이 블록 장치에 직접 액세스하거나 XFS 또는 ext4와 같은 Linux 파일 시스템을 통해 제공할 수 있습니다.

kvdo 에서 VDO 볼륨의 논리적 데이터 블록을 읽도록 요청을 받으면 요청된 논리 블록을 기본 물리 블록에 매핑한 다음 요청된 데이터를 읽고 반환합니다.

kvdo 에서 VDO 볼륨에 데이터 블록을 쓰기 요청을 수신하면 먼저 요청이 DISCARD 또는 TRIM 요청인지 또는 데이터가 균일하게 0인지 여부를 확인합니다. 이러한 조건 중 하나가 true이면 **kvdo** 는 블록 맵을 업데이트하고 요청을 승인합니다. 그렇지 않으면 VDO는 데이터를 처리하고 최적화합니다.

UDS

볼륨에서 UDS(Universal Deduplication Service) 인덱스와 통신하고 중복 데이터를 분석하는 커널 모듈. 새로운 데이터 각각에 대해 UDS는 이전에 저장된 데이터와 동일한지 신속하게 판별합니다. 인덱스가 일치하는 항목을 발견하면 스토리지 시스템은 기존 항목을 내부적으로 참조하여 동일한 정보를 두 번 이상 저장하지 않도록 할 수 있습니다.
UDS 인덱스는 **uds** 커널 모듈로 커널 내부에서 실행됩니다.

명령행 도구

최적화된 스토리지 구성 및 관리.

1.4. VDO 볼륨의 물리 및 논리 크기

이 섹션에서는 VDO가 활용할 수 있는 물리적 크기, 사용 가능한 물리적 크기 및 VDO에 대해 설명합니다.

물리 크기

이 크기는 기본 블록 장치와 동일합니다. VDO는 다음을 위해 이 스토리지를 사용합니다.

- 중복 제거 및 압축 가능한 사용자 데이터
- UDS 인덱스와 같은 VDO 메타데이터

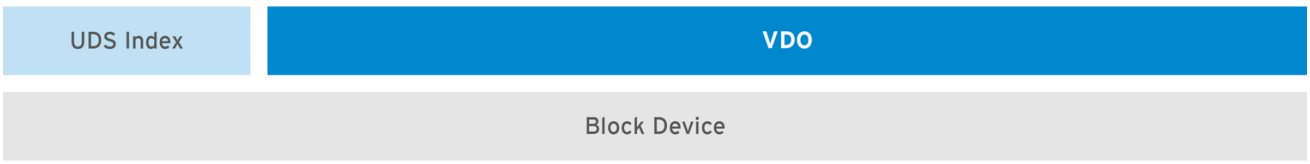
사용 가능한 물리 크기

이것은 VDO가 사용자 데이터에 사용할 수있는 물리적 크기의 부분입니다.
이 값은 물리 크기에서 메타데이터 크기를 뺀 값과 같고, 볼륨을 slab 크기로 분할한 후 나머지 부분을 slab 크기로 나눕니다.

논리 크기

이는 VDO 볼륨이 애플리케이션에 제공하는 프로비저닝된 크기입니다. 일반적으로 사용 가능한 실제 크기보다 큽니다. vdo **LogicalSize** 옵션이 지정되지 않은 경우 이제 논리 볼륨의 프로비저닝이 **1:1** 비율로 프로비저닝됩니다. 예를 들어, VDO 볼륨이 20GB 블록 장치에 배치된 경우 2.5GB는 UDS 인덱스용으로 예약됩니다(기본 인덱스 크기가 사용되는 경우). 나머지 17.5GB는 VDO 메타데이터 및 사용자 데이터에 대해 제공됩니다. 따라서 사용할 수 있는 스토리지가 17.5GB를 넘지 않으며 실제 VDO 볼륨을 구성하는 메타데이터로 인해 더 적을 수 있습니다.
VDO는 현재 절대 최대 논리 크기가 4PB인 물리 볼륨의 최대 254배의 논리 크기를 지원합니다.

그림 1.2. VDO 디스크 조직



RHEL_466924_0218

이 그림에서 VDO 중복 제거 스토리지 대상은 블록 장치에 완전히 배치되므로 VDO 볼륨의 물리적 크기가 기본 블록 장치와 동일한 크기입니다.

주가 리소스

- VDO 메타데이터가 다양한 크기의 블록 장치에 필요한 스토리지 VDO 메타데이터에 대한 자세한 내용은 1.6.4절. "물리 크기에 따른 VDO 요구 사항의 예" 을 참조하십시오.

1.5. VDO의 슬랩 크기

VDO 볼륨의 물리 스토리지는 여러 슬랩으로 나뉩니다. 각 조각은 물리적 공간의 인접 지역입니다. 지정된 볼륨에 대한 모든 슬래브는 크기가 같으며 2의 배수 128MB 최대 32GB의 거듭제곱이 될 수 있습니다.

기본 slab 크기는 작은 테스트 시스템에서 VDO를 쉽게 평가할 수 있도록 2GB입니다. 단일 VDO 볼륨은 최대 8192 개의 슬래브를 보유할 수 있습니다. 따라서 2GB의 slab이 있는 기본 구성에서 허용되는 최대 물리적 스토리지는 16TB입니다. 32GB slab을 사용하는 경우 허용되는 최대 물리 스토리지는 256TB입니다. VDO는 항상 메타데이터를 위해 최소 하나의 slab을 예약하므로 예약된 슬랩을 사용자 데이터를 저장하는 데 사용할 수 없습니다.

Slab 크기는 VDO 볼륨의 성능에 영향을 미치지 않습니다.

표 1.1. 물리 볼륨 크기 별 권장 VDO 슬래브 크기

물리 볼륨 크기	권장 슬랩 크기
10~99GB	1GB
100GB - 1TB	2GB
2-256 TB	32GB

참고

기본 설정 2GB slab 크기와 0.25 밀도 인덱스를 사용하는 VDO 볼륨의 최소 디스크 사용량에는 approx 4.7GB가 필요합니다. 이는 0% 중복 제거 또는 압축에서 쓸 수 있는 2GB의 물리적 데이터보다 약간 적습니다.

여기에서 최소 디스크 사용은 기본 slab 크기 및 dense 인덱스의 합계입니다.

lvcreate 명령에 **--config 'allocation/vdo_slab_size_mb=*size-in-megabytes*'** 옵션을 제공하여 slab 크기를 제어할 수 있습니다.

1.6. VDO 요구 사항

VDO에는 배치 및 시스템 리소스에 대한 특정 요구 사항이 있습니다.

1.6.1. VDO 메모리 요구 사항

각 VDO 볼륨에는 두 개의 고유한 메모리 요구 사항이 있습니다.

VDO 모듈

VDO에는 고정된 38MB의 RAM과 몇 가지 변수 양이 필요합니다.

- 구성된 블록 맵 캐시 크기 1MB당 1.15MB의 RAM. 블록 맵 캐시에는 최소 150MB의 RAM이 필요합니다.

- 논리 공간 1TB당 1.6MB의 RAM.
- 볼륨에서 관리하는 물리적 스토리지의 1TB당 268MB의 RAM.

UDS 인덱스

UDS(Universal Deduplication Service)에는 최소 250MB의 RAM이 필요합니다. 이는 중복 제거에서 사용하는 기본 양이기도 합니다. 이 값은 인덱스에 필요한 스토리지 크기에도 영향을 미치므로 VDO 볼륨을 포맷할 때 값을 구성할 수 있습니다.
UDS 인덱스에 필요한 메모리는 인덱스 유형 및 중복 제거 창의 필수 크기에 따라 결정됩니다.

인덱스 유형	중복 제거 창	참고
밀도	1GB RAM당 1TB	1GB 밀도 지수는 일반적으로 최대 4TB의 물리적 스토리지에 충분합니다.
스파스	1GB RAM당 10TB	일반적으로 1GB 스파스 인덱스는 최대 40TB의 물리적 스토리지에 충분합니다.

참고

기본 설정 2GB slab 크기와 0.25 밀도 인덱스를 사용하는 VDO 볼륨의 최소 디스크 사용량에는 approx 4.7GB가 필요합니다. 이는 0% 중복 제거 또는 압축에서 쓸 수 있는 2GB의 물리적 데이터보다 약간 적습니다.

여기에서 최소 디스크 사용은 기본 slab 크기 및 dense 인덱스의 합계입니다.

UDS Sparse Indexing 기능은 VDO에 권장되는 모드입니다. 데이터의 시간적 로컬성에 의존하고 메모리에서 가장 관련성이 높은 인덱스 항목만 유지하려고 합니다. 스파스 인덱스를 사용하면 UDS는 동일한 양의 메모리를 사용하는 동시에 밀도가 있는 것보다 10배 큰 중복 제거 창을 유지할 수 있습니다.

스파스 인덱스가 가장 큰 범위를 제공하지만, 밀도가 더 많은 중복 제거 조언을 제공합니다. 동일한 양의 메모리에 대해 대부분의 워크로드에서 밀도 및 스파스 인덱스 간의 중복 제거 비율의 차이는 무시할 수 있습니다.

추가 리소스

- 물리 크기에 따른 VDO 요구 사항의 예

1.6.2. VDO 스토리지 공간 요구 사항

최대 256TB의 물리 스토리지를 사용하도록 VDO 볼륨을 구성할 수 있습니다. 물리적 스토리지의 특정 부분만 데이터를 저장하는 데 사용할 수 있습니다. 이 섹션에서는 VDO 관리 볼륨의 사용 가능한 크기를 결정하는 계산 방법을 제공합니다.

VDO는 두 가지 유형의 VDO 메타데이터 및 UDS 인덱스에 대한 스토리지가 필요합니다.

- 첫 번째 유형의 VDO 메타데이터는 약 1MB의 *물리적 스토리지와* slab당 1MB를 추가로 사용합니다.
- 두 번째 유형의 VDO 메타데이터는 1GB의 *논리 스토리지* 마다 약 1.25MB를 사용하고 가장 가까운 슬랩으로 반올림합니다.

- UDS 인덱스에 필요한 스토리지 양은 인덱스 유형 및 인덱스에 할당된 RAM 크기에 따라 달라집니다. 1GB RAM마다 밀도가 높은 UDS 인덱스는 17GB의 스토리지를 사용하며 스파스 UDS 인덱스는 170GB의 스토리지를 사용합니다.

추가 리소스

- 1.6.4절. "물리 크기에 따른 VDO 요구 사항의 예"
- 1.5절. "VDO의 슬랩 크기"

1.6.3. 스토리지 스택에 VDO 배치

VDO 및 VDO 위의 다른 스토리지 계층 아래에 특정 스토리지 계층을 배치해야 합니다.

이 섹션에서는 계층 *A* 가 *B* *계층 위에 있을 때* *A* *가 장치* *B* 에 직접 저장되거나 *B* 에 저장된 계층에 간접적으로 저장된다는 것을 *의미합니다*. 마찬가지로, *B* *아래에* *A* 는 *B* 가 *A* 에 *저장* 됨을 의미합니다.

VDO 볼륨은 씬 프로비저닝된 블록 장치입니다. 물리 공간이 부족해지는 것을 방지하려면 나중에 확장할 수 있는 스토리지 계층 위에 볼륨을 배치합니다. 이러한 확장 가능한 스토리지의 예로는 LVM 볼륨 또는 MD RAID 배열이 있습니다.

VDO 위에 씩 프로비저닝된 계층을 배치할 수 있지만 이 경우 씩 프로비저닝의 보장을 받을 수 없습니다. VDO 계층은 씬 프로비저닝되므로 씬 프로비저닝의 영향은 위의 모든 계층에 적용됩니다. VDO 장치를 모니터링하지 않으면 VDO보다 씩 프로비저닝된 볼륨의 물리적 공간이 부족할 수 있습니다.

지원되는 구성

- VDO 아래에만 배치할 수 있는 계층:
  - DM Multipath
  - DM 암호화
  - 소프트웨어 RAID(LVM 또는 MD RAID)
- VDO 위에만 배치할 수 있는 계층:
  - LVM 캐시
  - LVM 스냅샷
  - LVM 씬 프로비저닝

지원되지 않는 로깅 구성

- 다른 VDO 볼륨 위의 VDO
- LVM 스냅샷 위의 VDO
- LVM 캐시 위의 VDO
- 루프백 장치 위의 VDO
- LVM 씬 프로비저닝 위의 VDO
- VDO 이상의 암호화된 볼륨

- VDO 볼륨의 파티션
- RAID(예: LVM RAID, MD RAID 또는 VDO 볼륨 위의 기타 유형)

추가 리소스

- LVM 계층을 사용한 VDO 스택에 대한 자세한 내용은 Stacking LVM volumes 문서를 참조하십시오.

1.6.4. 물리 크기에 따른 VDO 요구 사항의 예

다음 표에서는 기본 볼륨의 물리적 크기에 따라 VDO의 대략적인 시스템 요구 사항을 제공합니다. 각 테이블에는 기본 스토리지 또는 백업 스토리지와 같이 의도한 배포에 적합한 요구 사항이 나열되어 있습니다.

정확한 숫자는 VDO 볼륨 구성에 따라 다릅니다.

기본 스토리지 배포

1차 스토리지의 경우 UDS 인덱스는 물리적 크기의 크기에서 25% 사이입니다.

표 1.2. 기본 스토리지에 대한 스토리지 및 메모리 요구 사항

물리 크기	RAM 사용량: UDS	RAM 사용량: VDO	디스크 사용량	인덱스 유형
10GB–1TB	250MB	472MB	2.5GB	밀도
2–10TB	1GB	3GB	10GB	밀도
	250MB		22GB	스파스
11–50TB	2GB	14GB	170GB	스파스
51–100TB	3GB	27GB	255GB	스파스
101–256TB	12GB	69GB	1020GB	스파스

백업 스토리지 배포

백업 스토리지의 경우 UDS 인덱스는 백업 세트의 크기를 처리하지만 실제 크기보다 크지는 않습니다. 향후 백업 세트 또는 물리적 크기가 증가할 것으로 예상되는 경우 인덱스 크기로 고려합니다.

표 1.3. 백업 스토리지에 대한 스토리지 및 메모리 요구 사항

물리 크기	RAM 사용량: UDS	RAM 사용량: VDO	디스크 사용량	인덱스 유형
10GB–1TB	250MB	472MB	2.5GB	밀도
2–10TB	2GB	3GB	170GB	스파스
11–50TB	10GB	14GB	850GB	스파스

물리 크기	RAM 사용량: UDS	RAM 사용량: VDO	디스크 사용량	인덱스 유형
51–100TB	20GB	27GB	1700GB	스파스
101–256TB	26GB	69GB	3400GB	스파스

1.7. VDO 설치

이 절차에서는 VDO 볼륨을 생성, 마운트 및 관리하는 데 필요한 소프트웨어를 설치합니다.

절차

- **vdo** 및 **kmod-kvdo** 패키지를 설치합니다.

```
# yum install vdo kmod-kvdo
```

1.8. VDO 볼륨 생성

다음 절차에서는 블록 장치에 VDO 볼륨을 생성합니다.

사전 요구 사항

- VDO 소프트웨어를 설치합니다. 1.7절. "VDO 설치"을 참조하십시오.
- 확장 가능한 스토리지를 백업 블록 장치로 사용합니다. 자세한 내용은 1.6.3절. "스토리지 스택에 VDO 배치"의 내용을 참조하십시오.

절차

다음 모든 단계에서 *vdo-name* 을 VDO 볼륨에 사용하려는 식별자로 바꿉니다(예: **vdo1).** 시스템에서 VDO의 각 인스턴스에 다른 이름 및 장치를 사용해야 합니다.

1. VDO 볼륨을 생성하려는 블록 장치의 영구 이름을 찾습니다. 영구 이름에 대한 자세한 내용은 6 장. *영구 이름 지정 속성 개요* 을 참조하십시오.
   비영구적인 장치 이름을 사용하는 경우 향후 장치 이름이 변경되면 VDO가 제대로 시작되지 않을 수 있습니다.

2. VDO 볼륨을 생성합니다.

```
# vdo create \
    --name=vdo-name \
    --device=block-device \
    --vdoLogicalSize=logical-size
```

   - VDO 볼륨을 생성하려는 블록 장치의 영구적인 이름으로 block -*device* 를 바꿉니다. 예를 들면 **/dev/disk/by-id/scsi-3600508b1001c264ad2af21e903ad031f** 입니다.
   - *logical-size* 를 VDO 볼륨이 있어야 하는 논리 스토리지 양으로 교체합니다.
     - 활성 VM 또는 컨테이너 스토리지의 경우 블록 장치의 물리적 크기 10 배에 해당하는 논리적 크기를 사용합니다. 예를 들어 블록 장치의 크기가 1TB이면 여기에서 **10T** 를 사용합니

다.

- 오브젝트 스토리지의 경우 블록 장치의 물리적 크기 **세** 배인 논리 크기를 사용합니다. 예를 들어 블록 장치의 크기가 1TB인 경우 여기에서 **3T** 를 사용합니다.

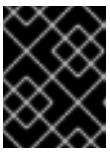
- 물리 블록 장치가 16TiB를 초과하는 경우 **--vdoSlabSize=32G** 옵션을 추가하여 볼륨의 slab 크기를 32GiB로 늘립니다.
  16TiB보다 큰 블록 장치에 2GiB의 기본 슬랩 크기를 사용하면 **vdo create** 명령이 다음 오류와 함께 실패합니다.

```
vdo: ERROR - vdoformat: formatVDO failed on '/dev/device': VDO Status: Exceeds maximum number of slabs supported
```

예 1.1. 컨테이너 스토리지를 위한 VDO 생성

예를 들어 1TB 블록 장치에 컨테이너 스토리지를 위한 VDO 볼륨을 생성하려면 다음을 사용할 수 있습니다.

```
# vdo create \
    --name=vdo1 \
    --device=/dev/disk/by-id/scsi-3600508b1001c264ad2af21e903ad031f \
    --vdoLogicalSize=10T
```



중요

VDO 볼륨을 생성할 때 오류가 발생하면 볼륨을 제거하여 정리합니다. 자세한 내용은 2.10.2절. "성공적으로 생성되지 않은 VDO 볼륨 제거" 을 참조하십시오.

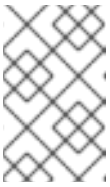
3. VDO 볼륨 상단에 파일 시스템을 생성합니다.

   - XFS 파일 시스템의 경우:

```
# mkfs.xfs -K /dev/mapper/vdo-name
```

   - ext4 파일 시스템의 경우:

```
# mkfs.ext4 -E nodiscard /dev/mapper/vdo-name
```



참고

새로 생성된 VDO 볼륨에 대한 **-K** 및 **-E nodiscard** 옵션의 목적은 할당되지 않은 블록에 영향을 미치지 않기 때문에 요청을 보내는 데 시간을 소비하지 않는 것입니다. 새로운 VDO 볼륨은 할당되지 않은 100%를 시작합니다.

4. 다음 명령을 사용하여 시스템이 새 장치 노드를 등록할 때까지 기다립니다.

```
# udevadm settle
```

다음 단계

1. 파일 시스템을 마운트합니다. 자세한 내용은 1.9절. "VDO 볼륨 마운트" 을 참조하십시오.

2. VDO 장치에서 파일 시스템의 **삭제** 기능을 활성화합니다. 자세한 내용은 1.10절. "주기적인 블록 삭제 활성화" 을 참조하십시오.

추가 리소스

- **vdo(8)** 도움말 페이지

1.9. VDO 볼륨 마운트

이 절차에서는 VDO 볼륨에 파일 시스템을 수동으로 또는 영구적으로 마운트합니다.

사전 요구 사항

- 시스템에 VDO 볼륨이 생성되었습니다. 자세한 내용은 1.8절. "VDO 볼륨 생성" 의 내용을 참조하십시오.

절차

- VDO 볼륨에 파일 시스템을 수동으로 마운트하려면 다음을 사용합니다.

```
# mount /dev/mapper/vdo-name mount-point
```

- 부팅 시 자동으로 마운트되도록 파일 시스템을 구성하려면 **/etc/fstab** 파일에 행을 추가합니다.
  - XFS 파일 시스템의 경우:

```
/dev/mapper/vdo-name mount-point xfs defaults 0 0
```

  - ext4 파일 시스템의 경우:

```
/dev/mapper/vdo-name mount-point ext4 defaults 0 0
```

  VDO 볼륨이 iSCSI와 같은 네트워크가 필요한 블록 장치에 있는 경우 **_netdev** 마운트 옵션을 추가합니다.

추가 리소스

- **vdo(8)** 도움말 페이지.
- 네트워크가 필요한 iSCSI 및 기타 블록 장치의 경우 **_netdev** 마운트 옵션에 대한 자세한 내용은 **systemd.mount(5)** 도움말 페이지를 참조하십시오.

1.10. 주기적인 블록 삭제 활성화

이 절차에서는 지원되는 모든 파일 시스템에서 사용하지 않는 블록을 정기적으로 삭제하는 **systemd** 타이머를 활성화합니다.

절차

- **systemd** 타이머를 활성화하고 시작합니다.

```
# systemctl enable --now fstrim.timer
```

1.11. VDO 모니터링

다음 절차에서는 VDO 볼륨에서 사용 및 효율성 정보를 얻는 방법을 설명합니다.

사전 요구 사항

- VDO 소프트웨어를 설치합니다. 1.7절. "VDO 설치"을 참조하십시오.

절차

- **vdostats** 유틸리티를 사용하여 VDO 볼륨에 대한 정보를 가져옵니다.

```
# vdostats --human-readable

Device                    1K-blocks      Used  Available  Use%  Space saving%
/dev/mapper/node1osd1       926.5G     21.0G      905.5G    2%            73%
/dev/mapper/node1osd2       926.5G     28.2G      898.3G    3%            64%
```

추가 리소스

- **vdostats(8)** 도움말 페이지.

2장. VDO 유지 관리

VDO 볼륨을 배포한 후 특정 작업을 수행하여 유지 관리 또는 최적화할 수 있습니다. 다음 작업 중 일부는 VDO 볼륨의 올바른 작동에 필요합니다.

사전 요구 사항

- VDO가 설치 및 배포됨. 1장. *VDO 배포*을 참조하십시오.

2.1. VDO 볼륨에서 여유 공간 관리

VDO는 씬 프로비저닝된 블록 스토리지 대상입니다. 이로 인해 VDO 볼륨에서 공간 사용량을 적극적으로 모니터링하고 관리해야 합니다.

2.1.1. VDO 볼륨의 물리 및 논리 크기

이 섹션에서는 VDO가 활용할 수 있는 물리적 크기, 사용 가능한 물리적 크기 및 VDO에 대해 설명합니다.

물리 크기

이 크기는 기본 블록 장치와 동일합니다. VDO는 다음을 위해 이 스토리지를 사용합니다.

- 중복 제거 및 압축 가능한 사용자 데이터
- UDS 인덱스와 같은 VDO 메타데이터

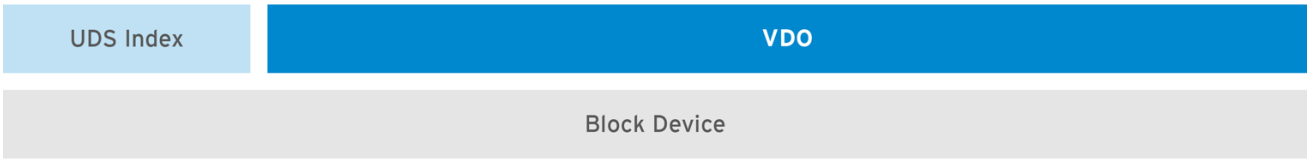
사용 가능한 물리 크기

이것은 VDO가 사용자 데이터에 사용할 수있는 물리적 크기의 부분입니다.
이 값은 물리 크기에서 메타데이터 크기를 뺀 값과 같고, 볼륨을 slab 크기로 분할한 후 나머지 부분을 slab 크기로 나눕니다.

논리 크기

이는 VDO 볼륨이 애플리케이션에 제공하는 프로비저닝된 크기입니다. 일반적으로 사용 가능한 실제 크기보다 큽니다. vdo **LogicalSize** 옵션이 지정되지 않은 경우 이제 논리 볼륨의 프로비저닝이 **1:1** 비율로 프로비저닝됩니다. 예를 들어, VDO 볼륨이 20GB 블록 장치에 배치된 경우 2.5GB는 UDS 인덱스용으로 예약됩니다(기본 인덱스 크기가 사용되는 경우). 나머지 17.5GB는 VDO 메타데이터 및 사용자 데이터에 대해 제공됩니다. 따라서 사용할 수 있는 스토리지가 17.5GB를 넘지 않으며 실제 VDO 볼륨을 구성하는 메타데이터로 인해 더 적을 수 있습니다.
VDO는 현재 절대 최대 논리 크기가 4PB인 물리 볼륨의 최대 254배의 논리 크기를 지원합니다.

그림 2.1. VDO 디스크 조직



RHEL_466924_0218

이 그림에서 VDO 중복 제거 스토리지 대상은 블록 장치에 완전히 배치되므로 VDO 볼륨의 물리적 크기가 기본 블록 장치와 동일한 크기입니다.

추가 리소스

- VDO 메타데이터가 다양한 크기의 블록 장치에 필요한 스토리지 VDO 메타데이터에 대한 자세한 내용은 1.6.4절. "물리 크기에 따른 VDO 요구 사항의 예" 을 참조하십시오.

2.1.2. VDO의 씬 프로비저닝

VDO는 씬 프로비저닝된 블록 스토리지 대상입니다. VDO 볼륨에서 사용하는 물리 공간의 양은 스토리지 사용자에게 표시되는 볼륨 크기와 다를 수 있습니다. 이러한 차이를 활용하여 스토리지 비용을 절감할 수 있습니다.

공간 외 조건

작성한 데이터가 예상 최적화 속도를 달성하지 못하는 경우 예기치 않게 스토리지 공간이 부족하지 않도록 주의하십시오.

논리 블록(가상 스토리지) 수가 물리적 블록(실제 저장소) 수를 초과할 때마다 파일 시스템 및 애플리케이션이 예기치 않게 공간이 부족해질 수 있습니다. 따라서 VDO를 사용하는 스토리지 시스템은 VDO 볼륨에서 사용 가능한 풀의 크기를 모니터링하는 방법을 제공해야 합니다.

vdostats 유틸리티를 사용하여 이 사용 가능한 풀의 크기를 확인할 수 있습니다. 이 유틸리티의 기본 출력에는 Linux **df** 유틸리티와 유사한 형식으로 실행되는 모든 VDO 볼륨에 대한 정보가 나열됩니다. 예를 들면 다음과 같습니다.

```
Device                1K-blocks  Used       Available  Use%
/dev/mapper/vdo-name  211812352  105906176  105906176  50%
```

VDO 볼륨의 물리적 스토리지 용량이 거의 가득 차면 VDO는 다음과 유사하게 시스템 로그에 경고를 보고합니다.

```
Oct  2 17:13:39 system lvm[13863]: Monitoring VDO pool vdo-name.
Oct  2 17:27:39 system lvm[13863]: WARNING: VDO pool vdo-name is now 80.69% full.
Oct  2 17:28:19 system lvm[13863]: WARNING: VDO pool vdo-name is now 85.25% full.
Oct  2 17:29:39 system lvm[13863]: WARNING: VDO pool vdo-name is now 90.64% full.
Oct  2 17:30:29 system lvm[13863]: WARNING: VDO pool vdo-name is now 96.07% full.
```

참고

이 경고 메시지는 **lvm2-monitor** 서비스가 실행 중인 경우에만 표시됩니다. 기본적으로 활성화되어 있습니다.

공간 부족 조건을 방지하는 방법

사용 가능한 풀 크기가 특정 수준 아래로 떨어지면 다음을 수행하여 작업을 수행할 수 있습니다.

- 데이터 삭제. 그러면 삭제된 데이터가 중복되지 않을 때마다 공간을 회수합니다. 데이터를 삭제하면 삭제가 발생한 후에만 공간을 사용할 수 있습니다.
- 물리적 스토리지 추가

중요

VDO 볼륨의 물리적 공간을 모니터링하여 공간 부족 상황을 방지합니다. 물리적 블록이 없으면 최근에 작성되었으며 VDO 볼륨에 승인되지 않은 데이터가 손실될 수 있습니다.

씬 프로비저닝 및 TRIM 및 DISCARD 명령

씬 프로비저닝의 스토리지 절감 효과를 얻기 위해서는 물리적 스토리지 계층에서 데이터를 삭제할 시기를 알아야 합니다. 씬 프로비저닝된 스토리지로 작동하는 파일 시스템은 **TRIM** 또는 **DISCARD** 명령을 전송하여 논리적 블록이 더 이상 필요하지 않은 경우 스토리지 시스템에 알립니다.

TRIM 또는 **DISCARD** 명령을 전송하는 몇 가지 방법을 사용할 수 있습니다.

- **discard** 마운트 옵션을 사용하면 파일 시스템은 블록이 삭제될 때마다 이러한 명령을 보낼 수 있습니다.
- **fstrim** 과 같은 유틸리티를 사용하여 제어되는 방식으로 명령을 보낼 수 있습니다. 이러한 유틸리티는 파일 시스템에 사용되지 않는 논리 블록을 감지하고 **TRIM** 또는 **DISCARD** 명령 형식으로 정보를 스토리지 시스템에 전송합니다.

사용하지 않는 블록에서 **TRIM** 또는 **DISCARD** 를 사용할 필요는 VDO에만 국한되지 않습니다. 씬 프로비저닝된 모든 스토리지 시스템에는 동일한 과제가 있습니다.

2.1.3. VDO 모니터링

다음 절차에서는 VDO 볼륨에서 사용 및 효율성 정보를 얻는 방법을 설명합니다.

사전 요구 사항

- VDO 소프트웨어를 설치합니다. 1.7절. "VDO 설치"을 참조하십시오.

절차

- **vdostats** 유틸리티를 사용하여 VDO 볼륨에 대한 정보를 가져옵니다.

```
# vdostats --human-readable

Device                 1K-blocks    Used    Available    Use%    Space saving%
/dev/mapper/node1osd1    926.5G     21.0G    905.5G      2%      73%
/dev/mapper/node1osd2    926.5G     28.2G    898.3G      3%      64%
```

추가 리소스

- **vdostats(8)** 도움말 페이지.

2.1.4. 파일 시스템에서 VDO 공간 확보

이 절차에서는 파일 시스템을 호스팅하는 VDO 볼륨의 스토리지 공간을 회수합니다.

파일 시스템이 **DISCARD**,**TRIM** 또는 **UNMAP** 명령을 사용하여 블록이 사용 가능한 것으로 통신하지 않는 한 VDO는 공간을 회수할 수 없습니다.

절차

- VDO 볼륨의 파일 시스템이 삭제 작업을 지원하는 경우 활성화합니다. 5장. *사용하지 않는 블록 삭제*을 참조하십시오.
- **DISCARD**,**TRIM** 또는 **UNMAP** 를 사용하지 않는 파일 시스템의 경우 수동으로 사용 가능한 공간을 회수할 수 있습니다. 바이너리 0으로 구성된 파일을 저장하여 사용 가능한 공간을 채운 다음 해당 파일을 삭제합니다.

2.1.5. 파일 시스템없이 VDO의 공간 확보

이 절차에서는 VDO 볼륨에서 파일 시스템 없이 블록 스토리지 대상으로 사용되는 스토리지 공간을 회수합니다.

절차

- **blkdiscard** 유틸리티를 사용합니다.
  예를 들어 단일 VDO 볼륨은 LVM을 그 위에 배포하여 여러 하위 볼륨으로 나눌 수 있습니다. 논리 볼륨을 프로비저닝 해제하기 전에 **blkdiscard** 유틸리티를 사용하여 이전에 해당 논리 볼륨에서 사용한 공간을 확보합니다.

  LVM은 **REQ_DISCARD** 명령을 지원하며 공간을 확보하기 위해 적절한 논리 블록 주소에서 VDO에 요청을 전달합니다. 다른 볼륨 관리자를 사용하는 경우 **REQ_DISCARD** 또는 이에 상응하는 SCSI 장치의 경우 **UNMAP** 또는 ATA 장치의 경우 **TRIM** 을 지원해야 합니다.

추가 리소스

- **blkdiscard(8)** 도움말 페이지

2.1.6. 파이버 채널 또는 이더넷 네트워크에서 VDO의 공간 확보

이 절차에서는 LIO 또는 SCST와 같은 SCSI 대상 프레임워크를 사용하여 파이버 채널 스토리지 패브릭 또는 이더넷 네트워크의 호스트에 프로비저닝되는 VDO 볼륨(또는 볼륨의 일부)의 스토리지 공간을 회수합니다.

절차

- SCSI 이니시에이터는 **UNMAP** 명령을 사용하여 씬 프로비저닝된 스토리지 대상의 공간을 확보할 수 있지만 이 명령에 대한 지원을 알리도록 SCSI 대상 프레임워크를 구성해야 합니다. 일반적으로 이러한 볼륨에서 씬 프로비저닝을 활성화하여 수행합니다.
  다음 명령을 실행하여 Linux 기반 SCSI 이니시에이터에서 **UNMAP** 에 대한 지원을 확인합니다.

  ```
  # sg_vpd --page=0xb0 /dev/device
  ```

  출력에서 *최대 매핑 해제 LBA 개수* 값이 0보다 큰지 확인합니다.

2.2. VDO 볼륨 시작 또는 중지

지정된 VDO 볼륨 또는 모든 VDO 볼륨 및 관련 UDS 색인을 시작하거나 중지할 수 있습니다.

2.2.1. 시작 및 활성화된 VDO 볼륨

시스템 부팅 중에 **vdo systemd** 장치는 *활성화* 로 구성된 모든 VDO 장치를 자동으로 *시작합니다.*

vdo systemd 장치는 **vdo** 패키지를 설치할 때 기본적으로 설치 및 활성화됩니다. 이 장치는 시스템 시작 시 **vdo start --all** 명령을 자동으로 실행하여 활성화된 모든 VDO 볼륨을 가져옵니다.

vdo create 명령에 --activate=disabled 옵션을 추가하여 자동으로 시작되지 않는 VDO 볼륨을 생성할 수도 있습니다.

시작 순서

일부 시스템에서는 VDO 볼륨 위와 그 아래에 LVM 볼륨을 배치할 수 있습니다. 이러한 시스템에서는 서비스를 올바른 순서로 시작해야 합니다.

1. LVM의 하부 계층이 먼저 시작되어야 합니다. 대부분의 시스템에서 이 계층을 시작하는 것은 LVM 패키지가 설치될 때 자동으로 구성됩니다.
2. 그러면 **vdo systemd** 유닛이 시작되어야 합니다.
3. 마지막으로 실행 중인 VDO 볼륨 위에서 LVM 볼륨 또는 기타 서비스를 시작하려면 추가 스크립트를 실행해야 합니다.

볼륨을 중지하는 데 걸리는 시간

VDO 볼륨을 중지하려면 스토리지 장치의 속도와 볼륨에서 쓰기에 필요한 데이터 양에 따라 시간이 걸립니다.

- 볼륨은 UDS 인덱스의 1GiB마다 항상 1GiB를 씁니다.
- 볼륨은 블록 맵 캐시 크기와 slab당 최대 8MiB에 해당하는 데이터 양을 추가로 씁니다.
- 볼륨은 미해결된 모든 IO 요청 처리를 완료해야 합니다.

2.2.2. VDO 볼륨 시작

이 절차는 주어진 VDO 볼륨 또는 시스템의 모든 VDO 볼륨을 시작합니다.

절차

- 지정된 VDO 볼륨을 시작하려면 다음을 사용합니다.

```
# vdo start --name=my-vdo
```

- 모든 VDO 볼륨을 시작하려면 다음을 사용합니다.

```
# vdo start --all
```

추가 리소스

- **vdo(8)** 도움말 페이지

2.2.3. VDO 볼륨 중지

이 절차에서는 주어진 VDO 볼륨 또는 시스템의 모든 VDO 볼륨을 중지합니다.

절차

1. 볼륨을 중지합니다.
   - 지정된 VDO 볼륨을 중지하려면 다음을 사용합니다.

```
# vdo stop --name=my-vdo
```

   - 모든 VDO 볼륨을 중지하려면 다음을 사용합니다.

```
# vdo stop --all
```

2. 볼륨이 디스크에 대한 데이터 쓰기를 완료할 때까지 기다립니다.

추가 리소스

- **vdo(8)** 도움말 페이지

2.2.4. 추가 리소스

- 불명확한 종료 후 다시 시작하면 VDO는 다시 빌드하여 메타데이터의 일관성을 확인하고 필요한 경우 이를 복구합니다. 재빌드 프로세스에 대한 자세한 내용은 2.5절. "불완전한 종료 후 VDO 볼륨 복구" 을 참조하십시오.

2.3. 시스템 부팅 시 VDO 볼륨 자동 시작

VDO 볼륨이 시스템 부팅 시 자동으로 시작되도록 구성할 수 있습니다. 자동 시작을 비활성화할 수도 있습니다.

2.3.1. 시작 및 활성화된 VDO 볼륨

시스템 부팅 중에 **vdo systemd** 장치는 *활성화* 로 구성된 모든 VDO 장치를 자동으로 *시작합니다.*

vdo systemd 장치는 **vdo** 패키지를 설치할 때 기본적으로 설치 및 활성화됩니다. 이 장치는 시스템 시작 시 **vdo start --all** 명령을 자동으로 실행하여 활성화된 모든 VDO 볼륨을 가져옵니다.

vdo create 명령에 --activate=disabled 옵션을 추가하여 자동으로 시작되지 않는 VDO 볼륨을 생성할 수도 있습니다.

시작 순서

일부 시스템에서는 VDO 볼륨 위와 그 아래에 LVM 볼륨을 배치할 수 있습니다. 이러한 시스템에서는 서비스를 올바른 순서로 시작해야 합니다.

1. LVM의 하부 계층이 먼저 시작되어야 합니다. 대부분의 시스템에서 이 계층을 시작하는 것은 LVM 패키지가 설치될 때 자동으로 구성됩니다.
2. 그러면 **vdo systemd** 유닛이 시작되어야 합니다.
3. 마지막으로 실행 중인 VDO 볼륨 위에서 LVM 볼륨 또는 기타 서비스를 시작하려면 추가 스크립트를 실행해야 합니다.

볼륨을 중지하는 데 걸리는 시간

VDO 볼륨을 중지하려면 스토리지 장치의 속도와 볼륨에서 쓰기에 필요한 데이터 양에 따라 시간이 걸립니다.

- 볼륨은 UDS 인덱스의 1GiB마다 항상 1GiB를 씁니다.
- 볼륨은 블록 맵 캐시 크기와 slab당 최대 8MiB에 해당하는 데이터 양을 추가로 씁니다.
- 볼륨은 미해결된 모든 IO 요청 처리를 완료해야 합니다.

2.3.2. VDO 볼륨 활성화

이 절차에서는 VDO 볼륨을 활성화하여 자동으로 시작되도록 활성화합니다.

절차

- 특정 볼륨을 활성화하려면 다음을 수행합니다.

```
# vdo activate --name=my-vdo
```

- 모든 볼륨을 활성화하려면 다음을 수행합니다.

```
# vdo activate --all
```

추가 리소스

- **vdo(8)** 도움말 페이지

2.3.3. VDO 볼륨 비활성화

이 절차에서는 VDO 볼륨이 자동으로 시작되지 않도록 비활성화합니다.

절차

- 특정 볼륨을 비활성화하려면 다음을 수행합니다.

```
# vdo deactivate --name=my-vdo
```

- 모든 볼륨을 비활성화하려면 다음을 수행합니다.

```
# vdo deactivate --all
```

추가 리소스

- **vdo(8)** 도움말 페이지

2.4. VDO 쓰기 모드 선택

기본 블록 장치에 필요한 내용에 따라 VDO 볼륨에 대한 쓰기 모드를 구성할 수 있습니다. 기본적으로 VDO는 쓰기 모드를 자동으로 선택합니다.

2.4.1. VDO 쓰기 모드

VDO는 다음과 같은 쓰기 모드를 지원합니다.

sync

VDO가 **동기화** 모드인 경우 위의 계층에서는 쓰기 명령이 영구 스토리지에 데이터를 쓰는 것으로 가정합니다. 따라서 파일 시스템 또는 애플리케이션에서 FLUSH를 발행하거나 장치 액세스(FUA) 요청을 강제 실행하여 데이터가 중요한 시점에 지속되도록 할 필요가 없습니다.
기본 스토리지를 통해 쓰기 명령이 완료되면 영구 스토리지에 데이터가 기록되도록 보장하는 경우에만 VDO를 **동기화** 모드로 설정해야 합니다. 즉, 스토리지에 휘발성 쓰기 캐시가 없거나 캐시를 통한 쓰기가 있어야 합니다.

async

VDO가 **비동기** 모드인 경우 쓰기 명령이 승인될 때 VDO는 데이터가 영구 스토리지에 기록되도록 보장하지 않습니다. 파일 시스템 또는 애플리케이션은 각 트랜잭션의 중요한 시점에서 데이터 지속성을 보장하기 위해 FLUSH 또는 FujiA 요청을 발행해야 합니다.
기본 스토리지에서 쓰기 명령이 완료되면 데이터가 영구 스토리지에 기록되도록 보장하지 않는 경우 VDO를 **async** 모드로 설정해야 합니다. 즉 스토리지에 휘발성 쓰기 캐시가 있는 경우 VDO를 설정해야 합니다.

async-unsafe

이 모드에는 **async** 와 동일한 속성이 있지만 Atomicity, Consistency, Isolation, Durability(ACID)와 호환되지 않습니다. **비동기** 와 비교할 때 **비동기-비보안은** 더 나은 성능을 제공합니다.

주의

ACID 준수를 가정하는 애플리케이션 또는 파일 시스템이 VDO 볼륨 위에서 작동하는 경우 **async-unsafe** 모드에서는 예기치 않은 데이터 손실이 발생할 수 있습니다.

auto

자동 모드에서는 각 장치의 특성에 따라 **동기화** 또는 **비동기** 를 자동으로 선택합니다. 기본 옵션입니다.

2.4.2. VDO 쓰기 모드의 내부 처리

이 섹션에서는 **동기화** 및 **비동기** VDO 쓰기 모드가 작동하는 방법에 대해 자세히 설명합니다.

kvdo 모듈이 동기 모드에서 작동하는 경우:

1. 할당된 블록에 요청에 데이터를 임시로 쓴 다음, 요청을 승인합니다.
2. 승인이 완료되면 VDO 인덱스로 전송되는 블록 데이터의 MurmurHash-3 서명을 컴퓨팅하여 블록을 삭제하려고 합니다.
3. VDO 인덱스에 동일한 서명이 있는 블록에 대한 항목이 포함된 경우 **kvdo** 는 표시된 블록을 읽고 두 블록의 바이트 단위 비교를 수행하여 동일한지 확인합니다.
4. 실제로 동일한 경우 **kvdo** 는 논리 블록이 해당 물리적 블록을 가리키고 할당된 실제 블록을 해제하도록 블록 맵을 업데이트합니다.
5. VDO 색인에 작성 중인 블록의 서명에 대한 항목이 없거나 표시된 블록에 실제로 동일한 데이터가 포함되어 있지 않은 경우 **kvdo** 는 해당 블록 맵을 업데이트하여 임시 물리적 블록을 영구적으로 만듭니다.

kvdo 가 비동기 모드에서 작동하는 경우:

1. 데이터를 작성하지 않고 즉시 요청을 승인합니다.
2. 그러면 위에 설명된 것과 동일한 방식으로 블록의 중복 제거를 시도합니다.

3. 블록이 중복된 것으로 판단되면 **kvdo** 는 해당 블록 맵을 업데이트하고 할당된 블록을 해제합니다. 그렇지 않으면 요청의 데이터를 할당된 블록에 쓰고 블록 맵을 업데이트하여 실제 블록을 영구적으로 만듭니다.

2.4.3. VDO 볼륨에서 쓰기 모드 확인

이 절차에서는 선택한 VDO 볼륨의 활성 쓰기 모드를 나열합니다.

절차

- 다음 명령을 사용하여 VDO 볼륨에서 사용하는 쓰기 모드를 확인합니다.

```
# vdo status --name=my-vdo
```

출력 목록은 다음과 같습니다.

  - **sync**,**async** 또는 **auto**에서 선택한 옵션인 *구성된 쓰기 정책*
  - **동기화** 또는 **비동기**인 VDO가 적용된 특정 쓰기 모드인 쓰기 *정책*

2.4.4. 휘발성 캐시 확인

이 절차에서는 블록 장치에 휘발성 캐시가 있는지 여부를 확인합니다. 정보를 사용하여 **비동기** VDO 쓰기 모드 **중** 하나를 선택할 수 있습니다.

절차

1. 다음 방법 중 하나를 사용하여 장치에 writeback 캐시가 있는지 확인합니다.
   - **/sys/block/*block-device* /*device*/scsi_disk/*identifier*/cache_type sysfs** 파일을 읽습니다. 예를 들면 다음과 같습니다.

```
$ cat '/sys/block/sda/device/scsi_disk/7:0:0:0/cache_type'

write back
```

```
$ cat '/sys/block/sdb/device/scsi_disk/1:2:0:0/cache_type'

None
```

   - 또는 위에서 언급한 장치에 쓰기 캐시가 있는지 또는 커널 부팅 로그에 없는지 여부를 확인할 수 있습니다.

```
sd 7:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO or
FUA
sd 1:2:0:0: [sdb] Write cache: disabled, read cache: disabled, supports DPO and FUA
```

2. 이전 예에서 다음을 수행합니다.
   - Devices **da** 는 writeback 캐시 *가* 있음을 나타냅니다. 이 모드에 **비동기** 모드를 사용합니다.
   - **sdb** 장치는 writeback 캐시 *가 없음을* 나타냅니다. **동기화** 모드를 사용합니다.

cache_type 값이 **None** 이거나 동시 쓰기 모드인 경우 **동기화** 쓰기 모드를 사용하도록 VDO 를 구성해야 합니다.

2.4.5. VDO 쓰기 모드 설정

이 절차에서는 기존 볼륨 또는 새 볼륨을 생성할 때 VDO 볼륨의 쓰기 모드를 설정합니다.

중요

잘못된 쓰기 모드를 사용하면 정전, 시스템 충돌 또는 디스크와의 예기치 않은 연결이 손실될 수 있습니다.

사전 요구 사항

- 장치에 올바른 쓰기 모드를 확인합니다. 2.4.4절. “휘발성 캐시 확인”을 참조하십시오.

절차

- 기존 VDO 볼륨에서 또는 새 볼륨을 생성할 때 쓰기 모드를 설정할 수 있습니다.
  - 기존 VDO 볼륨을 수정하려면 다음을 사용합니다.

```
# vdo changeWritePolicy --writePolicy=sync|async|async-unsafe|auto \
                        --name=vdo-name
```

  - VDO 볼륨을 생성할 때 쓰기 모드를 지정하려면 **--writePolicy=*sync*|*async*|*async-unsafe*|*auto*** 옵션을 **vdo create** 명령에 추가합니다.

2.5. 불완전한 종료 후 VDO 볼륨 복구

불완전한 종료 후 VDO 볼륨을 복구하여 작동을 계속할 수 있습니다. 이 작업은 대부분 자동화된 작업입니다. 또한 프로세스의 실패로 인해 VDO 볼륨이 성공적으로 생성되지 않은 후 정리할 수 있습니다.

2.5.1. VDO 쓰기 모드

VDO는 다음과 같은 쓰기 모드를 지원합니다.

sync

VDO가 **동기화** 모드인 경우 위의 계층에서는 쓰기 명령이 영구 스토리지에 데이터를 쓰는 것으로 가정합니다. 따라서 파일 시스템 또는 애플리케이션에서 FLUSH를 발행하거나 장치 액세스(FUA) 요청을 강제 실행하여 데이터가 중요한 시점에 지속되도록 할 필요가 없습니다.
기본 스토리지를 통해 쓰기 명령이 완료되면 영구 스토리지에 데이터가 기록되도록 보장하는 경우에만 VDO를 **동기화** 모드로 설정해야 합니다. 즉, 스토리지에 휘발성 쓰기 캐시가 없거나 캐시를 통한 쓰기가 있어야 합니다.

async

VDO가 **비동기** 모드인 경우 쓰기 명령이 승인될 때 VDO는 데이터가 영구 스토리지에 기록되도록 보장하지 않습니다. 파일 시스템 또는 애플리케이션은 각 트랜잭션의 중요한 시점에서 데이터 지속성을 보장하기 위해 FLUSH 또는 FujiA 요청을 발행해야 합니다.
기본 스토리지에서 쓰기 명령이 완료되면 데이터가 영구 스토리지에 기록되도록 보장하지 않는 경우 VDO를 **async** 모드로 설정해야 합니다. 즉 스토리지에 휘발성 쓰기 캐시가 있는 경우 VDO를 설정해야 합니다.

async-unsafe

이 모드에는 **async** 와 동일한 속성이 있지만 Atomicity, Consistency, Isolation, Durability(ACID)와 호환되지 않습니다. **비동기** 와 비교할 때 **비동기-비보안은** 더 나은 성능을 제공합니다.

주의

ACID 준수를 가정하는 애플리케이션 또는 파일 시스템이 VDO 볼륨 위에서 작동하는 경우 **async-unsafe** 모드에서는 예기치 않은 데이터 손실이 발생할 수 있습니다.

auto

자동 모드에서는 각 장치의 특성에 따라 **동기화** 또는 **비동기** 를 자동으로 선택합니다. 기본 옵션입니다.

2.5.2. VDO 볼륨 복구

VDO 볼륨이 불명확한 종료 후 다시 시작되면 VDO는 다음 작업을 수행합니다.

- 볼륨의 메타데이터의 일관성을 확인합니다.
- 필요한 경우 메타데이터 부분을 다시 빌드하여 복구합니다.

재구축은 자동이며 사용자 개입이 필요하지 않습니다.

VDO는 활성 쓰기 모드에 따라 다른 쓰기를 다시 빌드할 수 있습니다.

sync

VDO가 동기 스토리지에서 실행 중이고 쓰기 정책이 **동기화** 로 설정된 경우 볼륨에 기록된 모든 데이터가 완전히 복구됩니다.

async

쓰기 정책이 **async** 인 경우 지속되지 않은 경우 일부 쓰기가 복구되지 않을 수 있습니다. 이 작업은 VDO를 **FLUSH** 명령 또는 kubeA (force Unit access) 플래그로 태그가 지정된 쓰기 I/O를 전송하여 수행됩니다. fsync, **fdatasync** ,**sync** 또는 **umount** 와 같은 데이터 무결성 작업을 호출하여 사용자 모드에서 이 작업을 수행할 수 있습니다.

두 모드에서는 승인되지 않았거나 플러시되지 않은 일부 쓰기도 다시 빌드될 수 있습니다.

자동 및 수동 복구

VDO 볼륨이 **운영** 모드로 전환되면 VDO는 다시 온라인 상태가 되면 불명확한 VDO 볼륨을 자동으로 다시 빌드합니다. 이것을 *온라인 복구* 라고 합니다.

VDO 볼륨을 성공적으로 복구할 수 없는 경우 볼륨 재시작 시 지속되는 **읽기 전용** 운영 모드로 볼륨을 배치합니다. 다시 빌드를 강제 적용하여 문제를 수동으로 해결해야 합니다.

추가 리소스

- 자동 및 수동 복구 및 VDO 운영 모드에 대한 자세한 내용은 2.5.3절. "VDO 운영 모드" 을 참조하십시오.

2.5.3. VDO 운영 모드

이 섹션에서는 VDO 볼륨이 정상적으로 작동하는지 또는 오류를 복구하고 있는지를 나타내는 모드를 설명합니다.

vdostats --verbose *device* 명령을 사용하여 VDO 볼륨의 현재 운영 모드를 표시할 수 있습니다. 출력에서 *Operating mode* 특성을 참조하십시오.

normal

기본 운영 모드입니다. 다음 상태 중 하나가 다른 모드를 강제 적용하지 않는 한 VDO 볼륨은 항상 **일반** 모드입니다. 새로 생성된 VDO 볼륨은 **일반** 모드에서 시작됩니다.

복구 중

VDO 볼륨이 종료되기 전에 모든 메타데이터를 저장하지 않으면 다음에 시작될 때 자동으로 **복구** 모드로 전환됩니다. 이 모드로 전환하는 일반적인 이유는 갑작스러운 정전 또는 기본 스토리지 장치에서 발생하는 문제입니다.
복구 모드에서 VDO는 장치의 각 데이터 블록에 대한 참조 수를 수정합니다. 복구에는 일반적으로 시간이 오래 걸리지 않습니다. 시간은 VDO 볼륨의 크기, 기본 스토리지 장치의 속도 및 VDO가 동시에 처리하는 기타 요청 수에 따라 달라집니다. VDO 볼륨은 일반적으로 다음과 같은 예외와 함께 작동합니다.

- 처음에는 볼륨에 쓰기 요청에 사용할 수 있는 공간이 제한될 수 있습니다. 메타데이터가 많으면 더 많은 여유 공간을 사용할 수 있게 됩니다.
- VDO 볼륨을 복구하는 동안 작성된 데이터는 해당 데이터가 아직 복구되지 않은 볼륨의 일부인 경우 충돌 이전에 작성된 데이터에 중복 제거되지 않을 수 있습니다. VDO는 볼륨을 복구하는 동안 데이터를 압축할 수 있습니다. 압축된 블록을 계속 읽거나 덮어쓸 수 있습니다.
- 온라인 복구 중에 특정 통계(예: 사용 중인 블록 및 *사용 가능한 블록)를 사용할* 수 없습니다 . 이러한 통계는 재빌드가 완료되면 사용할 수 있습니다.
- 지속적인 복구 작업으로 인해 읽기 및 쓰기에 대한 응답 시간이 일반보다 느릴 수 있습니다.

복구 모드에서 VDO 볼륨을 안전하게 종료할 수 있습니다. 복구가 종료되기 전에 완료되지 않으면 다음에 시작될 때 장치가 **복구** 모드로 다시 시작됩니다.

VDO 볼륨은 **복구** 모드를 자동으로 종료하고 모든 참조 수를 수정하면 **일반** 모드로 이동합니다. 관리자 작업이 필요하지 않습니다. 자세한 내용은 2.5.4절. “온라인에서 VDO 볼륨 복구” 의 내용을 참조하십시오.

읽기 전용

VDO 볼륨이 치명적인 내부 오류가 발생하면 **읽기 전용** 모드로 전환됩니다. **읽기 전용** 모드를 유발할 수 있는 이벤트에는 메타데이터 손상 또는 백업 스토리지 장치가 읽기 전용으로 표시됩니다. 이 모드는 오류 상태입니다.
읽기 전용 모드에서는 데이터가 정상적으로 작동하지만 데이터 쓰기는 항상 실패합니다. VDO 볼륨은 관리자가 문제를 해결할 때까지 **읽기 전용** 모드로 유지됩니다.

VDO 볼륨을 **읽기 전용** 모드로 안전하게 종료할 수 있습니다. 이 모드는 일반적으로 VDO 볼륨을 다시 시작한 후에 유지됩니다. 드물지만 VDO 볼륨은 백업 스토리지 장치에 **읽기 전용** 상태를 기록할 수 없습니다. 이 경우 VDO는 복구를 대신 시도합니다.

볼륨이 읽기 전용 모드이면 볼륨의 데이터가 손실되거나 손상되지 않았음을 보장할 수 없습니다. 이러한 경우 Red Hat은 읽기 전용 볼륨에서 데이터를 복사하고 백업에서 볼륨을 복원할 것을 권장합니다.

데이터 손상 위험이 허용되는 경우 볼륨을 온라인으로 다시 가져와 사용할 수 있도록 VDO 볼륨 메타데이터를 오프라인으로 다시 빌드할 수 있습니다. 재구축된 데이터의 무결성은 보장할 수 없습니다. 자세한 내용은 2.5.5절. "VDO 볼륨 메타데이터의 오프라인 다시 빌드 강제" 의 내용을 참조하십시오.

2.5.4. 온라인에서 VDO 볼륨 복구

이 절차에서는 불명확한 종료 후 메타데이터를 복구하기 위해 VDO 볼륨에서 온라인 복구를 수행합니다.

절차

1. VDO 볼륨이 아직 시작되지 않은 경우 시작합니다.

```
# vdo start --name=my-vdo
```

   추가 단계가 필요하지 않습니다. 복구는 백그라운드에서 실행됩니다.

2. 사용 중인 블록과 *블록과 같은 볼륨 통계를 자유롭게 사용하는* 경우 사용할 수 있을 때까지 기다립니다.

2.5.5. VDO 볼륨 메타데이터의 오프라인 다시 빌드 강제

이 절차에서는 불명확한 종료 후 복구하기 위해 VDO 볼륨 메타데이터를 강제로 오프라인으로 다시 빌드합니다.

주의

이 절차에서는 볼륨에서 데이터가 손실될 수 있습니다.

사전 요구 사항

- VDO 볼륨이 시작됩니다.

절차

1. 볼륨이 읽기 전용 모드인지 확인합니다. 명령 출력에서 *operating mode* 속성을 확인합니다.

```
# vdo status --name=my-vdo
```

   볼륨이 읽기 전용 모드가 아닌 경우 오프라인 다시 빌드를 강제 수행할 필요가 없습니다. 2.5.4절. "온라인에서 VDO 볼륨 복구" 에 설명된 대로 온라인 복구를 수행합니다.

2. 볼륨이 실행 중인 경우 해당 볼륨을 중지합니다.

```
# vdo stop --name=my-vdo
```

3. **--forceRebuild** 옵션을 사용하여 볼륨을 다시 시작합니다.

```
# vdo start --name=my-vdo --forceRebuild
```

2.5.6. 성공적으로 생성되지 않은 VDO 볼륨 제거

이 절차에서는 중간 상태에서 VDO 볼륨을 정리합니다. 볼륨 생성 시 오류가 발생하는 경우 볼륨이 중간 상태로 유지됩니다. 예를 들면 다음과 같은 경우에 발생할 수 있습니다.

- 시스템이 충돌합니다
- 전원 실패
- 관리자는 실행 중인 **vdo create 명령을 중단합니다**.

절차

- 정리하려면 **--force** 옵션을 사용하여 성공적으로 생성되지 않은 볼륨을 제거합니다.

```
# vdo remove --force --name=my-vdo
```

볼륨이 성공적으로 생성되지 않았기 때문에 관리자가 시스템 구성을 변경하여 충돌이 발생할 수 있으므로 **--force** 옵션이 필요합니다.

--force 옵션을 사용하지 않으면 **vdo remove** 명령이 실패하고 다음 메시지가 표시됩니다.

```
[...]
A previous operation failed.
Recovery from the failure either failed or was interrupted.
Add '--force' to 'remove' to perform the following cleanup.
Steps to clean up VDO my-vdo:
umount -f /dev/mapper/my-vdo
udevadm settle
dmsetup remove my-vdo
vdo: ERROR - VDO volume my-vdo previous operation (create) is incomplete
```

2.6. UDS 인덱스 최적화

UDS 인덱스의 특정 설정을 구성하여 시스템에서 최적화할 수 있습니다.

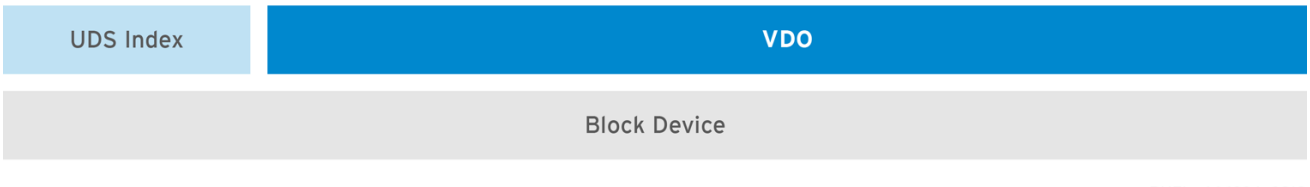
중요

VDO 볼륨을 생성한 *후에*는 UDS 인덱스의 속성을 변경할 수 없습니다.

2.6.1. VDO 볼륨의 구성 요소

VDO는 블록 장치를 백업 저장소로 사용하며, 이 저장소에는 하나 이상의 디스크, 파티션 또는 플랫 파일로 구성된 물리적 스토리지 집계가 포함될 수 있습니다. 스토리지 관리 툴에서 VDO 볼륨을 생성할 때 VDO는 UDS 인덱스 및 VDO 볼륨을 위한 볼륨 공간을 예약합니다. UDS 인덱스와 VDO 볼륨은 중복 제거 블록 스토리지를 제공하기 위해 상호 작용합니다.

그림 2.2. VDO 디스크 조직



VDO 솔루션은 다음 구성 요소로 구성됩니다.

kvdo

Linux 장치 매퍼 계층으로 로드되는 커널 모듈은 중복 제거, 압축 및 씬 프로비저닝된 블록 스토리지 볼륨을 제공합니다.
kvdo 모듈은 블록 장치를 노출합니다. 블록 스토리지에 대해 이 블록 장치에 직접 액세스하거나 XFS 또는 ext4와 같은 Linux 파일 시스템을 통해 제공할 수 있습니다.

kvdo 에서 VDO 볼륨의 논리적 데이터 블록을 읽도록 요청을 받으면 요청된 논리 블록을 기본 물리 블록에 매핑한 다음 요청된 데이터를 읽고 반환합니다.

kvdo 에서 VDO 볼륨에 데이터 블록을 쓰기 요청을 수신하면 먼저 요청이 DISCARD 또는 TRIM 요청인지 또는 데이터가 균일하게 0인지 여부를 확인합니다. 이러한 조건 중 하나가 true이면 **kvdo** 는 블록 맵을 업데이트하고 요청을 승인합니다. 그렇지 않으면 VDO는 데이터를 처리하고 최적화합니다.

UDS

볼륨에서 UDS(Universal Deduplication Service) 인덱스와 통신하고 중복 데이터를 분석하는 커널 모듈. 새로운 데이터 각각에 대해 UDS는 이전에 저장된 데이터와 동일한지 신속하게 판별합니다. 인덱스가 일치하는 항목을 발견하면 스토리지 시스템은 기존 항목을 내부적으로 참조하여 동일한 정보를 두 번 이상 저장하지 않도록 할 수 있습니다.
UDS 인덱스는 **uds** 커널 모듈로 커널 내부에서 실행됩니다.

명령행 도구

최적화된 스토리지 구성 및 관리.

2.6.2. UDS 인덱스

VDO는 UDS라는 고성능 중복 제거 인덱스를 사용하여 저장되는 데이터 블록을 감지합니다.

UDS 인덱스는 VDO 제품의 토대를 제공합니다. 새 데이터 각각에 대해 이 데이터가 이전에 저장된 데이터와 동일한지 신속하게 판별합니다. 인덱스가 일치하면 스토리지 시스템은 기존 항목을 내부적으로 참조하여 동일한 정보를 두 번 이상 저장하지 않도록 할 수 있습니다.

UDS 인덱스는 **uds** 커널 모듈로 커널 내부에서 실행됩니다.

중복 제거 창은 인덱스가 기억하는 이전에 작성된 블록 수입니다. 중복 제거 창의 크기는 구성할 수 있습니다. 지정된 창 크기에 대해 인덱스에는 특정 용량의 RAM과 특정 디스크 공간이 필요합니다. 일반적으로 창 크기는 **--indexMem=size** 옵션을 사용하여 인덱스 메모리 크기를 지정하여 결정됩니다. 그런 다음 VDO는 자동으로 사용할 디스크 공간의 양을 결정합니다.

UDS 인덱스는 다음 두 부분으로 구성됩니다.

- 고유한 블록당 최대 하나의 항목이 포함된 메모리에서 압축 표현이 사용됩니다.
- 발생 시 인덱스에 제공되는 연결된 블록 이름을 순서대로 기록하는 디스크상의 구성 요소입니다.

UDS는 캐시를 포함하여 메모리의 항목당 평균 4바이트를 사용합니다.

디스크의 구성 요소는 UDS에 전달된 데이터의 바인딩된 기록을 유지합니다. UDS는 최근 표시된 블록의 이름을 포함하여 이 중복 제거 창에 속하는 데이터에 대한 중복 제거 지침을 제공합니다. 중복 제거 창에서 UDS는 데이터를 최대한 효율적으로 인덱싱하는 동시에 대용량 데이터 리포지토리에 필요한 메모리 양을 제한할 수 있습니다. 중복 제거 창의 경계된 특성에도 불구하고, 중복 제거 수준이 높은 대부분의 데이터 집합은 일시적인 지역성을 나타냈습니다. 즉, 대부분의 중복 제거는 거의 동시에 작성된 블록 간에 발생합니다. 또한 일반적으로 작성 중인 데이터는 오래 전에 작성된 데이터보다 최근에 작성된 데이터가 복제될 가능성이 높습니다. 따라서 지정된 시간 간격 동안 지정된 워크로드의 경우 중복 제거 속도는 가장 최근 데이터 또는 모든 데이터만 색인화하는지 여부와 동일한 경우가 많습니다.

중복 데이터는 시간적 로컬을 나타내기 때문에 스토리지 시스템의 모든 블록을 인덱싱하는 것은 거의 필요하지 않습니다. 그렇지 않은 경우 인덱스 메모리 비용이 중복 제거로 인한 스토리지 비용 절감을 넘어섭니다. 인덱스 크기 요구 사항은 데이터 입수 속도와 더 긴밀하게 연관되어 있습니다. 예를 들어 총 용량이 100TB이지만 입수 속도는 주당 1TB인 스토리지 시스템을 고려해 보십시오. 4TB의 중복 제거 창에서 UDS는 지난 달 기록된 데이터 중에서 대부분의 중복성을 감지할 수 있습니다.

2.6.3. 권장 UDS 인덱스 구성

이 섹션에서는 의도한 사용 사례에 따라 UDS 인덱스와 함께 사용할 권장 옵션에 대해 설명합니다.

일반적으로 Red Hat은 모든 운용 환경에 **스파스** UDS 인덱스를 사용하는 것이 좋습니다. 이는 매우 효율적인 인덱싱 데이터 구조로, 중복 제거 창에서 블록당 약 1바이트의 RAM이 필요합니다. 디스크의 경우 블록당 약 72바이트의 디스크 공간이 필요합니다. 이 인덱스의 최소 구성은 256MB의 RAM과 디스크의 약 25GB 공간을 사용합니다.

이 구성을 사용하려면 **vdo create** 명령에 **--sparseIndex=enabled --indexMem=0.25** 옵션을 지정합니다. 이 구성으로 인해 2.5TB의 중복 제거 창이 생성됩니다(즉 2.5TB 기록을 기억함). 대부분의 사용 사례에서 2.5TB의 중복 제거 창은 최대 10TB 크기의 스토리지 풀 중복 제거에 적합합니다.

그러나 인덱스의 기본 구성은 **밀도가 높은** 인덱스를 사용하는 것입니다. 이 인덱스는 RAM에서 효율성이 훨씬 낮지만 필요한 최소 디스크 공간을 10배 이상 낮으므로 제한된 환경에서 평가할 수 있습니다.

일반적으로 VDO 볼륨의 물리적 크기의 4분의 1에 해당하는 중복 제거 창은 권장 구성입니다. 그러나 이것은 실제 요구 사항이 아닙니다. 물리 스토리지 용량과 비교되는 작은 중복 제거 창도 많은 사용 사례에서 상당한 양의 중복 데이터를 찾을 수 있습니다. 큰 창을 사용할 수도 있지만, 대부분의 경우 추가 이점이 거의 없습니다.

추가 리소스

- 이 중요한 시스템 매개 변수 조정에 대한 추가 지침은 Red Hat 기술 계정 관리자 담당자에게 문의하십시오.

2.7. VDO에서 중복 제거 활성화 또는 비활성화

경우에 따라 볼륨에서 읽고 쓸 수 있는 기능을 유지하면서 VDO 볼륨에 작성되는 데이터의 중복 제거를 일시적으로 비활성화할 수 있습니다. 중복 제거를 비활성화하면 후속 쓰기가 중복 제거되지 않지만 이미 중복 제거된 데이터는 계속 유지됩니다.

2.7.1. VDO에서 중복 제거

중복 제거는 중복 블록 복사본을 여러 개 제거하여 스토리지 리소스의 사용을 줄이는 기술입니다.

동일한 데이터를 두 번 이상 쓰는 대신 VDO는 각 중복 블록을 탐지하여 원래 블록에 대한 참조로 기록합니다. VDO는 VDO의 스토리지 계층에서 VDO 아래의 스토리지 계층에서 사용하는 물리적 블록 주소에 대한 논리 블록 주소에서 매핑을 유지 관리합니다.

중복 제거 후 여러 개의 논리적 블록 주소를 동일한 물리적 블록 주소에 매핑할 수 있습니다. 이를 공유 블록이라고 합니다. 블록 공유는 VDO가 없는 것처럼 블록을 읽고 쓰는 스토리지 사용자에게 표시되지 않습니다.

공유 블록을 덮어쓰면 VDO는 새 블록 데이터를 저장하기 위해 새 실제 블록을 할당하여 공유 실제 블록에 매핑된 다른 논리적 블록 주소가 수정되지 않도록 합니다.

2.7.2. VDO 볼륨에서 중복 제거 활성화

이 절차에서는 연결된 UDS 인덱스를 다시 시작하고 중복 제거가 다시 활성 상태임을 알립니다.

참고

중복 제거는 기본적으로 활성화되어 있습니다.

절차

- VDO 볼륨에서 중복 제거를 다시 시작하려면 다음 명령을 사용합니다.

```
# vdo enableDeduplication --name=my-vdo
```

2.7.3. VDO 볼륨에서 중복 제거 비활성화

이 절차에서는 연결된 UDS 인덱스를 중지하고 중복 제거가 더 이상 활성 상태가 아닌 VDO 볼륨에 알립니다.

절차

- VDO 볼륨에서 중복 제거를 중지하려면 다음 명령을 사용합니다.

```
# vdo disableDeduplication --name=my-vdo
```

- **vdo create 명령에 --deduplication=disabled 옵션을 추가하여 새 VDO 볼륨을 생성할** 때 중복 제거를 비활성화할 수도 있습니다.

2.8. VDO에서 압축 활성화 또는 비활성화

VDO는 데이터 압축을 제공합니다. 성능을 극대화하거나 압축할 가능성이 없는 데이터 처리 속도를 높이기 위해 비활성화하거나 공간을 다시 활성화하여 공간을 절약할 수 있습니다.

2.8.1. VDO에 압축

VDO는 블록 수준 중복 제거와 함께 KubeOPS Compression™ 기술을 사용하여 인라인 블록 수준 압축을 제공합니다.

VDO 볼륨 압축은 기본적으로 에 있습니다.

중복 제거는 가상 시스템 환경과 백업 애플리케이션에 가장 적합한 솔루션이지만 압축은 일반적으로 로그 파일 및 데이터베이스와 같이 블록 수준 중복성을 나타내지 않는 구조화되고 구조화되지 않은 파일 형식과 매우 잘 작동합니다.

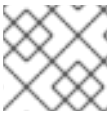
압축은 중복으로 확인되지 않은 블록에서 작동합니다. VDO는 처음으로 고유한 데이터를 볼 때 데이터를 압축합니다. 이미 저장된 데이터의 후속 복사본은 추가 압축 단계 없이 중복 제거됩니다.

압축 기능은 여러 압축 작업을 한 번에 처리할 수 있는 병렬화된 패키징 알고리즘을 기반으로 합니다. 먼저 블록을 저장하고 요청자에게 응답한 후, 압축 시 하나의 물리적 블록에 적합할 수 있는 여러 블록을 찾습니다. 특정 물리적 블록이 추가 압축 블록을 보유하지 않을 것으로 판단되면 스토리지에 기록되고 압축되지 않은 블록은 해제되고 재사용됩니다.

요청자에게 이미 응답한 후 압축 및 패키징 작업을 수행하면 압축을 사용하면 대기 시간이 거의 발생하지 않습니다.

2.8.2. VDO 볼륨에서 압축 활성화

이 절차를 통해 VDO 볼륨에서 압축하여 공간을 늘릴 수 있습니다.



참고

압축은 기본적으로 활성화되어 있습니다.

절차

- 다시 시작하려면 다음 명령을 사용합니다.

```
# vdo enableCompression --name=my-vdo
```

2.8.3. VDO 볼륨에서 압축 비활성화

이 절차에서는 성능을 최대화하거나 압축할 가능성이 없는 데이터를 신속하게 처리하기 위해 VDO 볼륨에서 압축을 중지합니다.

절차

- 기존 VDO 볼륨에서 압축을 중지하려면 다음 명령을 사용합니다.

```
# vdo disableCompression --name=my-vdo
```

- 또는 새 볼륨을 만들 때 **vdo create** 명령에 **--compression=disabled** 옵션을 추가하여 압축을 비활성화할 수 있습니다.

2.9. VDO 볼륨의 크기 증가

VDO 볼륨의 물리 크기를 늘리면 더 많은 기본 스토리지 용량 또는 논리 크기를 활용하여 볼륨에 더 많은 용량을 제공할 수 있습니다.

2.9.1. VDO 볼륨의 물리 및 논리 크기

이 섹션에서는 VDO가 활용할 수 있는 물리적 크기, 사용 가능한 물리적 크기 및 VDO에 대해 설명합니다.

물리 크기

이 크기는 기본 블록 장치와 동일합니다. VDO는 다음을 위해 이 스토리지를 사용합니다.

- 중복 제거 및 압축 가능한 사용자 데이터
- UDS 인덱스와 같은 VDO 메타데이터

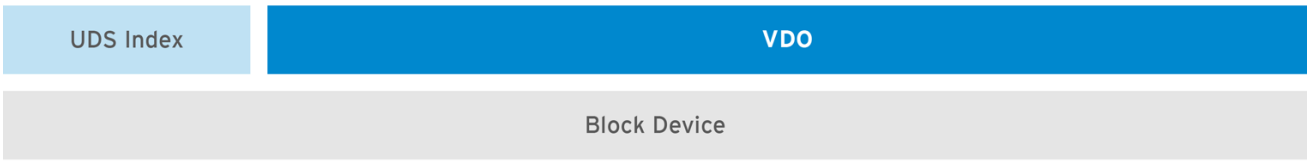
사용 가능한 물리 크기

이것은 VDO가 사용자 데이터에 사용할 수있는 물리적 크기의 부분입니다.
이 값은 물리 크기에서 메타데이터 크기를 뺀 값과 같고, 볼륨을 slab 크기로 분할한 후 나머지 부분을 slab 크기로 나눕니다.

논리 크기

이는 VDO 볼륨이 애플리케이션에 제공하는 프로비저닝된 크기입니다. 일반적으로 사용 가능한 실제 크기보다 큽니다. vdo **LogicalSize** 옵션이 지정되지 않은 경우 이제 논리 볼륨의 프로비저닝이 **1:1** 비율로 프로비저닝됩니다. 예를 들어, VDO 볼륨이 20GB 블록 장치에 배치된 경우 2.5GB는 UDS 인덱스용으로 예약됩니다(기본 인덱스 크기가 사용되는 경우). 나머지 17.5GB는 VDO 메타데이터 및 사용자 데이터에 대해 제공됩니다. 따라서 사용할 수 있는 스토리지가 17.5GB를 넘지 않으며 실제 VDO 볼륨을 구성하는 메타데이터로 인해 더 적을 수 있습니다.
VDO는 현재 절대 최대 논리 크기가 4PB인 물리 볼륨의 최대 254배의 논리 크기를 지원합니다.

그림 2.3. VDO 디스크 조직



RHEL_466924_0218

이 그림에서 VDO 중복 제거 스토리지 대상은 블록 장치에 완전히 배치되므로 VDO 볼륨의 물리적 크기가 기본 블록 장치와 동일한 크기입니다.

추가 리소스

- VDO 메타데이터가 다양한 크기의 블록 장치에 필요한 스토리지 VDO 메타데이터에 대한 자세한 내용은 1.6.4절. "물리 크기에 따른 VDO 요구 사항의 예" 을 참조하십시오.

2.9.2. VDO의 씬 프로비저닝

VDO는 씬 프로비저닝된 블록 스토리지 대상입니다. VDO 볼륨에서 사용하는 물리 공간의 양은 스토리지 사용자에게 표시되는 볼륨 크기와 다를 수 있습니다. 이러한 차이를 활용하여 스토리지 비용을 절감할 수 있습니다.

공간 외 조건

작성한 데이터가 예상 최적화 속도를 달성하지 못하는 경우 예기치 않게 스토리지 공간이 부족하지 않도록 주의하십시오.

논리 블록(가상 스토리지) 수가 물리적 블록(실제 저장소) 수를 초과할 때마다 파일 시스템 및 애플리케이션이 예기치 않게 공간이 부족해질 수 있습니다. 따라서 VDO를 사용하는 스토리지 시스템은 VDO 볼륨에서 사용 가능한 풀의 크기를 모니터링하는 방법을 제공해야 합니다.

vdostats 유틸리티를 사용하여 이 사용 가능한 풀의 크기를 확인할 수 있습니다. 이 유틸리티의 기본 출력에는 Linux **df** 유틸리티와 유사한 형식으로 실행되는 모든 VDO 볼륨에 대한 정보가 나열됩니다. 예를 들면 다음과 같습니다.

```
Device                  1K-blocks   Used       Available   Use%
/dev/mapper/vdo-name    211812352   105906176  105906176   50%
```

VDO 볼륨의 물리적 스토리지 용량이 거의 가득 차면 VDO는 다음과 유사하게 시스템 로그에 경고를 보고합니다.

```
Oct  2 17:13:39 system lvm[13863]: Monitoring VDO pool vdo-name.
Oct  2 17:27:39 system lvm[13863]: WARNING: VDO pool vdo-name is now 80.69% full.
Oct  2 17:28:19 system lvm[13863]: WARNING: VDO pool vdo-name is now 85.25% full.
Oct  2 17:29:39 system lvm[13863]: WARNING: VDO pool vdo-name is now 90.64% full.
Oct  2 17:30:29 system lvm[13863]: WARNING: VDO pool vdo-name is now 96.07% full.
```

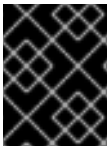
참고

이 경고 메시지는 **lvm2-monitor** 서비스가 실행 중인 경우에만 표시됩니다. 기본적으로 활성화되어 있습니다.

공간 부족 조건을 방지하는 방법

사용 가능한 풀 크기가 특정 수준 아래로 떨어지면 다음을 수행하여 작업을 수행할 수 있습니다.

- 데이터 삭제. 그러면 삭제된 데이터가 중복되지 않을 때마다 공간을 회수합니다. 데이터를 삭제하면 삭제가 발생한 후에만 공간을 사용할 수 있습니다.
- 물리적 스토리지 추가



중요

VDO 볼륨의 물리적 공간을 모니터링하여 공간 부족 상황을 방지합니다. 물리적 블록이 없으면 최근에 작성되었으며 VDO 볼륨에 승인되지 않은 데이터가 손실될 수 있습니다.

씬 프로비저닝 및 TRIM 및 DISCARD 명령

씬 프로비저닝의 스토리지 절감 효과를 얻기 위해서는 물리적 스토리지 계층에서 데이터를 삭제할 시기를 알아야 합니다. 씬 프로비저닝된 스토리지로 작동하는 파일 시스템은 **TRIM** 또는 **DISCARD** 명령을 전송하여 논리적 블록이 더 이상 필요하지 않은 경우 스토리지 시스템에 알립니다.

TRIM 또는 **DISCARD** 명령을 전송하는 몇 가지 방법을 사용할 수 있습니다.

- **discard** 마운트 옵션을 사용하면 파일 시스템은 블록이 삭제될 때마다 이러한 명령을 보낼 수 있습니다.
- **fstrim** 과 같은 유틸리티를 사용하여 제어되는 방식으로 명령을 보낼 수 있습니다. 이러한 유틸리티는 파일 시스템에 사용되지 않는 논리 블록을 감지하고 **TRIM** 또는 **DISCARD** 명령 형식으로 정보를 스토리지 시스템에 전송합니다.

사용하지 않는 블록에서 **TRIM** 또는 **DISCARD** 를 사용할 필요는 VDO에만 국한되지 않습니다. 씬 프로비저닝된 모든 스토리지 시스템에는 동일한 과제가 있습니다.

2.9.3. VDO 볼륨의 논리 크기 증가

이 절차에서는 지정된 VDO 볼륨의 논리적 크기를 늘립니다. 처음에는 논리 크기가 작은 VDO 볼륨을 만들 수 있으며 공간이 부족하여 안전하게 사용할 수 있습니다. 일정 기간 후에 실제 데이터 감소 비율을 평가하고 충분한 경우 공간 절약을 활용하기 위해 VDO 볼륨의 논리적 크기를 늘릴 수 있습니다.

VDO 볼륨의 논리 크기를 줄일 수 없습니다.

절차

- 논리 크기를 늘리려면 다음을 사용합니다.

```
# vdo growLogical --name=my-vdo \
                  --vdoLogicalSize=new-logical-size
```

  논리 크기가 증가하면 VDO는 새 크기의 볼륨 위에 장치 또는 파일 시스템을 알립니다.

2.9.4. VDO 볼륨의 물리 크기 증가

이 절차에서는 VDO 볼륨에서 사용할 수 있는 물리 스토리지의 양을 늘립니다.

이러한 방식으로 VDO 볼륨을 줄일 수 없습니다.

사전 요구 사항

- 기본 블록 장치는 현재 VDO 볼륨의 실제 크기보다 큰 용량을 갖습니다.
  그렇지 않으면 장치의 크기를 늘릴 수 있습니다. 정확한 절차는 장치 유형에 따라 다릅니다. 예를 들어 MBR 또는 GPT 파티션의 크기를 조정하려면 *스토리지 장치 관리* 가이드의 파티션 크기 조정 섹션을 참조하십시오.

절차

- VDO 볼륨에 새 물리 스토리지 공간을 추가합니다.

```
# vdo growPhysical --name=my-vdo
```

2.10. VDO 볼륨 제거

시스템에서 기존 VDO 볼륨을 제거할 수 있습니다.

2.10.1. 작동 중인 VDO 볼륨 제거

이 절차에서는 VDO 볼륨 및 관련 UDS 인덱스를 제거합니다.

절차

1. 파일 시스템을 마운트 해제하고 VDO 볼륨의 스토리지를 사용하는 애플리케이션을 중지합니다.
2. 시스템에서 VDO 볼륨을 제거하려면 다음을 사용합니다.

```
# vdo remove --name=my-vdo
```

2.10.2. 성공적으로 생성되지 않은 VDO 볼륨 제거

이 절차에서는 중간 상태에서 VDO 볼륨을 정리합니다. 볼륨 생성 시 오류가 발생하는 경우 볼륨이 중간 상태로 유지됩니다. 예를 들면 다음과 같은 경우에 발생할 수 있습니다.

- 시스템이 충돌합니다
- 전원 실패
- 관리자는 실행 중인 **vdo create 명령을 중단합니다**.

절차

- 정리하려면 **--force** 옵션을 사용하여 성공적으로 생성되지 않은 볼륨을 제거합니다.

```
# vdo remove --force --name=my-vdo
```

볼륨이 성공적으로 생성되지 않았기 때문에 관리자가 시스템 구성을 변경하여 충돌이 발생할 수 있으므로 **--force** 옵션이 필요합니다.

--force 옵션을 사용하지 않으면 **vdo remove** 명령이 실패하고 다음 메시지가 표시됩니다.

```
[...]
A previous operation failed.
Recovery from the failure either failed or was interrupted.
Add '--force' to 'remove' to perform the following cleanup.
Steps to clean up VDO my-vdo:
umount -f /dev/mapper/my-vdo
udevadm settle
dmsetup remove my-vdo
vdo: ERROR - VDO volume my-vdo previous operation (create) is incomplete
```

2.11. 추가 리소스

- **Ansible** 도구를 사용하여 VDO 배포 및 관리를 자동화할 수 있습니다. 자세한 내용은 다음을 참조하십시오.
  - Ansible 설명서: https://docs.ansible.com/
  - VDO Ansible 모듈 설명서: https://docs.ansible.com/ansible/latest/modules/vdo_module.html

3장. VDO 공간 테스트

일련의 테스트를 수행하여 VDO를 사용하여 절감할 수 있는 스토리지 공간을 확인할 수 있습니다.

사전 요구 사항

- 하나 이상의 물리적 블록 장치를 사용할 수 있습니다.
- 타겟 블록 장치는 512GiB보다 큽니다.
- VDO가 설치되어 있습니다.

3.1. VDO 테스트 목적 및 결과

Red Hat에서 제공하는 VDO 테스트는 VDO를 기존 스토리지 장치에 통합하는 평가를 수행하는 데 도움이 됩니다. 내부 평가는 대체하지 않고 보강하기 위한 것입니다.

테스트 결과를 통해 Red Hat 엔지니어는 특정 스토리지 환경에서 VDO 동작을 이해하는 데 도움이 됩니다. OEM(Original Equipment Manufacturer)은 중복 제거 및 압축이 가능한 장치를 설계하는 방법과 고객이 해당 장치를 위해 애플리케이션을 조정하는 방법을 배울 수 있습니다.

목표

- 테스트 장치에서 최적의 응답을 이끌어내는 구성 설정을 식별합니다.
- 제품 구성 오류를 방지하는 데 도움이 되는 기본 튜닝 매개 변수를 설명합니다.
- 실제 사용 사례와 비교할 성능 결과 참조를 생성합니다.
- 다양한 워크로드가 성능 및 데이터 효율성에 어떤 영향을 주는지 식별.
- VDO 구현으로 출시 기간 단축.

테스트 계획 및 테스트 조건

VDO 테스트에서는 VDO를 가장 현실적으로 평가할 수 있는 조건을 제공합니다. 테스트 절차 또는 매개 변수를 변경하면 결과가 무효화될 수 있습니다. Red Hat 영업 엔지니어가 테스트 계획을 수정할 때 안내할 수 있습니다.

효과적인 테스트 계획을 위해 VDO 아키텍처를 검토하고 다음 항목을 탐색해야 합니다.

- 고부하 환경에서의 성능
- 성능 튜닝 최종 사용자 애플리케이션을 위한 VDO의 구성 가능한 속성
- VDO가 네이티브 4 KiB 블록 장치로 미치는 영향
- 중복 제거 및 압축의 배포에 대한 액세스 패턴 및 배포에 대한 응답
- 주어진 애플리케이션의 성능 대비 비용 대 용량의 가치

3.2. VDO의 씬 프로비저닝

VDO는 씬 프로비저닝된 블록 스토리지 대상입니다. VDO 볼륨에서 사용하는 물리 공간의 양은 스토리지 사용자에게 표시되는 볼륨 크기와 다를 수 있습니다. 이러한 차이를 활용하여 스토리지 비용을 절감할 수 있습니다.

공간 외 조건

작성한 데이터가 예상 최적화 속도를 달성하지 못하는 경우 예기치 않게 스토리지 공간이 부족하지 않도록 주의하십시오.

논리 블록(가상 스토리지) 수가 물리적 블록(실제 저장소) 수를 초과할 때마다 파일 시스템 및 애플리케이션이 예기치 않게 공간이 부족해질 수 있습니다. 따라서 VDO를 사용하는 스토리지 시스템은 VDO 볼륨에서 사용 가능한 풀의 크기를 모니터링하는 방법을 제공해야 합니다.

vdostats 유틸리티를 사용하여 이 사용 가능한 풀의 크기를 확인할 수 있습니다. 이 유틸리티의 기본 출력에는 Linux **df** 유틸리티와 유사한 형식으로 실행되는 모든 VDO 볼륨에 대한 정보가 나열됩니다. 예를 들면 다음과 같습니다.

```
Device                1K-blocks   Used       Available   Use%
/dev/mapper/vdo-name  211812352   105906176  105906176   50%
```

VDO 볼륨의 물리적 스토리지 용량이 거의 가득 차면 VDO는 다음과 유사하게 시스템 로그에 경고를 보고합니다.

```
Oct  2 17:13:39 system lvm[13863]: Monitoring VDO pool vdo-name.
Oct  2 17:27:39 system lvm[13863]: WARNING: VDO pool vdo-name is now 80.69% full.
Oct  2 17:28:19 system lvm[13863]: WARNING: VDO pool vdo-name is now 85.25% full.
Oct  2 17:29:39 system lvm[13863]: WARNING: VDO pool vdo-name is now 90.64% full.
Oct  2 17:30:29 system lvm[13863]: WARNING: VDO pool vdo-name is now 96.07% full.
```

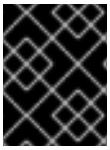
참고

이 경고 메시지는 **lvm2-monitor** 서비스가 실행 중인 경우에만 표시됩니다. 기본적으로 활성화되어 있습니다.

공간 부족 조건을 방지하는 방법

사용 가능한 풀 크기가 특정 수준 아래로 떨어지면 다음을 수행하여 작업을 수행할 수 있습니다.

- 데이터 삭제. 그러면 삭제된 데이터가 중복되지 않을 때마다 공간을 회수합니다. 데이터를 삭제하면 삭제가 발생한 후에만 공간을 사용할 수 있습니다.
- 물리적 스토리지 추가



중요

VDO 볼륨의 물리적 공간을 모니터링하여 공간 부족 상황을 방지합니다. 물리적 블록이 없으면 최근에 작성되었으며 VDO 볼륨에 승인되지 않은 데이터가 손실될 수 있습니다.

씬 프로비저닝 및 TRIM 및 DISCARD 명령

씬 프로비저닝의 스토리지 절감 효과를 얻기 위해서는 물리적 스토리지 계층에서 데이터를 삭제할 시기를 알아야 합니다. 씬 프로비저닝된 스토리지로 작동하는 파일 시스템은 **TRIM** 또는 **DISCARD** 명령을 전송하여 논리적 블록이 더 이상 필요하지 않은 경우 스토리지 시스템에 알립니다.

TRIM 또는 **DISCARD** 명령을 전송하는 몇 가지 방법을 사용할 수 있습니다.

- **discard** 마운트 옵션을 사용하면 파일 시스템은 블록이 삭제될 때마다 이러한 명령을 보낼 수 있습니다.

- **fstrim** 과 같은 유틸리티를 사용하여 제어되는 방식으로 명령을 보낼 수 있습니다. 이러한 유틸리티는 파일 시스템에 사용되지 않는 논리 블록을 감지하고 **TRIM** 또는 **DISCARD** 명령 형식으로 정보를 스토리지 시스템에 전송합니다.

사용하지 않는 블록에서 **TRIM** 또는 **DISCARD** 를 사용할 필요는 VDO에만 국한되지 않습니다. 씬 프로비저닝된 모든 스토리지 시스템에는 동일한 과제가 있습니다.

3.3. 각 VDO 테스트 전에 기록할 정보

테스트 환경을 완전히 이해하려면 각 테스트를 시작할 때 다음 정보를 기록해야 합니다. **sosreport** 유틸리티를 사용하여 필요한 정보의 대부분을 캡처할 수 있습니다.

필요한 정보

- 커널 빌드 번호를 포함하여 사용된 Linux 빌드
- **rpm -qa** 명령에서 가져온 대로 설치된 패키지의 전체 목록
- 전체 시스템 사양
  - CPU 유형 및 수량; **/proc/cpuinfo** 파일에서 사용 가능
  - rase OS가 실행된 후 사용할 수 있는 메모리 및 용량이 설치되어 있습니다. **/proc/meminfo** 파일에서 사용 가능
  - 사용된 드라이브 컨트롤러 유형
  - 사용된 디스크의 유형 및 수량
- 실행중인 프로세스의 전체 목록; **ps aux** 명령 또는 유사한 목록에서 사용할 수 있습니다.
- VDO와 함께 사용하기 위해 생성된 물리 볼륨 및 볼륨 그룹의 이름; **pvs 및 vgs** 명령에서 사용 가능
- VDO 볼륨을 포맷 할 때 사용되는 파일 시스템 (있는 경우)
- 마운트된 디렉토리에 대한 권한
- **/etc/vdoconf.yaml** 파일의 내용
- VDO 파일의 위치

3.4. VDO 테스트 볼륨 생성

이 절차에서는 테스트를 위해 512GiB 물리 볼륨에 논리 크기가 1TiB인 VDO 볼륨을 생성합니다.

절차

1. VDO 볼륨을 생성합니다.

```
# vdo create --name=vdo-test \
        --device=/dev/sdb \
        --vdoLogicalSize=1T \
        --writePolicy=policy \
        --verbose
```

- */dev/sdb* 를 블록 장치의 경로로 바꿉니다.
- 비동기 스토리지에서 VDO **비동기** 모드를 테스트하려면 **--writePolicy=async** 옵션을 사용하여 비동기 볼륨을 생성합니다.
- 동기 스토리지에서 VDO **동기화** 모드를 테스트하려면 **--writePolicy=sync** 옵션을 사용하여 동기 볼륨을 생성합니다.

2. XFS 또는 ext4 파일 시스템으로 새 볼륨을 포맷합니다.
   - XFS의 경우:

     ```
     # mkfs.xfs -K /dev/mapper/vdo-test
     ```

   - ext4의 경우:

     ```
     # mkfs.ext4 -E nodiscard /dev/mapper/vdo-test
     ```

3. 포맷된 볼륨을 마운트합니다.

   ```
   # mkdir /mnt/vdo-test

   # mount /dev/mapper/vdo-test /mnt/vdo-test && \
     chmod a+rwx /mnt/vdo-test
   ```

3.5. VDO 테스트 볼륨 테스트

이 절차에서는 VDO 테스트 볼륨의 읽기 및 쓰기가 작동하는지 테스트합니다.

사전 요구 사항

- 새로 생성된 VDO 테스트 볼륨이 마운트됩니다. 자세한 내용은 3.4절. "VDO 테스트 볼륨 생성" 의 내용을 참조하십시오.

절차

1. 32GiB 임의 데이터를 VDO 볼륨에 작성합니다.

   ```
   $ dd if=/dev/urandom of=/mnt/vdo-test/testfile bs=4096 count=8388608
   ```

2. VDO 볼륨에서 데이터를 읽고 다른 볼륨에 씁니다.

   ```
   $ dd if=/mnt/vdo-test/testfile of=another-location/testfile bs=4096
   ```

   - *another-location* 을 VDO 테스트 볼륨에 없는 쓰기 액세스 권한이 있는 디렉터리로 바꿉니다. 예를 들어 홈 디렉터리를 사용할 수 있습니다.

3. 두 파일을 비교하십시오.

   ```
   $ diff --report-identical-files /mnt/vdo-test/testfile another-location/testfile
   ```

   명령은 파일이 동일함을 보고해야 합니다.

4. 파일을 VDO 볼륨의 새 위치로 다시 복사합니다.

```
$ dd if=another-location/testfile of=/mnt/vdo-test/testfile2 bs=4096
```

5. 세 번째 파일을 두 번째 파일과 비교합니다.

```
$ diff --report-identical-files /mnt/vdo-test/testfile2 another-location/testfile
```

명령은 파일이 동일함을 보고해야 합니다.

정리 단계

- 3.6절. "VDO 테스트 볼륨 정리" 에 설명된 대로 VDO 테스트 볼륨을 제거합니다.

3.6. VDO 테스트 볼륨 정리

이 절차에서는 시스템에서 VDO 효율성을 테스트하는 데 사용되는 VDO 볼륨을 제거합니다.

사전 요구 사항

- VDO 테스트 볼륨이 마운트됩니다.

절차

1. VDO 볼륨에서 생성된 파일 시스템을 마운트 해제합니다.

```
# umount /mnt/vdo-test
```

2. 시스템에서 VDO 테스트 볼륨을 제거합니다.

```
# vdo remove --name=vdo-test
```

검증 단계

- 볼륨이 제거되었는지 확인합니다.

```
# vdo list --all | grep vdo-test
```

명령은 VDO 테스트 파티션을 나열하지 않아야 합니다.

3.7. VDO 중복 제거 측정

이 절차에서는 VDO 테스트 볼륨에서 VDO 데이터 중복 제거의 효율성을 테스트합니다.

사전 요구 사항

- 새로 생성된 VDO 테스트 볼륨이 마운트됩니다. 자세한 내용은 3.4절. "VDO 테스트 볼륨 생성" 의 내용을 참조하십시오.

절차

1. 테스트 결과를 기록할 수 있는 테이블을 준비합니다.

통계	베어 파일 시스템	시드 후	10개 복사본 후
파일 시스템 사용 크기			
VDO 데이터 사용			
VDO 논리 사용			

2. VDO 볼륨에 10개의 디렉터리를 생성하여 테스트 데이터 세트의 복사본 10개를 보유합니다.

```
$ mkdir /mnt/vdo-test/vdo{01..10}
```

3. 파일 시스템에서 보고한 디스크 사용량을 검사합니다.

```
$ df --human-readable /mnt/vdo-test
```

예 3.1. 디스크 사용량

```
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/vdo-test  1.5T  198M  1.4T   1% /mnt/vdo-test
```

4. 다음 값을 기록합니다.

```
# vdostats --verbose | grep "blocks used"
```

예 3.2. 사용된 블록

```
data blocks used               : 1090
overhead blocks used           : 538846
logical blocks used            : 6059434
```

- **사용된** 값은 VDO에서 실행되는 물리적 장치에서 최적화한 후 사용자 데이터가 사용하는 블록 수입니다.
- **논리 블록 사용된** 값은 최적화 전에 사용된 블록 수입니다. 측정의 시작점으로 사용됩니다.

5. VDO 볼륨에 데이터 소스 파일을 생성합니다.

```
$ dd if=/dev/urandom of=/mnt/vdo-test/sourcefile bs=4096 count=1048576

4294967296 bytes (4.3 GB) copied, 540.538 s, 7.9 MB/s
```

6. 사용된 물리적 디스크 공간을 다시 검사합니다.

```
$ df --human-readable /mnt/vdo-test
```

예 3.3. 데이터 소스 파일을 사용한 디스크 사용량

```
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/vdo-test  1.5T  4.2G  1.4T   1% /mnt/vdo-test
```

```
# vdostats --verbose | grep "blocks used"
```

예 3.4. 데이터 소스 파일과 함께 사용된 블록

```
data blocks used               : 1050093  # Increased by 4GiB
overhead blocks used           : 538846   # Did not significantly change
logical blocks used            : 7108036  # Increased by 4GiB
```

이 명령은 작성된 파일의 크기에 해당하는 사용된 블록 수의 증가를 표시해야 합니다.

7. 파일을 10개의 각 하위 디렉토리에 복사합니다.

```
$ for i in {01..10}; do
  cp /mnt/vdo-test/sourcefile /mnt/vdo-test/vdo$i
  done
```

8. 사용된 물리적 디스크 공간을 다시 검사합니다.

```
$ df -h /mnt/vdo-test
```

예 3.5. 파일을 복사한 후 디스크 사용량

```
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/vdo-test  1.5T   45G  1.3T   4% /mnt/vdo-test
```

```
# vdostats --verbose | grep "blocks used"
```

예 3.6. 파일을 복사한 후 사용된 블록

```
data blocks used               : 1050836   # Increased by 3 MiB
overhead blocks used           : 538846
logical blocks used            : 17594127  # Increased by 41 GiB
```

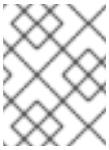
사용된 데이터 블록은 이전 목록의 결과와 유사해야 하며 파일 시스템 저널링 및 메타데이터로 인해 약간만 증가합니다.

9. 테스트 데이터를 쓰기 전에 찾은 값에서 파일 시스템에서 사용하는 공간의 새 값을 뺍니다. 파일 시스템의 관점에서 이 테스트에서 사용하는 공간입니다.

10. 기록된 통계에서 공간 절약을 관찰합니다.

예 3.7. 기록된 값

통계	베어 파일 시스템	시드 후	10개 복사본 후
파일 시스템 사용 크기	198MiB	4.2GiB	45GiB
VDO 데이터 사용	4MiB	4.1GiB	4.1GiB
VDO 논리 사용	23.6GiB *(1.6 TiB 형식의 드라이브의 파일 시스템 오버헤드)*	27.8GiB	GiB.7GiB



참고

표에서 값이 MiB 또는 GiB로 변환되었습니다. **vdostats** 출력의 블록 크기는 4,096 B입니다.

정리 단계

- 3.6절. "VDO 테스트 볼륨 정리" 에 설명된 대로 VDO 테스트 볼륨을 제거합니다.

3.8. VDO 압축 측정

이 절차에서는 VDO 테스트 볼륨에서 VDO 데이터 압축의 효율성을 테스트합니다.

사전 요구 사항

- 새로 생성된 VDO 테스트 볼륨이 마운트됩니다. 자세한 내용은 3.4절. "VDO 테스트 볼륨 생성" 의 내용을 참조하십시오.

절차

1. VDO 테스트 볼륨에서 중복 제거를 비활성화하고 압축을 활성화합니다.

```
# vdo disableDeduplication --name=vdo-test
# vdo enableCompression --name=vdo-test
```

2. VDO 볼륨을 동기화하여 완료되지 않은 압축을 완료합니다.

```
# sync && dmsetup message vdo-test 0 sync-dedupe
```

3. 전송 전에 VDO 통계를 검사합니다.

```
# vdostats --verbose | grep "blocks used"
```

사용된 **데이터 블록과 사용된 값을 기록해 두십시오.**

4. **VDO는 파일 시스템 오버헤드와 실제 사용자 데이터를 최적화합니다. 사용된 폐기 데이터 블록을 사용하는 논리적 블록으로 빈 파일 시스템의 압축으로 저장된 4 KiB 블록 수를 계산합니다.**

5. **/lib 디렉토리의 내용을 VDO 볼륨에 복사합니다.**

```
# cp --verbose --recursive /lib /mnt/vdo-test

...
sent 152508960 bytes  received 60448 bytes  61027763.20 bytes/sec
total size is 152293104  speedup is 1.00
```

복사된 데이터의 총 크기를 기록합니다.

6. **Linux 캐시 및 VDO 볼륨을 동기화합니다.**

```
# sync && dmsetup message vdo-test 0 sync-dedupe
```

7. **VDO 통계를 다시 검사합니다.**

```
# vdostats --verbose | grep "blocks used"
```

사용된 논리 블록과 사용된 값을 관찰합니다.

8. **다음 공식을 사용하여 압축을 통해 저장된 바이트 크기를 계산합니다.**

```
saved_bytes = (logical_blocks_used - data_blocks_used) * 4096
```

정리 단계

- **3.6절. "VDO 테스트 볼륨 정리" 에 설명된 대로 VDO 테스트 볼륨을 제거합니다.**

3.9. 총 VDO 공간 절감 측정

이 절차에서는 VDO 테스트 볼륨에서 VDO 데이터 중복 제거 및 압축의 결합된 효율성을 테스트합니다.

절차

1. 3.4절. “VDO 테스트 볼륨 생성” 에 설명된 대로 VDO 볼륨을 생성하고 마운트합니다.

2. VDO 중복 제거 및 VDO 압축을 제거하지 않고 동일한 볼륨에서 VDO 압축 측정에 설명된 테스트를 수행합니다. vdostats 출력에서 공간 절약에 대한 변경 사항을 관찰합니다.

3. 자체 데이터 세트를 사용해 보십시오.

3.10. VDO에서 TRIM 및 DISCARD의 효과 테스트

이 절차에서는 TRIM 및 DISCARD 명령이 VDO 테스트 볼륨에서 삭제된 파일에서 블록을 올바르게 확보할 수 있는지 여부를 테스트합니다. 삭제하면 VDO에 공간이 더 이상 사용되지 않음을 알 수 있습니다.

사전 요구 사항

- 새로 생성된 VDO 테스트 볼륨이 마운트됩니다. 자세한 내용은 3.4절. “VDO 테스트 볼륨 생성” 의 내용을 참조하십시오.

절차

1. 테스트 결과를 기록할 수 있는 테이블을 준비합니다.

Step	사용된 파일 공간 (MB)	사용된 데이터 블록	사용된 논리 블록
초기			
1GiB 파일 추가			
fstrim 실행			
1GiB 파일 삭제			
fstrim 실행			

2. 불필요한 블록을 제거하기 위해 파일 시스템을 줄입니다.

```
# fstrim /mnt/vdo-test
```

이 명령은 시간이 오래 걸릴 수 있습니다.

3. 파일 시스템에 초기 공간 사용량을 기록합니다.

```
$ df -m /mnt/vdo-test
```

4. VDO 볼륨에서 사용하는 물리 및 논리적 데이터 블록 수를 확인하십시오.

```
# vdostats --verbose | grep "blocks used"
```

5. VDO 볼륨에서 중복되지 않은 데이터를 사용하여 1GiB 파일을 생성합니다.

```
$ dd if=/dev/urandom of=/mnt/vdo-test/file bs=1M count=1K
```

6. 공간 사용량을 다시 기록합니다.

```
$ df -m /mnt/vdo-test

# vdostats --verbose | grep "blocks used"
```

파일 시스템은 추가 1GiB를 사용해야 합니다. 사용된 데이터 블록과 사용된 값의 논리 블록도 마찬가지로 증가해야 합니다.

7. 파일 시스템을 다시 줄입니다.

```
# fstrim /mnt/vdo-test
```

8. 공간 사용량을 다시 검사하여 트림이 물리 볼륨 사용량에 영향을 미치지 않았는지 확인합니다.

```
$ df -m /mnt/vdo-test

# vdostats --verbose | grep "blocks used"
```

9. 1GiB 파일을 삭제합니다.

```
$ rm /mnt/vdo-test/file
```

10. 공간 사용량을 다시 확인하고 기록합니다.

```
$ df -m /mnt/vdo-test

# vdostats --verbose | grep "blocks used"
```

파일 시스템은 파일이 삭제되었음을 인식하지만, 파일 삭제가 기본 스토리지에 전송되지 않았기 때문에 실제 또는 논리 블록의 수는 변경되지 않습니다.

11. 파일 시스템을 다시 줄입니다.

```
# fstrim /mnt/vdo-test
```

12. 공간 사용량을 다시 확인하고 기록합니다.

```
$ df -m /mnt/vdo-test

# vdostats --verbose | grep "blocks used"
```

fstrim 유틸리티는 파일 시스템에서 사용 가능한 블록을 찾고, **TRIM** 명령을 사용하지 않은 주소에 대해 **VDO** 볼륨으로 보냅니다. 이 명령은 관련 논리 블록을 해제합니다. **VDO**는 **TRIM** 명령을 처리하여 기본 물리 블록을 해제합니다.

추가 리소스

- **TRIM** 및 **DISCARD** 명령, **fstrim** 유틸리티 및 **discard** 마운트 옵션에 대한 자세한 내용은 를 참조하십시오. **5**장. *사용하지 않는 블록 삭제*

4장. VDO 성능 테스트

일련의 테스트를 수행하여 VDO 성능을 측정하고, VDO를 사용하여 시스템의 성능 프로필을 확보하고, VDO에서 제대로 수행하는 애플리케이션을 확인할 수 있습니다.

사전 요구 사항

- 하나 이상의 Linux 물리적 블록 장치를 사용할 수 있습니다.
- 대상 블록 장치(예: /dev/sdb)는 512GiB보다 큽니다.
- 유연한 I/O 테스터(fio)가 설치되어 있습니다.
- VDO가 설치되어 있습니다.

4.1. VDO 성능 테스트를 위한 환경 준비

VDO 성능을 테스트하기 전에 호스트 시스템 구성, VDO 구성 및 테스트 중에 사용할 워크로드를 고려해야 합니다. 이러한 선택 사항은 공간 효율성, 대역폭 및 대기 시간의 벤치마킹에 영향을 줍니다.

한 테스트가 다른 테스트의 결과에 영향을 미치지 않도록 하려면 각 테스트의 각 반복에 대해 새 VDO 볼륨을 만들어야 합니다.

4.1.1. VDO 성능 테스트 전 고려 사항

다음 조건 및 구성은 VDO 테스트 결과에 영향을 미칩니다.

시스템 설정

- 사용 가능한 CPU 코어 수와 유형. taskset 유틸리티를 사용하여 이 정보를 나열할 수 있습니다.
- 사용 가능한 메모리 및 총 설치된 메모리

- 스토리지 장치 구성
- 활성 디스크 스케줄러
- Linux 커널 버전
- 패키지 설치

VDO 구성

- 파티션 구성
- VDO 볼륨에 사용되는 파일 시스템
- VDO 볼륨에 할당된 물리 스토리지의 크기
- 생성된 논리 VDO 볼륨의 크기
- 스파스 또는 밀도가 높은 UDS 인덱싱
- 메모리 크기의 UDS 인덱스
- VDO 스레드 구성

워크로드

- 테스트 데이터를 생성하는 데 사용되는 도구 유형
- 동시 클라이언트 수

- 서면 데이터에서 중복 된 **4 KiB** 블록 수
- 패턴 읽기 및 쓰기
- 작업 세트 크기

4.1.2. VDO 읽기 성능 테스트를 위한 특수 고려 사항

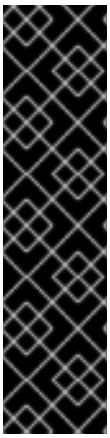
VDO 읽기 성능을 테스트하기 전에 이러한 추가 요소를 고려해야 합니다.

- **4 KiB** 블록이 작성되지 않은 경우 **VDO**는 스토리지를 읽지 않고 즉시 **0** 블록으로 응답합니다.
- **4 KiB** 블록이 작성되었지만 모두 **0**이 포함된 경우 **VDO**는 스토리지에서 읽지 않고 즉시 **0** 블록으로 응답합니다.

이 동작은 읽을 데이터가 없는 경우 읽기 속도가 매우 빠릅니다. 따라서 읽기 테스트를 통해 볼륨을 실제 데이터로 미리 채워야 합니다.

4.1.3. VDO 성능 테스트를 위한 시스템 준비

이 절차에서는 테스트 중에 최적의 **VDO** 성능을 제공하도록 시스템 설정을 구성합니다.



중요

특정 테스트에 나열된 바운드를 초과하면 비정상적인 결과로 인해 테스트 시간이 손실될 수 있습니다.

예를 들어, **VDO** 테스트에서는 **100GiB** 주소 범위에서 임의 읽기를 수행하는 테스트를 설명합니다. 작업 중인 **500GiB** 세트를 테스트하려면 **VDO** 블록 맵 캐시에 할당된 **RAM** 크기를 적절하게 늘려야 합니다.

절차

1. CPU가 최고 성능 설정으로 실행되고 있는지 확인합니다.

2. 가능한 경우 BIOS 구성 또는 Linux cpupower 유틸리티를 사용하여 CPU 빈도 확장을 비활성화합니다.

3. 가능한 경우 CPU에 대해 동적 프로세서 빈도 조정(Turbo Boost 또는 Turbo Core)을 활성화합니다. 이 기능을 사용하면 테스트 결과에 약간의 변동성이 발생하지만 전반적인 성능이 향상됩니다.

4. 파일 시스템은 성능에 고유한 영향을 줄 수 있습니다. 이들은 종종 성능 측정을 불일치하여 결과에 대한 VDO의 영향을 격리하기 어렵게 만듭니다.

   적절한 경우 원시 블록 장치의 성능을 측정합니다. 이 기능을 사용할 수 없는 경우 VDO가 대상 구현에 사용할 파일 시스템을 사용하여 장치를 포맷합니다.

4.2. 성능 테스트를 위한 VDO 볼륨 생성

이 절차에서는 VDO 성능을 테스트하기 위해 512GiB 물리 볼륨에 논리 크기가 1TiB인 VDO 볼륨을 생성합니다.

절차

- VDO 볼륨을 생성합니다.

```
# vdo create --name=vdo-test \
        --device=/dev/sdb \
        --vdoLogicalSize=1T \
        --writePolicy=policy \
        --verbose
```

  - */dev/sdb* 를 블록 장치의 경로로 바꿉니다.
  - 비동기 스토리지에서 VDO 비동기 모드를 테스트하려면 **--writePolicy=async** 옵션을 사용하여 비동기 볼륨을 생성합니다.
  - 동기 스토리지에서 VDO 동기화 모드를 테스트하려면 **--writePolicy=sync** 옵션을 사용하여 동기 볼륨을 생성합니다.

4.3. VDO 성능 테스트 볼륨 정리

이 절차에서는 시스템에서 VDO 성능을 테스트하는 데 사용되는 VDO 볼륨을 제거합니다.

사전 요구 사항

- VDO 테스트 볼륨이 시스템에 있습니다.

절차

- 시스템에서 VDO 테스트 볼륨을 제거합니다.

```
# vdo remove --name=vdo-test
```

검증 단계

- 볼륨이 제거되었는지 확인합니다.

```
# vdo list --all | grep vdo-test
```

명령은 VDO 테스트 파티션을 나열하지 않아야 합니다.

4.4. VDO 성능에서 I/O 깊이의 영향 테스트

이 테스트에서는 최적의 처리량과 VDO 구성에 가장 낮은 대기 시간을 생성하는 I/O 깊이를 결정합니다. I/O 깊이는 fio 툴이 한 번에 제출하는 I/O 요청 수를 나타냅니다.

VDO는 4KiB 섹터 크기를 사용하기 때문에 테스트에서는 4KiB I/O 작업과 1, 8, 16, 32, 64, 128, 256, 512, 1024의 I/O 깊이를 테스트합니다.

4.4.1. 순차적 100%에서 I/O 깊이의 효과를 VDO에서 테스트

이 테스트에서는 다양한 I/O 깊이 값에서 VDO 볼륨에서 순차적으로 100% 읽기 작업을 수행하는 방식을 결정합니다.

절차

1. 새 VDO 볼륨을 만듭니다.

   자세한 내용은 4.2절. “성능 테스트를 위한 VDO 볼륨 생성” 의 내용을 참조하십시오.

2. 테스트 볼륨에서 쓰기 fio 작업을 수행하여 테스트에서 액세스할 수 있는 모든 영역을 미리 채웁니다.

```
# fio --rw=write \
      --bs=8M \
      --name=vdo \
      --filename=/dev/mapper/vdo-test \
      --ioengine=libaio \
      --thread \
      --direct=1 \
      --scramble_buffers=1
```

3. 순차적 100% 읽기에 대해 보고된 처리량 및 대기 시간을 기록합니다.

```
# for depth in 1 2 4 8 16 32 64 128 256 512 1024 2048; do
  fio --rw=read \
      --bs=4096 \
      --name=vdo \
      --filename=/dev/mapper/vdo-test \
      --ioengine=libaio \
      --numjobs=1 \
      --thread \
      --norandommap \
      --runtime=300 \
      --direct=1 \
      --iodepth=$depth \
      --scramble_buffers=1 \
      --offset=0 \
      --size=100g
  done
```

4. VDO 테스트 볼륨을 제거합니다.

   자세한 내용은 4.3절. “VDO 성능 테스트 볼륨 정리” 의 내용을 참조하십시오.

4.4.2. VDO에서 순차적 100% 쓰기에서 I/O 깊이의 효과 테스트

이 테스트는 다른 I/O 깊이 값에서 VDO 볼륨에서 순차적으로 100% 쓰기 작업을 수행하는 방식을 결정

합니다.

절차

1. 새 VDO 테스트 볼륨을 만듭니다.

   자세한 내용은 4.2절. "성능 테스트를 위한 VDO 볼륨 생성" 의 내용을 참조하십시오.

2. 쓰기 fio 작업을 수행하여 테스트에서 액세스할 수 있는 모든 영역을 미리 채웁니다.

```
# fio --rw=write \
    --bs=8M \
    --name=vdo \
    --filename=/dev/mapper/vdo-test \
    --ioengine=libaio \
    --thread \
    --direct=1 \
    --scramble_buffers=1
```

3. 순차적 100% 쓰기에 대해 보고된 처리량 및 대기 시간을 기록합니다.

```
# for depth in 1 2 4 8 16 32 64 128 256 512 1024 2048; do
 fio --rw=write \
    --bs=4096 \
    --name=vdo \
    --filename=/dev/mapper/vdo-test \
    --ioengine=libaio \
    --numjobs=1 \
    --thread \
    --norandommap \
    --runtime=300 \
    --direct=1 \
    --iodepth=$depth \
    --scramble_buffers=1 \
    --offset=0 \
    --size=100g
 done
```

4. VDO 테스트 볼륨을 제거합니다.

   자세한 내용은 4.3절. "VDO 성능 테스트 볼륨 정리" 의 내용을 참조하십시오.

4.4.3. VDO에서 임의의 100% 읽기에서 I/O 깊이의 효과 테스트

이 테스트에서는 다양한 I/O 깊이 값에서 VDO 볼륨에서 임의의 100% 읽기 작업을 수행하는 방식을 결정합니다.

절차

1. 새 VDO 테스트 볼륨을 만듭니다.

   자세한 내용은 4.2절. "성능 테스트를 위한 VDO 볼륨 생성" 의 내용을 참조하십시오.

2. 쓰기 fio 작업을 수행하여 테스트에서 액세스할 수 있는 모든 영역을 미리 채웁니다.

```
# fio --rw=write \
    --bs=8M \
    --name=vdo \
    --filename=/dev/mapper/vdo-test \
    --ioengine=libaio \
    --thread \
    --direct=1 \
    --scramble_buffers=1
```

3. 임의 100% 읽기에 대해 보고된 처리량 및 대기 시간을 기록합니다.

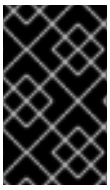
```
# for depth in 1 2 4 8 16 32 64 128 256 512 1024 2048; do
 fio --rw=randread \
    --bs=4096 \
    --name=vdo \
    --filename=/dev/mapper/vdo-test \
    --ioengine=libaio \
    --numjobs=1 \
    --thread \
    --norandommap \
    --runtime=300 \
    --direct=1 \
    --iodepth=$depth \
    --scramble_buffers=1 \
    --offset=0 \
    --size=100g
 done
```

4. VDO 테스트 볼륨을 제거합니다.

   자세한 내용은 4.3절. "VDO 성능 테스트 볼륨 정리" 의 내용을 참조하십시오.

4.4.4. VDO에서 임의의 100% 쓰기에서 I/O 깊이의 영향 테스트

이 테스트에서는 다양한 I/O 깊이 값의 VDO 볼륨에서 임의 100% 쓰기 작업이 수행하는 방식을 결정합니다.



중요

각 I/O 심층 테스트 실행 간에 VDO 볼륨을 다시 생성해야 합니다.

절차

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048의 I/O 깊이 값에 대해 다음 일련의 단계를 별도로 수행합니다.

1. 새 VDO 테스트 볼륨을 만듭니다.

   자세한 내용은 4.2절. "성능 테스트를 위한 VDO 볼륨 생성" 의 내용을 참조하십시오.

2. 쓰기 fio 작업을 수행하여 테스트에서 액세스할 수 있는 모든 영역을 미리 채웁니다.

```
# fio --rw=write \
    --bs=8M \
    --name=vdo \
    --filename=/dev/mapper/vdo-test \
    --ioengine=libaio \
    --thread \
    --direct=1 \
    --scramble_buffers=1
```

3. 임의 100% 쓰기에 대해 보고된 처리량 및 대기 시간을 기록합니다.

```
# fio --rw=randwrite \
    --bs=4096 \
    --name=vdo \
    --filename=/dev/mapper/vdo-test \
    --ioengine=libaio \
    --numjobs=1 \
    --thread \
    --norandommap \
    --runtime=300 \
    --direct=1 \
    --iodepth=depth-value
```

```
    --scramble_buffers=1 \
    --offset=0 \
    --size=100g
done
```

4. VDO 테스트 볼륨을 제거합니다.

   자세한 내용은 4.3절. “VDO 성능 테스트 볼륨 정리” 의 내용을 참조하십시오.

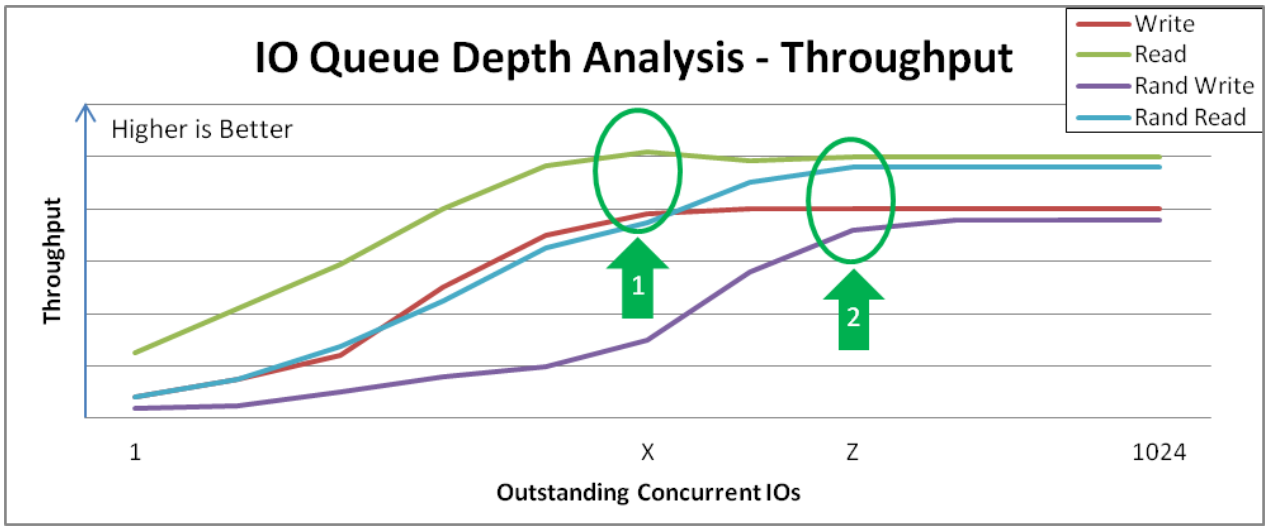
4.4.5. 다른 I/O 깊이에서의 VDO 성능 분석

다음 예제에서는 다양한 I/O 깊이 값에 기록된 VDO 처리량 및 대기 시간을 분석합니다.

I/O 깊이가 증가하여 처리량이 감소하는 변동 지점과 범위 전반에 걸친 동작을 확인합니다. 순차적 액세스와 임의 액세스는 서로 다른 값에서 최고치일 수 있지만 모든 유형의 스토리지 구성에서는 피크 시간이 다를 수 있습니다.

예 4.1. I/O 깊이 분석

그림 4.1. VDO 처리량 분석



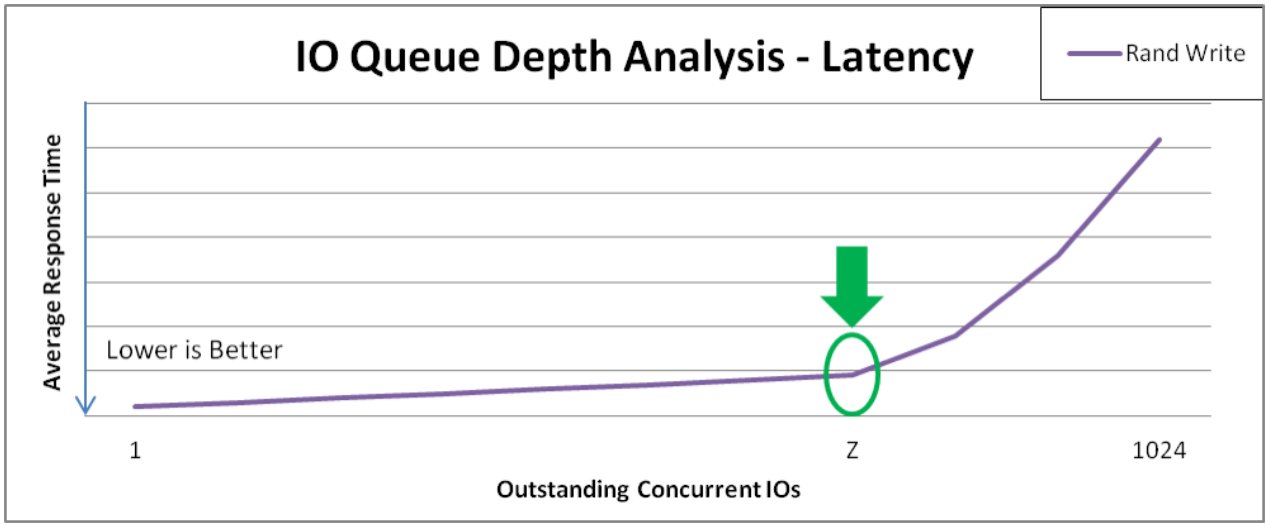
각 성능 곡선에서 "knee"를 확인합니다.

- 마커 1은 지점 X 에서 최대 순차적 처리량을 식별합니다. 이 특정 구성은 X 보다 큰 순차 4 KiB I/O 깊이에서 혜택을 받지 않습니다.
- 마커 2는 지점 Z 에서 최대 임의 4KiB 처리량을 식별합니다. 이 특정 구성은 Z 보다 큰 임의의 4 KiB I/O 깊이를 사용할 수 없습니다.

X 및 Z 의 I/O 깊이를 넘어 대역폭 증가는 줄어들고 평균 요청 대기 시간이 추가 I/O 요청에 대해 1:1 증가합니다.

다음 이미지는 이전 그래프에서 곡선의 "knee" 뒤의 임의 쓰기 대기 시간의 예를 보여줍니다. 이러한 시점에서 응답 시간이 가장 적은 최대 처리량을 테스트해야 합니다.

그림 4.2. VDO 대기 시간 분석



최적의 I/O 깊이

점 Z 는 최적의 I/O 깊이를 표시합니다. 테스트 계획은 Z 와 동일한 I/O 깊이를 사용하여 추가 데이터를 수집합니다.

4.5. VDO 성능에서 I/O 요청 크기의 영향 테스트

이러한 테스트를 사용하면 최적의 I/O 깊이에서 VDO의 최상의 성능을 생성하는 블록 크기를 확인할 수 있습니다.

테스트에서는 고정된 I/O 깊이에서 4Ker 테스트를 수행하고, 8KiB ~ 1MiB 범위의 블록 크기가 다양합니다.

사전 요구 사항

- 최적의 I/O 깊이 값을 결정했습니다. 자세한 내용은 4.4절. “VDO 성능에서 I/O 깊이의 영향 테스트” 의 내용을 참조하십시오.

다음 테스트에서 최적의 깊이를 *최적의* I/O 깊이 값으로 바꿉니다.

4.5.1. VDO에서 순차적 쓰기에 I/O 요청 크기의 효과 테스트

이 테스트에서는 다양한 I/O 요청 크기의 VDO 볼륨에서 순차적 쓰기 작업이 수행하는 방식을 결정합니다.

절차

1. 새 VDO 볼륨을 만듭니다.

   자세한 내용은 4.2절. "성능 테스트를 위한 VDO 볼륨 생성" 의 내용을 참조하십시오.

2. 테스트 볼륨에서 쓰기 fio 작업을 수행하여 테스트에서 액세스할 수 있는 모든 영역을 미리 채웁니다.

```
# fio --rw=write \
      --bs=8M \
      --name=vdo \
      --filename=/dev/mapper/vdo-test \
      --ioengine=libaio \
      --thread \
      --direct=1 \
      --scramble_buffers=1
```

3. 순차적 쓰기 테스트에 대해 보고된 처리량 및 대기 시간을 기록합니다.

```
# for iosize in 4 8 16 32 64 128 256 512 1024; do
  fio --rw=write \
      --bs=${iosize}k \
      --name=vdo \
      --filename=/dev/mapper/vdo-test \
      --ioengine=libaio \
      --numjobs=1 \
      --thread \
      --norandommap \
      --runtime=300 \
      --direct=1 \
      --iodepth=optimal-depth \
      --scramble_buffers=1 \
      --offset=0 \
      --size=100g
  done
```

4. VDO 테스트 볼륨을 제거합니다.

   자세한 내용은 4.3절. "VDO 성능 테스트 볼륨 정리" 의 내용을 참조하십시오.

4.5.2. VDO에서 임의의 쓰기에 I/O 요청 크기의 효과 테스트

이 테스트에서는 다양한 I/O 요청 크기의 VDO 볼륨에서 임의의 쓰기 작업을 수행하는 방식을 결정합니다.

중요

각 I/O 요청 크기 테스트 실행 간에 VDO 볼륨을 다시 생성해야 합니다.

절차

4k,8k,16k,32k,64k,128k, 256 k, 512k 및 1024k 의 I/O 요청 크기를 별도로 다음 시리즈 단계를 수행합니다.

1. 새 VDO 볼륨을 만듭니다.

   자세한 내용은 4.2절. "성능 테스트를 위한 VDO 볼륨 생성" 의 내용을 참조하십시오.

2. 테스트 볼륨에서 쓰기 fio 작업을 수행하여 테스트에서 액세스할 수 있는 모든 영역을 미리 채웁니다.

```
# fio --rw=write \
    --bs=8M \
    --name=vdo \
    --filename=/dev/mapper/vdo-test \
    --ioengine=libaio \
    --thread \
    --direct=1 \
    --scramble_buffers=1
```

3. 임의 쓰기 테스트에 대해 보고된 처리량 및 대기 시간을 기록합니다.

```
# fio --rw=randwrite \
    --bs=request-size \
    --name=vdo \
```

```
    --filename=/dev/mapper/vdo-test \
    --ioengine=libaio \
    --numjobs=1 \
    --thread \
    --norandommap \
    --runtime=300 \
    --direct=1 \
    --iodepth=optimal-depth \
    --scramble_buffers=1 \
    --offset=0 \
    --size=100g
done
```

4. VDO 테스트 볼륨을 제거합니다.

   자세한 내용은 4.3절. “VDO 성능 테스트 볼륨 정리” 의 내용을 참조하십시오.

4.5.3. VDO에서 순차적 읽기에서 I/O 요청 크기의 효과 테스트

이 테스트에서는 다양한 I/O 요청 크기에서 VDO 볼륨에서 순차 읽기 작업을 수행하는 방식을 결정합니다.

절차

1. 새 VDO 볼륨을 만듭니다.

   자세한 내용은 4.2절. “성능 테스트를 위한 VDO 볼륨 생성” 의 내용을 참조하십시오.

2. 테스트 볼륨에서 쓰기 fio 작업을 수행하여 테스트에서 액세스할 수 있는 모든 영역을 미리 채웁니다.

```
# fio --rw=write \
    --bs=8M \
    --name=vdo \
    --filename=/dev/mapper/vdo-test \
    --ioengine=libaio \
    --thread \
    --direct=1 \
    --scramble_buffers=1
```

3. 순차적 읽기 테스트에 대해 보고된 처리량 및 대기 시간을 기록합니다.

```
# for iosize in 4 8 16 32 64 128 256 512 1024; do
```

```
fio --rw=read \
    --bs=${iosize}k \
    --name=vdo \
    --filename=/dev/mapper/vdo-test \
    --ioengine=libaio \
    --numjobs=1 \
    --thread \
    --norandommap \
    --runtime=300 \
    --direct=1 \
    --iodepth=optimal-depth \
    --scramble_buffers=1 \
    --offset=0 \
    --size=100g
done
```

4. **VDO 테스트 볼륨을 제거합니다.**

   자세한 내용은 4.3절. "VDO 성능 테스트 볼륨 정리" 의 내용을 참조하십시오.

4.5.4. VDO에서 임의의 읽기에서 I/O 요청 크기의 효과 테스트

이 테스트에서는 다양한 I/O 요청 크기의 VDO 볼륨에서 임의의 읽기 작업을 수행하는 방식을 결정합니다.

절차

1. **새 VDO 볼륨을 만듭니다.**

   자세한 내용은 4.2절. "성능 테스트를 위한 VDO 볼륨 생성" 의 내용을 참조하십시오.

2. **테스트 볼륨에서 쓰기 fio 작업을 수행하여 테스트에서 액세스할 수 있는 모든 영역을 미리 채웁니다.**

```
# fio --rw=write \
    --bs=8M \
    --name=vdo \
    --filename=/dev/mapper/vdo-test \
    --ioengine=libaio \
    --thread \
    --direct=1 \
    --scramble_buffers=1
```

3. 임의 읽기 테스트에 대해 보고된 처리량 및 대기 시간을 기록합니다.

```
# for iosize in 4 8 16 32 64 128 256 512 1024; do
  fio --rw=read \
      --bs=${iosize}k \
      --name=vdo \
      --filename=/dev/mapper/vdo-test \
      --ioengine=libaio \
      --numjobs=1 \
      --thread \
      --norandommap \
      --runtime=300 \
      --direct=1 \
      --iodepth=optimal-depth \
      --scramble_buffers=1 \
      --offset=0 \
      --size=100g
  done
```

4. VDO 테스트 볼륨을 제거합니다.

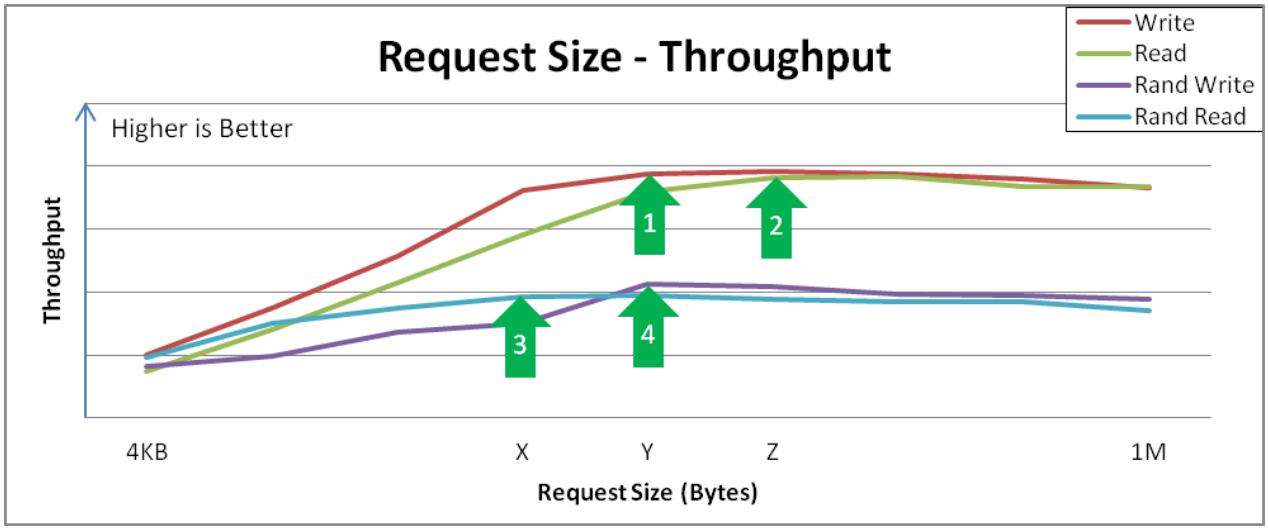
자세한 내용은 4.3절. “VDO 성능 테스트 볼륨 정리” 의 내용을 참조하십시오.

4.5.5. 다양한 I/O 요청 크기에서 VDO 성능 분석

다음 예제에서는 다양한 I/O 요청 크기에 기록된 VDO 처리량 및 대기 시간을 분석합니다.

예 4.2. I/O 요청 크기 분석

그림 4.3. 요청 크기 및 처리량 분석 및 키 변형 지점 비교



예제 결과 분석:

- 순차적 쓰기는 요청 크기 Y 에서 최대 처리량에 도달합니다.

  이 곡선은 구성 가능하거나 특정 요청 크기에 따라 자연스럽게 제어되는 애플리케이션이 성능을 인식하는 방법을 보여줍니다. 4KiB I/O 작업이 병합하면 도움이 될 수 있기 때문에 요청 크기가 커질수록 처리량이 증가합니다.

- 순차적 읽기는 지점 Z 에서 유사한 최대 처리량에 도달합니다.

  이러한 기능이 정점되면 I/O 작업이 완료되기 전의 전체 대기 시간이 추가 처리량 없이 증가합니다. 이 크기보다 큰 I/O 작업을 수락하지 않도록 장치를 튜닝해야 합니다.

- 임의의 읽기는 지점 X 에서 최대 처리량을 달성합니다.

  특정 장치는 대규모 요청 크기 임의 액세스에서 순차적 처리 속도를 거의 달성할 수 있지만, 다른 장치는 순차적으로 순차적으로 액세스하지 않는 경우 더 많은 페널티를 겪게 됩니다.

- 임의의 쓰기는 지점 Y 에서 최대 처리량을 달성합니다.

  임의 쓰기는 중복 제거 장치의 가장 많은 상호 작용을 포함하며, VDO는 특히 요청 크기 또는 I/O 깊이가 큰 경우 고성능을 달성합니다.

4.6. VDO 성능에서 혼합 I/O 부하의 효과 테스트

이 테스트에서는 VDO 구성이 혼합 읽기 및 쓰기 I/O 부하로 작동하는 방식을 확인하고 최적의 임의 대기열 깊이 및 요청 크기 4KB에서 1MB까지의 혼합 읽기 및 쓰기의 효과를 분석합니다.

이 절차에서는 고정된 I/O 깊이, 8KB ~ 256KB 범위의 다양한 블록 크기, 읽기 백분율을 10% 단위로 설정하며 0%부터 시작합니다.

사전 요구 사항

- 최적의 I/O 깊이 값을 결정했습니다. 자세한 내용은 4.4절. “VDO 성능에서 I/O 깊이의 영향 테스트” 의 내용을 참조하십시오.

다음 절차에서는 최적의 깊이를 *최적의* I/O 깊이 값으로 바꿉니다.

절차

1. 새 VDO 볼륨을 만듭니다.

   자세한 내용은 4.2절. "성능 테스트를 위한 VDO 볼륨 생성" 의 내용을 참조하십시오.

2. 테스트 볼륨에서 쓰기 fio 작업을 수행하여 테스트에서 액세스할 수 있는 모든 영역을 미리 채웁니다.

```
# fio --rw=write \
      --bs=8M \
      --name=vdo \
      --filename=/dev/mapper/vdo-test \
      --ioengine=libaio \
      --thread \
      --direct=1 \
      --scramble_buffers=1
```

3. 읽기 및 쓰기 입력 stimuls에 대해 보고된 처리량 및 대기 시간을 기록합니다.

```
# for readmix in 0 10 20 30 40 50 60 70 80 90 100; do
    for iosize in 4 8 16 32 64 128 256 512 1024; do
      fio --rw=rw \
          --rwmixread=$readmix \
          --bs=${iosize}k \
          --name=vdo \
          --filename=/dev/mapper/vdo-test \
          --ioengine=libaio \
          --numjobs=1 \
          --thread \
          --norandommap \
          --runtime=300 \
          --direct=0 \
          --iodepth=optimal-depth \
          --scramble_buffers=1 \
          --offset=0 \
          --size=100g
    done
  done
```

4. VDO 테스트 볼륨을 제거합니다.

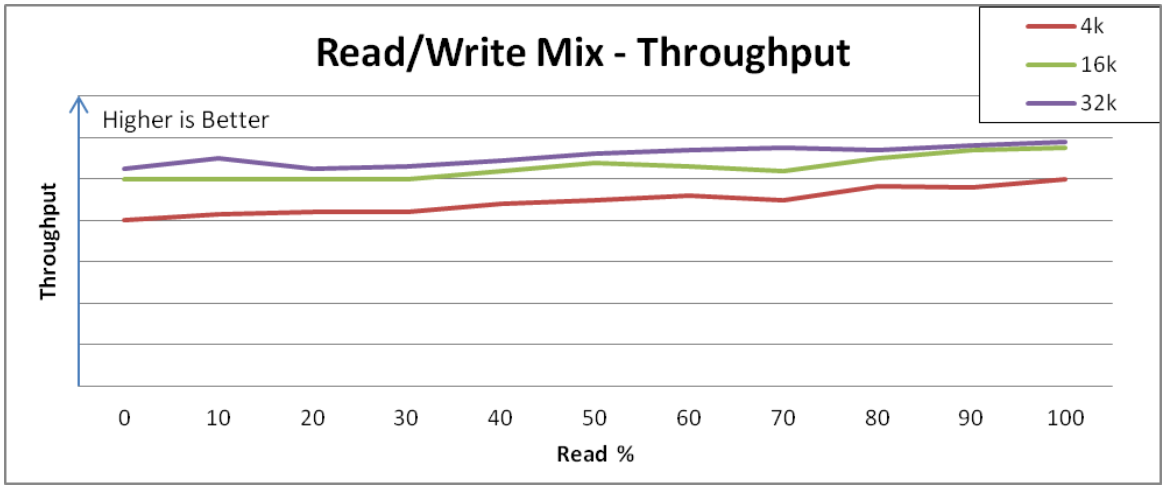
자세한 내용은 4.3절. “VDO 성능 테스트 볼륨 정리” 의 내용을 참조하십시오.

5. 테스트 결과를 그래프로 표시합니다.

예 4.3. 혼합 I/O 로드 분석

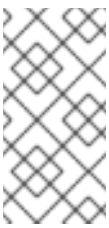
다음 이미지는 VDO가 혼합 I/O 로드에 응답하는 방법의 예를 보여줍니다.

그림 4.4. 다양한 읽기 및 쓰기 혼합에 따라 성능이 일관되게 유지됩니다.



집계 성능 및 집계 대기 시간은 읽기 및 쓰기 범위를 혼합하는 범위 전체에 비교적 일관적이며, 낮은 최대 쓰기 처리량에서 더 높은 최대 읽기 처리량으로의 추세입니다.

이 동작은 스토리지마다 다를 수 있지만 중요한 관찰은 성능이 상이한 부하에서 일관되게 유지되었거나 특정 읽기 및 쓰기 혼합을 보여주는 애플리케이션의 성능을 이해할 수 있다는 것입니다.



참고

시스템에 비슷한 응답 일관성이 표시되지 않으면 하위 최적화 구성의 신호일 수 있습니다. 이러한 경우 Red Hat 영업 엔지니어에게 문의하십시오.

4.7. VDO 성능에서 애플리케이션 환경의 영향 테스트

이 테스트에서는 혼합 실제 애플리케이션 환경에 배포 시 VDO 구성이 작동하는 방식을 결정합니다. 예상 환경에 대한 자세한 내용도 알고 있으면 테스트하십시오.

사전 요구 사항

- 구성에서 허용되는 대기열 깊이를 제한하는 것이 좋습니다.

- 가능한 경우 **VDO** 성능에 가장 유용한 블록 크기로 요청을 발행하도록 애플리케이션을 조정합니다.

절차

1. 새 **VDO** 볼륨을 만듭니다.

   자세한 내용은 **4.2**절. "성능 테스트를 위한 **VDO** 볼륨 생성" 의 내용을 참조하십시오.

2. 테스트 볼륨에서 쓰기 **fio** 작업을 수행하여 테스트에서 액세스할 수 있는 모든 영역을 미리 채웁니다.

```
# fio --rw=write \
      --bs=8M \
      --name=vdo \
      --filename=/dev/mapper/vdo-test \
      --ioengine=libaio \
      --thread \
      --direct=1 \
      --scramble_buffers=1
```

3. 읽기 및 쓰기 입력 **stimuls**에 대해 보고된 처리량 및 대기 시간을 기록합니다.

```
# for readmix in 20 50 80; do
    for iosize in 4 8 16 32 64 128 256 512 1024; do
      fio --rw=rw \
          --rwmixread=$readmix \
          --bsrange=4k-256k \
          --name=vdo \
          --filename=/dev/mapper/vdo-name \
          --ioengine=libaio \
          --numjobs=1 \
          --thread \
          --norandommap \
          --runtime=300 \
          --direct=0 \
          --iodepth=$iosize \
          --scramble_buffers=1 \
          --offset=0 \
```

```
        --size=100g
  done
done
```

4. **VDO 테스트 볼륨을 제거합니다.**

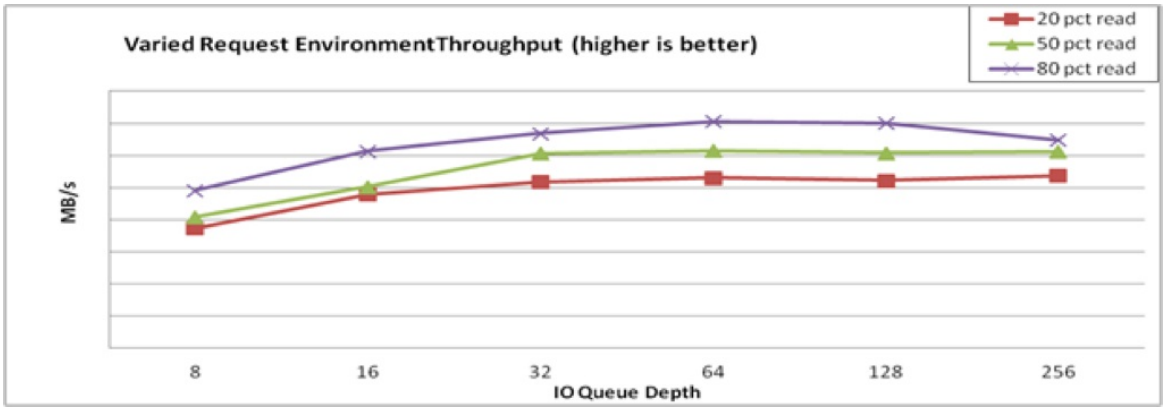
   자세한 내용은 4.3절. "VDO 성능 테스트 볼륨 정리" 의 내용을 참조하십시오.

5. **테스트 결과를 그래프로 표시합니다.**

예 4.4. 애플리케이션 환경 분석

다음 이미지는 **VDO**가 혼합 **I/O** 로드에 응답하는 방법의 예를 보여줍니다.

그림 4.5. 혼합 환경 성능



4.8. FIO를 사용한 VDO 성능 테스트에 사용되는 옵션

VDO 테스트에서는 fio 유틸리티를 사용하여 반복 가능한 특성을 가진 데이터를 엄격하게 생성합니다. 테스트에서 실제 워크로드를 시뮬레이션하려면 다음 fio 옵션이 필요합니다.

표 4.1. 사용된 fio 옵션

인수	설명	테스트에 사용되는 값
--size	**fio** 가 작업당 대상으로 전송하는 데이터의 양입니다. num **jobs 옵션도** 참조하십시오.	100GiB

인수	설명	테스트에 사용되는 값
--bs	**fio** 에서 생성한 각 읽기 및 쓰기 요청의 블록 크기입니다. Red Hat은 VDO의 기본값 4KiB와 일치하는 4KiB 블록 크기를 권장합니다.	4k
--numjobs	**fio** 가 벤치마크를 위해 생성하는 작업 수입니다. 각 작업은 **--size** 옵션으로 지정한 데이터 양을 전송합니다. 첫 번째 작업은 **--offset** 옵션으로 지정한 오프셋에 있는 장치에 데이터를 전송합니다. 이후 작업은 **--offset_increment** 옵션을 제공하지 않는 한 디스크의 동일한 지역을 덮어씁니다. 이 옵션은 이전 작업이 해당 값으로 시작된 위치에서 각 작업을 오프셋합니다. 플래시 디스크(SSD)에서 최대 성능을 얻으려면 Red Hat은 최소한 두 개의 작업을 권장합니다. 한 가지 작업은 일반적으로 HDD(재귀 디스크) 처리량을 포화하기에 충분합니다.	HDD의 경우 1, SSD의 경우 2
--thread	**fio 작업에 fork** 가 아닌 스레드에서 실행되도록 지시합니다. 이 작업은 컨텍스트 전환을 제한하여 더 나은 성능을 제공할 수 있습니다.	*none*
--ioengine	**fio** 가 벤치마크에 사용하는 I/O 엔진. Red Hat 테스팅에서는 **libaio** 라는 비동기 디버깅되지 않은 엔진을 사용하여 하나 이상의 프로세스가 동시에 임의 요청을 생성하는 워크로드를 테스트합니다. **libaio** 엔진을 사용하면 데이터를 검색하기 전에 단일 스레드에서 여러 요청을 비동기적으로 만들 수 있습니다. 이렇게 하면 많은 스레드에서 요청을 제공하는 경우 동기 엔진에 필요한 컨텍스트 전환 수가 제한됩니다.	**libaio**
--direct	옵션을 사용하면 장치에 제출된 요청을 통해 커널 페이지 캐시를 우회할 수 있습니다. **--direct** 옵션과 함께 **libaio** 엔진을 사용해야 합니다. 그렇지 않으면 커널에서 모든 I/O 요청에 동기화 API를 사용합니다.	1 (**libaio**)
--iodepth	언제든지 이동 중인 I/O 버퍼 수입니다. 값이 높으면 특히 임의 읽기 또는 쓰기의 경우 성능이 향상됩니다. 높은 값을 사용하면 컨트롤러에 항상 배치 요청이 있습니다. 그러나 값을 너무 높게 설정하면(일반적으로 1K보다 크면 바람직하지 않은 대기 시간이 발생할 수 있습니다. Red Hat은 128에서 512 사이의 값을 권장합니다. 최종 값은 장단점이며 애플리케이션에서 대기 시간을 허용하는 방법에 따라 달라집니다.	128 최소

인수	설명	테스트에 사용되는 값
--iodepth_batch_submit	I/O 깊이 버퍼 풀이 비어 있기 시작할 때 생성할 I/O 요청 수입니다. 이 옵션은 테스트 중에 I/O 작업에서 버퍼 생성으로 작업을 제한합니다.	16
--iodepth_batch_complete	배치를 제출하기 전에 완료할 I/O 작업 수입니다. 이 옵션은 테스트 중에 I/O 작업에서 버퍼 생성으로 작업을 제한합니다.	16
--gtod_reduce	대기 시간을 계산하기 위해 시간 호출을 비활성화합니다. 이 설정은 활성화된 경우 처리량을 줄입니다. 대기 시간 측정이 필요하지 않은 경우 옵션을 활성화합니다.	1

5장. 사용하지 않는 블록 삭제

해당 장치를 지원하는 블록 장치에서 삭제 작업을 수행하거나 예약할 수 있습니다.

5.1. 블록 삭제 작업

블록 삭제 작업에서는 마운트된 파일 시스템에서 더 이상 사용하지 않는 블록을 삭제합니다. 이는 다음에서 유용합니다.

- SSD(반도체 드라이브)
- 씬 프로비저닝 스토리지

요구 사항

파일 시스템의 기본 블록 장치는 물리적 삭제 작업을 지원해야 합니다.

/sys/block/*장치*/queue/discard_max_bytes 파일의 값이 **0**이 아닌 경우 물리적 삭제 작업이 지원됩니다.

5.2. 블록 삭제 작업 유형

다양한 방법을 사용하여 삭제 작업을 실행할 수 있습니다.

배치 삭제

사용자가 명시적으로 실행합니다. 선택한 파일 시스템에서 사용되지 않은 모든 블록을 폐기합니다.

온라인 삭제

마운트 시 지정됩니다. 사용자 개입 없이 실시간으로 실행됩니다. 온라인 삭제 작업은 사용에서 자유로 전환되는 블록만 폐기합니다.

주기적인 삭제

은 **systemd** 서비스에서 정기적으로 실행하는 배치 작업입니다.

모든 유형은 **XFS** 및 **ext4** 파일 시스템과 **VDO**에서 지원합니다.

권장 사항

배치 또는 정기적 삭제를 사용하는 것이 좋습니다.

다음과 같은 경우에만 온라인 삭제를 사용하십시오.

- 시스템의 워크로드를 배치 삭제를 사용할 수 없거나
- 성능을 유지하려면 온라인 삭제 작업이 필요합니다.

5.3. 배치 블록 삭제 수행

다음 절차에서는 배치 블록 삭제 작업을 수행하여 마운트된 파일 시스템에서 사용하지 않는 블록을 폐기합니다.

사전 요구 사항

- 파일 시스템이 마운트됩니다.
- 파일 시스템의 기본 블록 장치는 물리적 삭제 작업을 지원합니다.

절차

- **fstrim** 유틸리티를 사용합니다.
  - 선택한 파일 시스템에서만 삭제하려면 다음을 사용합니다.

```
# fstrim mount-point
```

  - 마운트된 모든 파일 시스템에서 삭제하려면 다음을 사용합니다.

```
# fstrim --all
```

fstrim 명령을 실행하는 경우 다음을 실행합니다.

- 삭제 작업을 지원하지 않는 장치 또는
- 여러 장치로 구성된 **LVM** 또는 **MD.** 장치 중 하나에서 삭제 작업을 지원하지 않는 논리 장치 **(LVM** 또는 **MD)**

다음 메시지가 표시됩니다.

```
# fstrim /mnt/non_discard

fstrim: /mnt/non_discard: the discard operation is not supported
```

추가 리소스

- **fstrim(8)** 도움말 페이지.

5.4. 온라인 블록 삭제 활성화

이 절차에서는 지원되는 모든 파일 시스템에서 사용하지 않는 블록을 자동으로 삭제하는 온라인 블록 삭제 작업을 활성화합니다.

절차

- 마운트 시 온라인 삭제를 활성화합니다.
  - 파일 시스템을 수동으로 마운트할 때 **-o discard** 마운트 옵션을 추가합니다.

    ```
    # mount -o discard device mount-point
    ```

  - 파일 시스템을 영구적으로 마운트할 때 **/etc/fstab** 파일의 마운트 항목에 삭제 옵션을 추가합니다.

추가 리소스

- **mount(8)** 도움말 페이지.

- **fstab(5)** 도움말 페이지.

5.5. 주기적인 블록 삭제 활성화

이 절차에서는 지원되는 모든 파일 시스템에서 사용하지 않는 블록을 정기적으로 삭제하는 **systemd** 타이머를 활성화합니다.

절차

- **systemd** 타이머를 활성화하고 시작합니다.

```
# systemctl enable --now fstrim.timer
```

6장. 영구 이름 지정 속성 개요

시스템 관리자는 영구 명명 속성을 사용하여 여러 시스템 부팅을 통해 신뢰할 수 있는 스토리지 설정을 빌드하는 스토리지 볼륨을 참조해야 합니다.

6.1. 비영구적 명명 속성의 단점

Red Hat Enterprise Linux는 스토리지 장치를 식별하는 다양한 방법을 제공합니다. 특히 드라이브에 설치하거나 다시 포맷할 때 잘못된 장치에 실수로 액세스하지 않으려면 올바른 옵션을 사용하여 각 장치를 식별하는 것이 중요합니다.

전통적으로 **/dev/sd***(주 번호)*형식의 비영구적 이름은 **Linux**에서 스토리지 장치를 참조하기 위해*사용됩니다*. 주요 번호 범위 및 부 번호 범위 및 관련 **sd** 이름은 감지되면 각 장치에 할당됩니다. 즉, 장치 탐지 순서가 변경되면 주요 번호 범위와 연결된 **sd** 이름 간의 연결이 변경될 수 있습니다.

이러한 순서 변경은 다음과 같은 상황에서 발생할 수 있습니다.

- 시스템 부팅 프로세스의 병렬화는 각 시스템 부팅 시 다른 순서로 스토리지 장치를 감지합니다.
- 디스크의 전원을 켜거나 **SCSI** 컨트롤러에 응답하지 못합니다. 그러면 일반 장치 프로브에서 검색되지 않습니다. 디스크는 시스템에서 액세스할 수 없으며 그 이후의 장치에는 연결된 **sd** 이름이 아래로 이동한 등 주요 번호 및 부 번호가 있습니다. 예를 들어 일반적으로 **sdb** 라고 하는 디스크를 탐지하지 않으면 일반적으로 **as sdc** 라고 하는 디스크가 **sdb** 로 표시됩니다.
- **SCSI** 컨트롤러(호스트 버스 어댑터 또는 **HBA**)를 초기화하지 못하면 해당 **HBA**에 연결된 모든 디스크를 탐지하지 않습니다. 나중에 조사하는 **HBA**에 연결된 모든 디스크에는 서로 다른 주요 및 부 번호 범위가 할당되며 다양한 관련 **sd** 이름이 할당됩니다.
- 시스템에 다른 유형의 **HBA**가 있으면 드라이버 초기화 순서가 변경됩니다. 이로 인해 해당 **HBA**에 연결된 디스크가 다른 순서로 탐지됩니다. **HBA**가 시스템의 다른 **PCI** 슬롯으로 이동되는 경우에도 발생할 수 있습니다.
- **Fibre** 채널, **iSCSI** 또는 **FCoE** 어댑터를 사용하여 시스템에 연결된 디스크는 스토리지 장치를 프로브할 때 액세스할 수 없을 수 있습니다(예: 스토리지 어레이 또는 교차 스위치의 전원이 꺼짐). 이 문제는 정전 후 시스템을 재부팅할 때 시스템을 부팅하는 것보다 스토리지 어레이가 온라인 상태가 되는 데 시간이 오래 걸리는 경우 발생할 수 있습니다. 일부 파이버 채널 드라이버는

WWPN 매핑에 영구 **SCSI** 대상 **ID**를 지정하는 메커니즘을 지원하지만 이는 주요 번호 범위와 연결된 **sd** 이름을 예약하지 않습니다. 일관된 **SCSI** 대상 **ID** 번호만 제공합니다.

이러한 이유로 **/etc/fstab** 파일에서와 같이 장치를 참조할 때 주 번호 범위 또는 관련 **sd** 이름을 사용하는 것이 바람직하지 않게 합니다. 잘못된 장치가 마운트되고 데이터 손상이 발생할 수 있습니다.

그러나 경우에 따라 장치가 오류를 보고하는 경우와 같이 다른 메커니즘을 사용하는 경우에도 **sd** 이름을 참조해야 합니다. 이는 **Linux** 커널이 장치와 관련된 커널 메시지에서 **sd** 이름(및 **SCSI host/channel/target/LUNples**)을 사용하기 때문입니다.

6.2. 파일 시스템 및 장치 식별자

이 섹션에서는 파일 시스템 및 블록 장치를 식별하는 영구 속성의 차이점에 대해 설명합니다.

파일 시스템 식별자

파일 시스템 식별자는 블록 장치에 생성된 특정 파일 시스템과 연결됩니다. 식별자는 파일 시스템의 일부로도 저장됩니다. 파일 시스템을 다른 장치로 복사해도 여전히 동일한 파일 시스템 식별자를 전달합니다. 반면, **mkfs** 유틸리티로 포맷하는 등 장치를 다시 작성하는 경우 장치는 속성이 손실됩니다.

파일 시스템 식별자는 다음과 같습니다.

- 고유 식별자 **(UUID)**
- 레이블

장치 식별자

장치 식별자는 블록 장치(예: 디스크 또는 파티션)에 연결됩니다. **mkfs** 유틸리티로 포맷하는 등의 장치를 다시 작성하는 경우 장치는 파일 시스템에 저장되지 않기 때문에 속성을 유지합니다.

장치 식별자는 다음과 같습니다.

- **WWID(WWID)**

- 파티션 **UUID**

- 일련 번호

권장 사항

- 논리 볼륨과 같은 일부 파일 시스템은 여러 장치에 걸쳐 있습니다. **Red Hat**은 장치 식별자가 아닌 파일 시스템 식별자를 사용하여 이러한 파일 시스템에 액세스하는 것을 권장합니다.

6.3. /DEV/DISK/의 UDEV 메커니즘에서 관리하는 장치 이름

이 섹션에는 **udev** 서비스가 **/dev/disk/** 디렉터리에 제공하는 다양한 종류의 영구 명명 속성이 나열되어 있습니다.

udev 메커니즘은 저장 장치뿐만 아니라 **Linux**의 모든 유형의 장치에 사용됩니다. 스토리지 장치의 경우 **Red Hat Enterprise Linux**에는 **/dev/disk/** 디렉터리에 심볼릭 링크를 생성하는 **udev** 규칙이 포함되어 있습니다. 이를 통해 다음을 통해 스토리지 장치를 참조할 수 있습니다.

- 해당 콘텐츠

- 고유 식별자

- 일련 번호.

udev 명명 속성은 영구적이지만 시스템 재부팅 시 자체적으로 변경되지 않지만 일부 속성도 구성할 수 있습니다.

6.3.1. 파일 시스템 식별자

/dev/disk/by-uuid/의 **UUID** 속성

이 디렉터리의 항목은 장치에 저장된 콘텐츠(즉, 데이터)의 고유 식별자 **(UUID)**로 스토리지 장치를 참조하는 심볼릭 이름을 제공합니다. 예를 들면 다음과 같습니다.

```
/dev/disk/by-uuid/3e6be9de-8139-11d1-9106-a43f08d823a6
```

UUID를 사용하여 다음 구문으로 **/etc/fstab** 파일의 장치를 참조할 수 있습니다.

UUID=*3e6be9de-8139-11d1-9106-a43f08d823a6*

파일 시스템을 만들 때 **UUID** 속성을 구성할 수 있으며 나중에 변경할 수도 있습니다.

/dev/disk/by-label/의 **Label** 속성

이 디렉터리의 항목은 장치에 저장된 콘텐츠(즉, 데이터)의 레이블 별로 스토리지 장치를 참조하는 심볼릭 이름을 제공합니다.

예를 들면 다음과 같습니다.

/dev/disk/by-label/*Boot*

라벨을 사용하여 다음 구문으로 **/etc/fstab** 파일의 장치를 참조할 수 있습니다.

LABEL=*Boot*

파일 시스템을 생성할 때 **Label** 속성을 구성할 수 있으며 나중에 변경할 수도 있습니다.

6.3.2. 장치 식별자

/dev/disk/by-id/의 **WWID** 속성

WWID(World Wide Identifier)는 **SCSI** 표준에서 모든 **SCSI** 장치에서 요구하는 영구적인 시스템 독립 식별자 입니다. **WWID** 식별자는 모든 스토리지 장치에 대해 고유하며 장치에 액세스하는 데 사용되는 경로와 독립적입니다. 식별자는 장치의 속성이지만 장치의 콘텐츠(즉, 데이터)에 저장되지 않습니다.

이 식별자는 장치 식별 **Vital Product Data(0 x83** 페이지) 또는 유닛 일련 번호 **(0x80**페이지)를 검색하기 위해 **SCSI Inquiry**를 발행하여 가져올 수 있습니다.

Red Hat Enterprise Linux는 **WWID** 기반 장치 이름에서 해당 시스템의 현재 **/dev/sd** 이름으로의 적절한 매핑을 자동으로 유지합니다. 애플리케이션은 **/dev/disk/by-id/** 이름을 사용하여 장치 경로가 변경되더라도, 여러 시스템에서 장치에 액세스하는 경우에도 디스크의 데이터를 참조할 수 있습니다.

예 6.1. WWID 매핑

WWID symlink	비영구적 장치	참고
/dev/disk/by-id/scsi-3600508b400105e210000900000490000	**/dev/sda**	페이지가 **0x83** 식별자인 장치
/dev/disk/by-id/scsi-SSEAGATE_ST373453LW_3HW1RHM6	**/dev/sdb**	페이지가 **0x80** 식별자인 장치
/dev/disk/by-id/ata-SAMSUNG_MZNLN256HMHQ-000L7_S2WDNX0J336519-part3	**/dev/sdc3**	디스크 파티션

시스템에서 제공하는 이러한 영구 이름 외에도 **udev** 규칙을 사용하여 스토리지의 **WWID**에 매핑되는 고유한 영구 이름을 구현할 수도 있습니다.

/dev/disk/by-partuuid의 파티션 **UUID** 속성

PARTUUID(파티션 **UUID**) 특성은 **GPT** 파티션 테이블에 정의된 파티션을 식별합니다.

예 6.2. 파티션 UUID 매핑

PARTUUID symlink	비영구적 장치
/dev/disk/by-partuuid/4cd1448a-01	**/dev/sda1**
/dev/disk/by-partuuid/4cd1448a-02	**/dev/sda2**
/dev/disk/by-partuuid/4cd1448a-03	**/dev/sda3**

/dev/disk/by-path/의 **Path** 속성

이 특성은 장치에 액세스하는 데 사용되는 하드웨어 경로에서 스토리지 장치를 참조하는 심볼릭 이름을 제공합니다.

Path 속성은 하드웨어 경로의 일부(예: **PCI ID**, 대상 포트 또는 **LUN** 번호)가 변경되면 실패합니다. 따라서 **Path** 속성은 신뢰할 수 없습니다. 그러나 **Path** 속성은 다음 시나리오 중 하나에서 유용할 수 있습니다.

- 나중에 교체할 디스크를 식별해야 합니다.

- 특정 위치에 있는 디스크에 스토리지 서비스를 설치할 계획입니다.

6.4. DM MULTIPATH를 사용한 전 세계 식별자

이 섹션에서는 장치 매퍼 다중 경로 구성의 WWID(World Wide Identifier)와 비영구적 장치 이름 간의 매핑에 대해 설명합니다.

시스템에서 장치로 이동하는 경로가 여러 개인 경우 DM Multipath는 WWID를 사용하여 이를 탐지합니다. DM Multipath는 /dev/mapper/wwid 디렉토리에 단일 "pseudo-device"를 제공합니다(예: /dev/mapper/3600508b400105df70000e00000ac0000).

multipath -l 명령은 비영구 식별자에 대한 매핑을 표시합니다.

- *Host*:*Channel*:*Target*:*LUN*
- /dev/sd 이름
- *major*:*마이너* 번호

예 6.3. 다중 경로 구성의 WWID 매핑

multipath -l 명령의 예제 출력:

```
3600508b400105df70000e00000ac0000 dm-2 vendor,product
[size=20G][features=1 queue_if_no_path][hwhandler=0][rw]
\_ round-robin 0 [prio=0][active]
 \_ 5:0:1:1 sdc 8:32  [active][undef]
 \_ 6:0:1:1 sdg 8:96  [active][undef]
\_ round-robin 0 [prio=0][enabled]
 \_ 5:0:0:1 sdb 8:16  [active][undef]
 \_ 6:0:0:1 sdf 8:80  [active][undef]
```

DM Multipath는 각 WWID 기반 장치 이름의 적절한 매핑을 시스템의 해당 /dev/sd 이름에 자동으로 유지합니다. 이러한 이름은 경로 변경 시 영구적이며 서로 다른 시스템에서 장치에 액세스할 때 일관적입니다.

DM Multipath의 **user_friendly_names** 기능을 사용하면 **WWID**가 **/dev/mapper/mpath*N*** 형식의 이름에 매핑됩니다. 기본적으로 이 매핑은 **/etc/multipath/bindings** 파일에서 유지됩니다. 이러한 **mpath*N*** 이름은 해당 파일이 유지되는 동안 유지됩니다.

중요

user_friendly_names 를 사용하는 경우 클러스터에서 일관된 이름을 얻으려면 추가 단계가 필요합니다.

6.5. UDEV 장치 명명 규칙의 제한 사항

다음은 **udev** 명명 규칙의 몇 가지 제한 사항입니다.

- **udev** 메커니즘이 **udev** 이벤트에 대해 **udev** 규칙이 처리될 때 스토리지 장치를 쿼리하는 기능을 사용할 수 있으므로 쿼리를 수행할 때 장치에 액세스할 수 없습니다. 서버가 서버 새시에 있지 않은 경우 파이버 채널, **iSCSI** 또는 **FCoE** 스토리지 장치에서 이 문제가 발생할 가능성이 높습니다.

- 커널은 언제든지 **udev** 이벤트를 보낼 수 있으므로 규칙이 처리되고 장치에 액세스할 수 없는 경우 **/dev/disk/by-*/** 링크가 제거될 수 있습니다.

- **udev** 이벤트가 생성되고 많은 장치가 감지되고 사용자 공간 **udev d** 서비스에서 각 이벤트를 처리하는 데 약간의 시간이 걸리는 경우와 같이 **udev** 이벤트가 생성될 때 사이에 지연이 발생할 수 있습니다. 이로 인해 커널이 장치를 감지하는 경우와 **/dev/disk/by-*/** 이름을 사용할 수 있는 시기 사이에 지연이 발생할 수 있습니다.

- 규칙에서 호출한 **blkid** 와 같은 외부 프로그램은 잠시 동안 장치를 열 수 있으므로 다른 용도로는 장치에 액세스할 수 없습니다.

- **/dev/disk/**의 **udev** 메커니즘에서 관리하는 장치 이름은 주요 릴리스 간에 변경될 수 있으므로 링크를 업데이트해야 합니다.

6.6. 영구 이름 지정 속성 나열

이 절차에서는 비영구적 스토리지 장치의 영구 이름 지정 속성을 찾는 방법을 설명합니다.

절차

- **UUID** 및 레이블 속성을 나열하려면 **lsblk** 유틸리티를 사용합니다.

```
$ lsblk --fs storage-device
```

예를 들면 다음과 같습니다.

예 6.4. 파일 시스템의 **UUID** 및 레이블 보기

```
$ lsblk --fs /dev/sda1

NAME FSTYPE LABEL UUID                                 MOUNTPOINT
sda1 xfs    Boot  afa5d5e3-9050-48c3-acc1-bb30095f3dc4 /boot
```

- **PARTUUID** 특성을 나열하려면 **--output +PARTUUID** 옵션과 함께 **lsblk** 유틸리티를 사용합니다.

```
$ lsblk --output +PARTUUID
```

예를 들면 다음과 같습니다.

예 6.5. 파티션의 **PARTUUID** 속성 보기

```
$ lsblk --output +PARTUUID /dev/sda1

NAME MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT PARTUUID
sda1   8:1    0  512M  0 part /boot      4cd1448a-01
```

- **WWID** 특성을 나열하려면 **/dev/disk/by-id/** 디렉터리에 있는 심볼릭 링크의 대상을 검사합니다. 예를 들면 다음과 같습니다.

예 6.6. 시스템의 모든 스토리지 장치의 **WWID** 보기

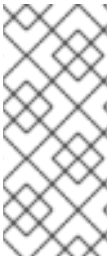
```
$ file /dev/disk/by-id/*

/dev/disk/by-id/ata-QEMU_HARDDISK_QM00001
symbolic link to ../../sda
/dev/disk/by-id/ata-QEMU_HARDDISK_QM00001-part1
symbolic link to ../../sda1
/dev/disk/by-id/ata-QEMU_HARDDISK_QM00001-part2
symbolic link to ../../sda2
```

```
/dev/disk/by-id/dm-name-rhel_rhel8-root
symbolic link to ../../dm-0
/dev/disk/by-id/dm-name-rhel_rhel8-swap
symbolic link to ../../dm-1
/dev/disk/by-id/dm-uuid-LVM-
QIWtEHtXGobe5bewlIUDivKOz5ofkgFhP0RMFsNyySVihqEl2cWWbR7MjXJolD6g
symbolic link to ../../dm-1
/dev/disk/by-id/dm-uuid-LVM-
QIWtEHtXGobe5bewlIUDivKOz5ofkgFhXqH2M45hD2H9nAf2qfWSrlRLhzfMyOKd
symbolic link to ../../dm-0
/dev/disk/by-id/lvm-pv-uuid-atlr2Y-vuMo-ueoH-CpMG-4JuH-AhEF-wu4QQm
symbolic link to ../../sda2
```

6.7. 영구 이름 지정 속성 수정

다음 절차에서는 파일 시스템의 **UUID** 또는 레이블 영구 명명 속성을 변경하는 방법을 설명합니다.



참고

udev 속성 변경은 백그라운드에서 수행되며 시간이 오래 걸릴 수 있습니다. **udevadm spring** 명령은 변경 사항이 완전히 등록될 때까지 대기하여 다음 명령이 새 특성을 올바르게 활용할 수 있도록 합니다.

다음 명령에서 다음을 수행합니다.

- ***new-uuid*** 를 설정할 **UUID**로 바꿉니다(예: **1cdfbc07-1c90-4984-b5ec-f61943f5ea50**). **uuidgen** 명령을 사용하여 **UUID**를 생성할 수 있습니다.
- ***new-label*** 을 레이블로 바꿉니다(예: **backup_data**).

사전 요구 사항

- **XFS** 파일 시스템의 속성을 수정하는 경우 먼저 마운트 해제합니다.

절차

- **XFS** 파일 시스템의 **UUID** 또는 레이블 속성을 변경하려면 **xfs_admin** 유틸리티를 사용합니다.

```
# xfs_admin -U new-uuid -L new-label storage-device
# udevadm settle
```

- **ext4, ext3** 또는 **ext 2** 파일 시스템의 **UUID** 또는 레이블 속성을 변경하려면 **tune2fs** 유틸리티를 사용합니다.

```
# tune2fs -U new-uuid -L new-label storage-device
# udevadm settle
```

- 스왑 볼륨의 **UUID** 또는 레이블 속성을 변경하려면 **swaplabel** 유틸리티를 사용합니다.

```
# swaplabel --uuid new-uuid --label new-label swap-device
# udevadm settle
```

7장. 웹 콘솔을 사용하여 가상 데이터 최적화 볼륨 관리

RHEL 8 웹 콘솔을 사용하여 VDO(가상 데이터 최적화 도구)를 구성합니다.

다음을 수행하는 방법을 배우게 됩니다.

- VDO 볼륨 만들기
- VDO 볼륨 포맷
- VDO 볼륨 확장

사전 요구 사항

- RHEL 8 웹 콘솔이 설치되고 액세스할 수 있습니다. 자세한 내용은 웹 콘솔 설치를 참조하십시오.
- cockpit-storaged 패키지가 시스템에 설치되어 있습니다.

7.1. 웹 콘솔의 VDO 볼륨

Red Hat Enterprise Linux 8은 VDO(Virtual Data Optimizer)를 지원합니다.

VDO는 다음을 결합한 블록 가상화 기술입니다.

압축

자세한 내용은 VDO에서 압축 활성화 또는 비활성화를 참조하십시오.

중복 제거

자세한 내용은 VDO에서 압축 활성화 또는 비활성화를 참조하십시오.

씬 프로비저닝

자세한 내용은 씬 프로비저닝된 볼륨(타사 볼륨) 생성 및 관리를 참조하십시오.

이러한 기술을 사용하여 VDO:

- 스토리지 공간 인라인 저장
- 파일 압축
- 복제 제거
- 물리적 또는 논리적 스토리지가 제공하는 것보다 더 많은 가상 공간을 할당할 수 있습니다.
- 확장을 통해 가상 스토리지를 확장할 수 있습니다

VDO는 여러 유형의 스토리지 위에 만들 수 있습니다. RHEL 8 웹 콘솔에서 다음과 같이 VDO를 구성할 수 있습니다.

- LVM

  참고

  씬 프로비저닝된 볼륨 위에 VDO를 구성할 수 없습니다.

- 물리 볼륨
- 소프트웨어 RAID

스토리지 스택의 VDO 배치에 대한 자세한 내용은 시스템 요구 사항을 참조하십시오.

추가 리소스

- VDO에 대한 자세한 내용은 스토리지 분할 및 압축을 참조하십시오.

7.2. 웹 콘솔에서 VDO 볼륨 생성

RHEL 웹 콘솔에서 VDO 볼륨을 만듭니다.

사전 요구 사항

- VDO를 생성하려는 물리 드라이브, LVM 또는 RAID.

절차

1. RHEL 8 웹 콘솔에 로그인합니다.

   자세한 내용은 웹 콘솔에 로그인을 참조하십시오.

2. Storage(스토리지)를 클릭합니다.

3. VDO Devices (VDO 장치) 상자에서 + 아이콘을 클릭합니다.

4. Name (이름) 필드에 공백 없이 VDO 볼륨의 이름을 입력합니다.

5. 사용할 드라이브를 선택합니다.

6. Logical Size 표시줄에서 VDO 볼륨의 크기를 설정합니다. 10회 이상 확장할 수 있지만 VDO 볼륨을 생성할 용도를 고려하십시오.

   - 활성 VM 또는 컨테이너 스토리지의 경우 볼륨의 물리적 크기 10배인 논리 크기를 사용합니다.

   - 오브젝트 스토리지의 경우 볼륨의 물리적 크기 세 배인 논리 크기를 사용합니다.

자세한 내용은 Deploying VDO 을 참조하십시오.

7. **Index Memory** 표시줄에서 VDO 볼륨에 메모리를 할당합니다.

   VDO 시스템 요구 사항에 대한 자세한 내용은 시스템 요구 사항을 참조하십시오.

8. **Compression** (압축) 옵션을 선택합니다. 이 옵션은 다양한 파일 형식을 효율적으로 줄일 수 있습니다.

   자세한 내용은 VDO에서 압축 활성화 또는 비활성화를 참조하십시오.

9. **Deduplication** (삭제) 옵션을 선택합니다.

   이 옵션을 사용하면 중복 블록 복사본을 여러 개 제거하여 스토리지 리소스의 소비를 줄입니다. 자세한 내용은 VDO에서 압축 활성화 또는 비활성화를 참조하십시오.

10. [선택 사항] 512바이트 블록 크기가 필요한 애플리케이션에서 VDO 볼륨을 사용하려면 **Use 512** 바이트 에뮬레이션 사용을 선택합니다. 따라서 VDO 볼륨의 성능이 저하되지만 거의 필요하지 않습니다. 의심스러운 경우 남겨 두십시오.

11. 생성을 클릭합니다.

검증 단계

- **Storage** (스토리지) 섹션에서 새 VDO 볼륨이 표시되는지 확인합니다. 그러면 파일 시스템으로 포맷할 수 있습니다.

7.3. 웹 콘솔에서 VDO 볼륨 포맷

VDO 볼륨은 물리적 드라이브 역할을 합니다. 사용하려면 파일 시스템으로 포맷해야 합니다.

주의

VDO 포맷은 볼륨의 모든 데이터를 지웁니다.

다음 단계에서는 VDO 볼륨을 포맷하는 절차를 설명합니다.

사전 요구 사항

- VDO 볼륨이 생성됩니다. 자세한 내용은 웹 콘솔에서 VDO 볼륨 생성을 참조하십시오.

절차

1. RHEL 8 웹 콘솔에 로그인합니다. 자세한 내용은 웹 콘솔에 로그인을 참조하십시오.
2. Storage(스토리지)를 클릭합니다.
3. VDO 볼륨을 클릭합니다.
4. Uncognized Data(인식되지 않은 데이터) 탭을 클릭합니다.
5. Format(형식)을 클릭합니다.
6. Erase 드롭다운 메뉴에서 다음을 선택합니다.

기존 데이터를 덮어쓰지 마십시오.

RHEL 웹 콘솔은 디스크 헤더만 다시 작성합니다. 이 옵션의 장점은 포맷 속도입니다.

기존 데이터를 제로로 덮어쓰기

RHEL 웹 콘솔은 전체 디스크를 0으로 다시 작성합니다. 프로그램이 전체 디스크를 통과해야 하므로 이 옵션이 더 느립니다. 디스크에 데이터가 포함되어 다시 작성해야 하는 경우 이

옵션을 사용합니다.

7. **Type** (유형) 드롭다운 메뉴에서 파일 시스템을 선택합니다.

- **XFS** 파일 시스템은 대용량 논리 볼륨을 지원하고, 시스템 중단 없이 온라인에서 물리적 드라이브를 전환하며 늘어나는 작업을 지원합니다. 다른 기본 설정이 없는 경우 이 파일 시스템을 선택한 상태로 둡니다.

  XFS는 볼륨 축소를 지원하지 않습니다. 따라서 **XFS**로 포맷된 볼륨을 줄일 수 없습니다.

- **ext4** 파일 시스템은 시스템 중단, 증가 및 축소 없이 물리적 드라이브를 온라인으로 전환하는 논리 볼륨을 지원합니다.

LUKS(Linux Unified Key Setup) 암호화로 버전을 선택할 수도 있으며, 이를 통해 암호를 사용하여 볼륨을 암호화할 수도 있습니다.

8. **Name** (이름) 필드에 논리 볼륨 이름을 입력합니다.

9. **Mounting**(마운팅) 드롭다운 메뉴에서 **Custom**(사용자 지정)을 선택합니다.

   Default(기본값) 옵션은 다음 부팅 시 파일 시스템이 마운트되는지 확인하지 않습니다.

10. **Mount Point**(마운트 지점) 필드에 마운트 경로를 추가합니다.

11. 부팅 시 마운트를 선택합니다.

12. **Format**(형식)을 클릭합니다.

    포맷은 사용된 포맷 옵션 및 볼륨 크기에 따라 몇 분이 걸릴 수 있습니다.

성공적으로 완료되면 **Filesystem**(파일 시스템) 탭에서 포맷된 **VDO** 볼륨의 세부 정보를 확인할 수 있습니다.

13. VDO 볼륨을 사용하려면 **Mount** (마운트)를 클릭합니다.

이때 시스템은 마운트되고 포맷된 VDO 볼륨을 사용합니다.

7.4. 웹 콘솔에서 VDO 볼륨 확장

RHEL 8 웹 콘솔에서 VDO 볼륨을 확장합니다.

사전 요구 사항

- **cockpit-storaged** 패키지가 시스템에 설치되어 있습니다.
- VDO 볼륨이 생성되었습니다.

절차

1. RHEL 8 웹 콘솔에 로그인합니다.

   자세한 내용은 웹 콘솔에 로그인을 참조하십시오.

2. **Storage**(스토리지)를 클릭합니다.

3. VDO 장치 상자에서 VDO 볼륨을 클릭합니다.

4. VDO 볼륨 세부 사항에서 **Grow** 단추를 클릭합니다.

5. VDO 대화 상자의 논리 크기에서 VDO 볼륨의 논리 크기를 확장합니다.

1. **Grow**(검색)를 클릭합니다.

검증 단계

- 새 크기에 대한 VDO 볼륨 세부 정보를 확인하여 변경 사항이 성공했는지 확인합니다.