FreeBSD Mastery: ZFS

Michael W. Lucas와 Allan Jude의 FreeBSD Mastery: ZFS를 번역한 것입니다 .

FreeBSD관련 책은 거의 나오는 것이 없으므로 가능한한 책도 좀 사주시길 부탁드립니다 .

이건 제가 한글로 보고 싶어서 번역한 내용입니다 .

챕터 0: 소개

대부분의 시스템 관리 교육은 파일시스템에 초점을 맞춥니다. 컴퓨터의 파일시스템은 컴퓨터의 성능과 동작의 많은 부분을 좌우합니다. 지난 수십 년 동안 우리는 주요 파일시스템이 잘못 구성되었거나, 선택한 파일시스템이 작업에 적합하지 않았거나, 파일 전체에 미묘한 파일시스템 손상이 확산되어 운영체제와 함께 제공된 기본 프로그램조차 신뢰할 수 없게 되어 시스템 전체를 재구축한 적이 있습니다. 시스템 관리자로 몇 년 이상 근무한 사람이라면 누구나 파일 시스템을 복구하고, 파일 시스템을 재구성하고, 성가신 버그부터 치명적인 버그까지 해결하고, 파일 시스템 제한을 지원하기 위해 디스크를 재정렬하고, 9개 이상의 언어로 파일 시스템에 대해 광범위하게 욕하는 방법을 배웠을 것입니다.

오늘날 가장 널리 사용되는 파일시스템 중 일부는 컴퓨팅 규모로 볼 때 오래된 것입니다. 우리는 5년이 지나 너무 느리다는 이유로 하드웨어를 폐기하고 그 자리에 30년 된 파일시스템을 설치합니다. extfs, UFS2, NTFS와 같은 최신 파일시스템도 그 핵심에 오래된 아이디어를 사용합니다. Z 파일 시스템, 즉 ZFS는 이 모든 것을 바꾸기 위해 등장했습니다.

ZFS란 무엇인가?

ZFS는 파일시스템이 데이터 무결성을 보장할 수 있어야 한다는 핵심 아이디어로 구축된 최신 파일시스템입니다. ZFS는 디스크의 모든 데이터에 대해 체크섬을 계산하여 스토리지 미디어에 오류가 발생하여 데이터가 손상된 시점을 식별할 수 있습니다. 모든 메타데이터에 대해 동일한 암호화 서명을 수행합니다. 기본 하드웨어에 문제가 발생하거나 오작동이 발생하면 ZFS는 검색한 데이터가 기록과 일치하지 않음을 인식하고 조치를 취할 수 있습니다. 심지어 ZFS는 발견된 오류를 자동으로 수정하기도 합니다! ZFS는 손상된 것으로 확인된 데이터의 제공을 거부합니다.

파일시스템 설계자들은 30년 전에 이러한 아이디어를 가지고 있었지만, 당시의 하드웨어로는 이 정도의 오류 검사를 적절한 성능으로 수행할 수 없었습니다. ZFS의 개발자들은 현재의 하드웨어와 하드웨어의 발전 방향을 살펴본 후, ZFS가 새로운 하드웨어를 최대한 활용하기로 결정했습니다. 그 결과 기존 파일시스템보다 더 안정적일 뿐만 아니라 더 빠른 파일시스템이 탄생했습니다.

오늘날의 전통적인 파일시스템은 "지금은 이 정도면 충분하다"는 철학으로 작성된 것으로 보입니다. 많은 파일시스템은 임의의 크기 제한으로 인해 5년, 10년, 심지어 20년 동안은 충분했지만 결국 재작성 및 재작업이 필요했습니다. 많은 구형 파일시스템은 2기가바이트보다 큰 파티션을 처리할 수 없었는데, 요즘은 병따개에 부착해 무료로 제공하는 플래시 드라이브보다 작은 크기입니다. (실제로는 병따개가 필요해서 구입한 것입니다.) 하지만 1980년대 초 UFS가 처음 출시되었을 때 2기가바이트는 수백만 달러에 달하는 엄청나게 큰 용량의 스토리지였습니다. FAT와 같은 파일시스템은 360KB 플로피 디스크의 공간을 효율적으로 사용해야 했습니다. UFS는 당장은 물론 앞으로도 당분간 '충분'했습니다.

ZFS는 가까운 미래에도 살아남을 수 있도록 의도적으로 설계되었습니다. 많은 새로운 파일시스템은 내부적으로 64비트 식별자와 인덱스를 사용하므로 향후 10년 또는 20년 동안 변경 없이 사용할 수 있습니다. ZFS는 내부적으로 128비트 식별자를 사용하므로 향후 수천 년 동안 스토리지 시스템에서 작동할 수 있는 충분한 용량을 제공합니다. 스타트렉에 등장하는 엔터프라이즈호의 컴퓨터는 아마도 ZFS를 실행할 것입니다. ZFS의 기본 제공 제약을 초과하는 디스크, 파티션 및 파일을 처리해야 하는 미래의 시스템 관리자는 우리를

원시인 및 최초의 성간 여행자들과 함께 역사에 남게 될 것입니다.

엄밀히 말하면 ZFS는 단순한 파일 시스템이 아닙니다. 파일시스템과 볼륨 관리자가 결합된 형태입니다. 이 두 가지 기능을 하나의 소프트웨어에 결합하면 나중에 설명하겠지만 몇 가지 제한이 따르지만 매우 흥미로운 작업도 가능합니다. ZFS는 디스크에서 데이터가 어디로 가는지 정확히 알고 있기 때문에 파일과 스트라이프를 위에서 아래로 최적으로 배열할 수 있습니다. ZFS는 보조 고속 스토리지를 특수 목적의 캐시로 사용하여 성능을 더욱 향상시킬 수 있습니다. FreeBSD 마스터리: ZFS 에센셜은 이 최신의 고성능 미래형 파일 시스템을 실행하기 위해 반드시 알아야 할 사항을 안내합니다.

ZFS 역사

매트 아렌스와 제프 본윅은 썬 마이크로시스템즈의 솔라리스® 운영 체제용 ZFS를 만들었습니다. Sun은 모든 규모의 시스템을 판매했지만, 주로 하이엔드 서버 하드웨어에 주력했습니다. Sun 하드웨어는 전 세계 대부분의 대형 데이터베이스를 구동했습니다. Sun은 공통 개발 및 배포 라이선스(CDDL)에 따라 ZFS 소스 코드를 전 세계에 제공했습니다. 사람들은 FreeBSD를 비롯한 다른 운영 체제로 ZFS를 포팅하고 변경 사항을 다시 Sun에 기여하기 시작했습니다.

그러던 중 오라클이 썬 마이크로시스템을 인수했습니다. 오라클은 MySQL과 같은 일부 오픈 소스 소프트웨어를 보유하고 있지만, 대부분의 소프트웨어는 독점 소프트웨어입니다. 오라클은 ZFS 개발을 완전히 사내로 가져왔고 오픈 소스 라이선스에 따른 소스 코드 제공을 중단했습니다.

하지만 ZFS 코드는 이미 공개되어 있었고, CDDL 라이선스 조건에 따라 오라클은 사람들이 이를 사용하는 것을 막을 수 없었습니다. 다양한 오픈 소스 프로젝트가 자체적인 ZFS 개발 노력을 시작했습니다.

현재 OpenZFS 프로젝트(http://open-zfs.org)가 오픈 소스 버전의 ZFS를 총괄하고 있습니다. OpenZFS는 Linux, OS X, Illumos, FreeBSD를 비롯한 여러 회사 및 운영 체제의 ZFS 개발자를 한데 모으고 있습니다. 매트 아렌스가 이 프로젝트를 이끌고 있습니다.

다른 FreeBSD와 달리, ZFS는 제한적인 라이선스를 가지고 있습니다. CDDL은 특허 소송을 제기할 수 있는 능력을 제한하며, ZFS에 대한 기여는 자동으로 CDDL의 적용을 받습니다. FreeBSD의 2항 BSD 라이선스는 누구나 특허 소송을 포함한 모든 용도로 코드를 사용할 수 있도록 허용합니다. 그러나 CDDL은 코드의 재사용, 재배포 및 변경을 허용하므로 일반적인 기업 환경에서는 누구나 코드를 사용할 수 있습니다. 하지만 ZFS를 기반으로 제품을 만들거나 다른 제품에 ZFS를 포함하려면 변호사와 상담하세요.

사전 요구 사항

이 책은 ZFS에 관심이 있는 FreeBSD 시스템 관리자를 위해 작성되었습니다. 우리는 여러분이 설치, 사용자 구성, GEOM 기반 스토리지 관리 등 FreeBSD의 기본 사항에 익숙하다고 가정합니다. "스토리지 제공자"가 무엇이며 왜 우리가 이 용어를 사용하는지 알고 계실 것입니다. 자신의 기술에 대해 확신이 서지 않는다면, Absolute FreeBSD(No Starch Press, 2007)와 같은 책이나 다른 FreeBSD 마스터리 타이틀을 구입하여 지식을 보강하거나 온라인 문서와 매뉴얼 페이지를 참조할 수 있습니다.

FreeBSD가 아닌 다른 운영 체제에서 OpenZFS를 사용하는 경우, 이 책은 실용적인 ZFS 지식과 경험을 많이 제공합니다. FreeBSD와 관련된 내용은 무시해도 되지만, ZFS를 최적화하고 관리하는 방법을 배울 수 있습니다.

이 책은 오라클 ZFS에 대해 테스트하지 않았습니다. 오라클은 비공개 소스인 ZFS를 독자적인 방향으로 발전시켜 왔으며, 오라클 솔라리스를 실행해야 하는 경우 오라클의 공식 문서를 읽는 것이 더 좋습니다.

디스크 기술에 대해 알아야 합니다. FreeBSD의 ZFS는 모든 GEOM 제공업체에서 실행할 수 있지만, 원시 디스크에서 실행하면 특정 이점이 있습니다. RAID 컨테이너에서 ZFS를 실행하면 이러한 이점이 사라집니다.

RAID 컨트롤러를 여러 개의 디스크처럼 작동하도록 설정할 수 있어야 합니다. 또한 ZFS를 플레이할 수 있는 테스트 머신이 있어야 합니다. 이 책을 구입하고 바로 메인 데이터베이스 서버를 ZFS로 마이그레이션하지 마세요! 테스트 머신에 ZFS를 설치한 다음 덜 중요한 머신에 설치하세요. 시스템 용도에 맞게 ZFS를 가장 잘 구성할 수 있다는 확신이 들 때까지 ZFS 기능을 실험해 보세요. ZFS는 빠르지만, 시스템 관리자가 성능을 저하시키도록 구성할 수 없을 정도로 내성이 강한 파일 시스템은 없습니다.

ZFS는 어디에서 사용하나요?

ZFS는 어디에서나 사용할 수 있지만 일부 환경에서는 제대로 작동하지 않을 수 있습니다.

특정 가상화 시스템에서는 ZFS가 최선의 선택이 아닐 수도 있습니다. 저희는 ZFS 파일 시스템에 막히는 Linux KVM 기반 가상화 시스템을 두 개 이상 사용해 보았으며, 다른 시스템에서도 비슷한 문제가 발생한다고 해도 놀라지 않을 것입니다. 호스트 간 마이그레이션이나 이미지 기반 백업에서 복원하는 것과 같은 기능은 문제가 될 수 있습니다. 가상화 시스템에서 ZFS를 사용하기 전에 가상화 시스템에서 충분히 테스트하는 것이 좋습니다. Lucas는 이러한 시스템에 ZFS를 배포했지만 다른 백업 및 마이그레이션 전략을 사용합니다.

ZFS는 최신 하드웨어용으로 작성되었습니다. 최소한 몇 기가바이트의 RAM이 있어야 합니다. 라즈베리 파이와 같은 임베디드 시스템은 UFS2와 같은 기존 파일시스템에 더 적합합니다.

특정 하드웨어의 특정 고강도 워크로드는 특히 실제 512바이트 블록이 있는 하드 드라이브가 있는 경우 ZFS 보다 UFS2에서 더 잘 수행됩니다. 데이터베이스의 성능을 최대한 끌어올리고 싶다면 UFS2와 ZFS 모두에서 성능을 테스트해 보세요.

ZFS 하드웨어

많은 사람들이 ZFS에 하이엔드 하드웨어를 추천합니다. 저희도 하이엔드 하드웨어를 좋아합니다. 멋지죠. 하지만 하드웨어의 한계를 이해한다면 일반 하드웨어에서도 ZFS가 잘 작동합니다. 인터넷에서 찾을 수 있는 대부분의 ZFS 문서에는 최신 ZFS에 적용되지 않거나 FreeBSD에 적용되지 않는 권장 사항이 포함되어 있습니다.

RAM

Sun의 문서에서 ZFS를 잘 사용하려면 ECC RAM이 필요하다고 말한 것은 놀라운 일이 아닙니다. 썬은 하이엔드 서버를 판매했기 때문입니다. 하지만 매트 아렌스에 따르면, "ECC가 없는 시스템의 ZFS는 ECC가 있는 시스템의 다른 파일 시스템보다 더 위험하지 않다"고 합니다. ZFS에 내장된 오류 수정 기능은 메모리로 인한 오류를 대부분 보정하지만 모든 오류를 보정하지는 않습니다.

물론 ECC RAM을 선호하는 일반적인 논거는 여전히 유효합니다. 비 ECC 메모리가 장착된 시스템에서는 메모리 손상이 발생할 수 있으며, 이러한 오류 중 일부가 디스크에 기록될 수 있습니다. 그러나 이러한 오류는 사용 중인 파일시스템에 관계없이 발생할 수 있으며, ZFS 체크섬을 통해 문제를 식별할 수 있습니다. 고가용성

서비스를 실행하는 경우, 일반적인 모든 이유로 ECC가 필요합니다. 하지만 ZFS 노트북이나 홈 무비 서버는 일반 RAM으로도 정상적으로 작동합니다.

RAID 컨트롤러

하드웨어 RAID 컨트롤러를 사용하지 마세요. 절대로. 하드웨어 RAID 장치 위에서 ZFS를 실행하면 RAID의 모든 단점과 ZFS의 모든 단점이 있는 구성이 만들어집니다. 디스크 컨트롤러에는 RAID가 아닌 HBA(호스트 버스 어댑터)를 사용하십시오.

모든 RAID는 소프트웨어 RAID입니다. 하드웨어 RAID 컨트롤러는 사용자 지정 운영 체제를 실행하여 RAID 작업을 수행하며, 이 과정에서 운영 체제에서 하드웨어를 가립니다. 이는 스토리지 관리를 위해 소비자 운영 체제를 신뢰할 수 없었던 상업용 컴퓨팅이 널리 보급되던 초창기에는 의미가 있었습니다. Windows 3.1에서 OS 수준의 소프트웨어 RAID를 실행하는 것에 대해 3초만 생각해 보면 하드웨어 RAID가 왜 그렇게 보편화되었는지 이해할 수 있을 것입니다.

운영 체제가 개선되었습니다. 하드웨어는 수십억 배 더 강력해졌습니다. 환경이 바뀌었습니다.

ZFS는 하드웨어에 직접 액세스할 수 있도록 설계되었습니다. 의도적으로 중요한 메타데이터를 여러 디스크에 저장합니다. 그리고 이러한 디스크에 오류가 있는지 감시하고 오류를 기반으로 결정을 내립니다. 하드웨어 RAID 장치는 운영 체제에서 이 모든 걱정스러운 세부 사항을 숨겨서 ZFS가 스스로 복구할 수 있는 기능을 제거합니다. 하드웨어 RAID는 경쟁 기능이 없습니다.

ZFS 어레이를 재구축하는 것이 RAID 어레이를 재구축하는 것보다 훨씬 빠르며, 이는 파일 시스템과 중복성을 통합한 ZFS의 기능 덕분입니다. 하드웨어 RAID로 중복성을 관리하면 이러한 속도를 잃게 됩니다.

그러나 많은 RAID 컨트롤러는 일종의 RAID가 없는 디스크를 사용할 수 없습니다. 컨트롤러에 "디스크만 여러 개", 즉 JBOD를 구성하더라도 이러한 컨트롤러는 실제로 각 드라이브를 단일 디스크 RAID-0으로 포맷합니다. 이렇게 하면 운영 체제에서 블록 크기 및 여러 디스크 오류와 같은 특정 정보가 가려집니다. 더 나쁜 것은 이러한 컨트롤러에서 사용되는 디스크는 이 브랜드의 컨트롤러에서만 읽을 수 있으며, 때로는 이 모델의 RAID 컨트롤러에서만 읽을 수 있다는 것입니다! ZFS에 이러한 RAID 컨트롤러를 사용하면 새 박스에 정확히 동일한 RAID 카드가 있는 경우가 아니라면 하드 드라이브를 다시 포맷하지 않고는 다른 컴퓨터로 옮길 수 없습니다. 이렇게 하면 ZFS의 모든 풀 이동성이 사라집니다.

일부 하드웨어 RAID 카드는 재플래시하여 JBOD 컨트롤러로 사용할 수 있습니다. 불량 플래시는 RAID 카드를 망가뜨릴 수 있습니다. 두 저자 모두 하드웨어 RAID를 다시 사용하지 않을 것이므로 이 위험을 감수합니다.

전생에 아주 나쁜 사람이었기 때문에 하드웨어 RAID를 사용할 수밖에 없다면 운영 체제에 단일 디스크를 제시하세요.

RAID 컨트롤러가 각 디스크를 RAID 0으로 포맷해야 한다고 고집하면 문제가 발생합니다. 컨트롤러에서 "다시 쓰기" 모드를 비활성화하세요. 그렇지 않으면 컨트롤러의 쓰기 캐시로 인해 파일 시스템이 손상될 수 있습니다. 복잡성 증가, 성능 저하, 위험 증가를 감수하세요. 피할 수 없는 장애가 발생했을 때 "내가 말했잖아"라고 말하며 백업에서 복원하는 수고로움을 보상받을 수 있도록 이 모든 것을 경영진에게 이메일로 문서화하세요.

호스트 버스 어댑터는 RAID 카드보다 비용뿐 아니라 시간 면에서도 훨씬 저렴합니다. 사용하세요.

SATA 대 SAS 대 SSD

. ZFS SAS , SATA , SSD

. SAS SATA . ZFS ZFS

ZFS . SAS SATA ZFS .

(DiskRedundancy)

. .

FreeBSD . , , , , .

(Physical Redundancy)

FreeBSD . , , FreeBSD Mastery: Advanced ZFS .

.

/

) GUID(Globally Unique ID) (GPT) . FreeBSD /dev/da239 , , ZFS .

FreeBSD .

설치 중에 미리 준비해 두면 장애가 발생한 디스크, 심지어 원격 시설에 있는 디스크도 바로 복구할 수 있습니다. 전 세계 여러 곳에서 매우 고밀도의 스토리지 어레이를 운영하는 Jude는 이 방식을 사용하여 하드 드라이브 유지보수로 인한 과부하를 방지하고 있습니다.

스토리지 어레이의 이름 지정 및 번호 지정 체계를 마련하세요. 대부분의 스토리지 어레이에는 표준 명명 체계가 있으며, 장비에 인쇄되어 있는 경우가 많습니다. 장비에 이미 번호가 매겨진 선반이 있다면 그 번호를 사용하세요. 그렇지 않은 경우에는 "선반 0은 항상 맨 위에 있고 디스크 0은 항상 왼쪽에 있다"와 같은 간단한 규칙을 만드세요. 앞쪽에는 접두사 'f'를, 뒤쪽에는 접두사 'b'를 사용하거나 자신에게 적합한 것을 사용할 수 있습니다.

어레이에 설치할 때 각 드라이브의 일련 번호를 기록해 두세요. 각 드라이브 트레이를 설치할 때 실제 위치와 일련 번호에 따라 물리적으로 라벨을 붙입니다. 예, 이 작업이 지루하긴 하지만 결국에는 이 정보가 필요할 것입니다. 이 작업은 자신의 속도에 맞춰 평화롭고 조용하게 할 수도 있고, 인위적으로 길어지고 불필요하게 스트레스를 주는 정전 중에 필사적으로 서둘러서 할 수도 있습니다.

이제 FreeBSD를 설치하거나 라이브 미디어를 부팅합니다. 캠컨트롤 devlist를 사용해 모든 저장장치 목록을 확인한 다음, 각 저장장치 노드에서 diskinfo -v를 실행해 일련번호를 확인합니다. (캠컨트롤(8)에서도 일련번호를 추출할 수 있습니다.) 이렇게 하면 /dev/da0 디스크가 실제로는 선반 4에 있는 디스크 3이라는 것을 알 수 있습니다.

이제 장치 노드 목록과 관련 일련 번호, 물리적 위치 및 일련 번호 목록이 생겼습니다. 이 목록들을 서로 일치시킨 다음 GPT 레이블을 사용하여 사용 중인 디스크 파티션에 위치와 일련 번호를 붙이세요. (GPT 레이블에 대한 자세한 내용은 FreeBSD 문서 또는 FreeBSD Mastery: Storage Essentials을 참조하십시오.) GPT 레이블의 최대 길이는 15자이므로, 긴 일련 번호는 잘라내야 할 수도 있습니다. 대부분의 일련번호에서 마지막 숫자가 가장 고유하므로 앞부분을 잘라내야 합니다.

예를 들어, 선반 2에 있는 디스크 9의 일련 번호가 WD-WCAW36477223인 경우 **/dev/gpt/s2d9-AW36477223**과 같은 레이블이 표시될 수 있습니다.

시스템에서 이 레이블을 사용하려면 이 레이블만 사용해야 합니다. 시스템에서 GPTID 및 디스크 식별 레이블을 비활성화합니다. 이렇게 하면 나중에 혼동을 피할 수 있습니다. 이렇게 설정하면 하드웨어 장애가 발생했을 때 FreeBSD가 4번 선반에 있는 세 번째 디스크(일련 번호 등등)가 불량임을 알려줄 수 있습니다. 이 정보만 있으면 코로케이션 제공업체의 말단 기술자라도 올바른 디스크를 가져올 수 있습니다.[1] 설치 전에 기술자에게 교체 드라이브의 일련 번호를 알려주면 적절한 레이블을 만들 수 있습니다.

사전 계획을 세우면 가동 중단으로 인한 충격이 훨씬 줄어듭니다. 이를 적극 권장합니다.

챕터 1: ZFS 소개

시작하셨습니다. ZFS를 사용해 본 적이 없다면, 잠시 시간을 내어 테스트 시스템이나 가상 머신에 ZFS가 포함된 새로운 FreeBSD를 설치해 보세요. 암호화나 화려한 커스터마이징 옵션은 선택하지 마세요. 이 간단한 설치는 더 복잡한 설정으로 들어가기 전에 몇 가지 ZFS 기본 사항을 살펴볼 수 있는 기회를 제공합니다.

ZFS는 기존 파일 시스템과 볼륨 관리자의 기능을 결합한 것입니다. 따라서 개별 파일에 대한 권한과 어떤 파일이 어떤 디렉터리에 있는지부터 어떤 저장 장치가 어떤 용도로 사용되고 그 저장 장치가 어떻게 배열되는지 추적하는 것까지 모든 것을 처리할 수 있습니다. 시스템 관리자는 디스크와 파일을 정렬하도록 ZFS에 지시하지만, 그 아래의 전체 스토리지 스택은 ZFS가 관리합니다. 이 장에서는 기본 ZFS 설정으로 설치된 FreeBSD 10.1 호스트를 사용하여 ZFS 스택을 파일 시스템, 스토리지 풀 및 가상 장치의 세 가지 계층으로 분리합니다.

방향을 잡기 위해 스토리지 스택에서 가장 눈에 잘 띄는 부분부터 시작해서 아래로 내려갑니다. 각 계층이 서로 어떻게 결합되어 있는지 이해했다면, 이 책의 나머지 부분은 기초부터 시작하여 위로 올라갑니다.

ZFS 데이터 세트 (ZFS Datasets)

ZFS 파일시스템은 기존 파일시스템과 정확히 유사하지 않으므로 데이터 세트라고 부릅니다. 고전적인 유닉스 파일 시스템(UFS)과 그 파생물, 그리고 최신 BSD의 UFS2와 Linux의 extfs와 같은 유사 파일 시스템은 다양한 프로그램으로 파일 시스템을 관리합니다. **df(1), newfs(8), mount(8), umount(8), dump(8), restore(8)** 및 이와 유사한 명령어를 사용하는 데 익숙하실 것입니다. ZFS는 이러한 모든 기능을 zfs(8) 프로그램에 통합하여 ZFS 데이터세트를 생성, 삭제, 조회, 스핀들링할 수 있게 해줍니다.

zfslist로 기존 ZFS 데이터세트를 보는 것으로 시작하세요.

```
$ zfs list
NAME              USED  AVAIL  REFER  MOUNTPOINT
zroot            429M  13.0G    96K  none
zroot/ROOT         428M  13.0G    96K  none
zroot/ROOT/default  428M  13.0G   428M  /
zroot/tmp          104K  13.0G   104K
/tmp zroot/usr          428K  13.0G    96K  /usr
…
```

이것은 mount(8)과 df(1)의 출력을 결합한 것으로, UFS나 extfs를 관리해 본 사람이라면 누구나 익숙하게 볼 수 있을 것입니다.

각 데이터세트에는 이름이 있습니다. ZFS 데이터 세트 이름은 데이터 세트가 있는 ZFS storage polol, 즉 zpool로 시작합니다. 첫 번째 항목은 그냥 평범한 zroot라고 합니다. 이 항목은 다른 모든 것이 매달려 있는 풀의 root dataset를 나타냅니다.

다음 두 열은 사용된 공간과 사용 가능한 공간의 양을 보여줍니다. 풀 루트에는 429MB와 13GB의 여유 공간이 있습니다.

REFER 열은 ZFS에만 있는 특별한 열입니다. 이것은 데이터 집합에서 액세스할 수 있는 데이터의 양이며, 사용된 공간의 양과 반드시 같지는 않습니다. 스냅샷과 같은 일부 ZFS 기능은 서로 간에 데이터를 공유합니다. **zroot** 항목은 429MB를 "사용"했지만 96KB의 데이터만 참조합니다. 풀 전체에는 13GB의 여유 공간이 있지만, 이 특정 데이터 집합을 통해 96KB에 액세스할 수 있습니다. 그다지 많지 않은 양입니다. 나머지 공간은 이 데이터 세트의 자식들을 위해 사용됩니다. 6장에서는 ZFS 디스크 사용에 대해 자세히 설명합니다. 데이터 세트의 자식에는 이 책 전체에서 볼 수 있듯이 스냅샷, 볼륨 및 자식 데이터 세트가 포함됩니다.

마지막으로 파일 시스템 마운트 지점이 있습니다. **zroot** ZFS는 마운트되지 않았습니다.

두 번째 항목인 **zroot/ROOT**를 살펴보세요. 이것은 루트 파일시스템용으로 생성된 ZFS 데이터 세트입니다. **zroot** 풀과 마찬가지로 마운트되지 않습니다. 96KB의 데이터를 참조합니다. 이것은 루트 파일 시스템에서는 이상하게도 사용되지 않는 것으로 보입니다.

세 번째 항목인 **zroot/ROOT/default**는 현재 루트 파일시스템입니다. 이 파일은 428MB의 데이터를 사용하며, 유닉스 루트인 /에 마운트됩니다. 428MB라는 의미는 이 데이터 세트에 그 정도의 데이터가 있다는 뜻입니다.

ZFS가 이 데이터를 루트 파일 시스템에서 분리하는 이유는 무엇일까요? ZFS를 사용하면 여러 루트 파일시스템 중에서 쉽게 선택할 수 있습니다. 이 호스트는 FreeBSD 10.1을 실행하는데 일부 보안 업데이트를 적용하고 재부팅해야 한다고 가정해 보겠습니다. 운영 체제 패치를 적용할 때면 시스템 관리자는 항상 두려움과 희망이 뒤섞인 고민에 빠지게 됩니다. 잘 테스트된 업그레이드도 잘못되면 모든 사람의 하루를 망칠 수 있습니다. 하지만 ZFS를 사용하면 데이터 세트를 복제하고 스냅샷할 수 있습니다. FreeBSD 10.1-p1로 업그레이드할 때, **zroot/ROOT/10.1-p1**과 같은 새로운 데이터 세트를 생성하고 이를 루트 파티션으로 사용하도록 FreeBSD에 지시할 수 있습니다. **zroot/ROOT/default**를 마운트하지 않거나 **/oldroot**와 같은 다른 위치에 마운트합니다. 업그레이드가 제대로 진행되지 않으면 되돌리기가 쉽지 않습니다.

다음 데이터 세트인 **zroot/tmp**는 거의 비어 있습니다. 이 데이터셋은 **/tmp**에 마운트됩니다. 이 데이터 세트는 전통적인 임시 디렉터리입니다.

ZFS 파티션 및 속성 (ZFS partitions and properties)

ZFS에는 전통적인 파티션이 없습니다. 파티션은 디스크의 논리적 세분화로, 스토리지 장치에서 매우 특정한 LBA(논리 블록 주소)를 채웁니다. 파티션은 파티션에 있는 데이터를 인식하지 못합니다. 파티션을 변경한다는 것은 그 위에 있는 파일 시스템을 파괴하고 (아마도) 다시 구축하는 것을 의미합니다.

파티션이 없는 파일시스템을 처음 본 루카스는 스토리지를 어떻게 관리할지 궁금해졌습니다. 이는 추운 미시간의 긴 겨울이 지나고 몇 달 만에 처음으로 바깥에 나와 따뜻한 자연 공기를 느꼈을 때 느끼는 혼란과 비슷합니다. 혼란은 해방의 일부입니다. 파티션을 통해 스토리지를 관리하는 방법을 배웠던 것은 파티션이 좋거나 최고의 솔루션이기 때문이 아니라 해야만 했기 때문입니다. 파티션 없이 기존 파일 시스템을 실행하는 것은 좋지 않은 습관이지만, ZFS는 기존 파일 시스템이 아닙니다.

ZFS는 파일시스템과 하위 스토리지 계층을 긴밀하게 통합합니다. 즉, 필요에 따라 다양한 파일시스템 간에 스토리지 공간을 동적으로 분할할 수 있습니다. ZFS 파일시스템에 특정 크기 제한을 설정할 수 있지만, 데이터

세트에는 전통적인 크기가 없습니다. 풀에 파일을 위한 충분한 공간이 있는 경우 이를 사용할 수 있습니다. 이전에 **/var/log**에 제한된 양의 디스크 공간을 할당하여 광란의 로그가 디스크를 가득 채우는 것을 막았다면, 이제 ZFS 수준에서 이러한 제한을 설정해야 합니다.

데이터 세트가 사용할 수 있는 공간의 양은 ZFS 속성의 한 예입니다. 예를 들어, **quota** 속성은 데이터 세트가 얼마나 커질 수 있는지를 제어하는 등 ZFS는 수십 개의 데이터 세트 속성을 지원합니다. ZFS 속성을 설정하려면 **zfs(8)**을 사용합니다.

```
$ zfs set quota=2G zroot/var/log
```

zfs get 명령으로 속성을 조회합니다.

```
$ zfs get quota zroot/var/log
NAME          PROPERTY  VALUE  SOURCE
zroot/var/log  quota      2G  local
```

zfs get all로 ZFS 데이터셋 이름과 데이터셋의 모든 속성을 볼 수 있습니다.
4장에서는 ZFS 속성을 자세히 살펴보고, 6장에서는 데이터 집합 크기 제한에 대해 설명합니다.

ZFS 제한 (ZFS Limits)

파일시스템에는 항상 최대 크기와 제한이 있었습니다. 우리가 모두 알고 있는 FAT 파일시스템은 최대 크기인 32MB를 극복하기 위해 여러 차례의 수정이 필요했고, 그 후 2GB, 4GB로 늘어났습니다. 요즘은 FAT32의 2TB 제한이 조금 비좁아 보이기 시작했습니다. UFS와 ext2/3/4fs도 비슷하게 임의적인 제한을 두고 있습니다. 이러한 제한은 파일시스템 개발자가 어딘가에 한도를 설정해야 했고, 향후 몇 년 동안은 괜찮을 것으로 예상되는 값을 선택했기 때문에 존재합니다. 그러나 인기 있는 파일시스템은 이러한 한계에 도달할 때까지 계속 사용되기 때문에 시스템 관리자는 이러한 한계에 반복적으로 대처해야 했습니다.

ZFS 옹호자들은 ZFS가 이러한 임의의 한계에 영향을 받지 않는다고 주장하지만, 이는 사실이 아닙니다. ZFS는 대부분의 값을 저장하는 데 128비트를 사용하기 때문에 오늘날 시스템 관리 업무를 하는 사람이라면 한계를 넘을 수 없을 정도로 높은 한계를 설정합니다. 하나의 디렉터리에는 각각 최대 16엑사바이트의 2^{48}개 파일을 저장할 수 있습니다. 단일 풀은 최대 256제타바이트, 즉 2^{78}바이트까지 가능합니다. 스토리지 풀에는 최대 2^{64}개의 장치가 포함될 수 있으며, 단일 호스트에는 최대 2^{64}개의 스토리지 풀이 있을 수 있습니다.

좋은 소식은 우리가 이러한 한계에 도달할 만큼 오래 살지는 않을 것이라는 점입니다. 나쁜 소식은 파일 시스템 간 마이그레이션에 대한 모든 전문 지식을 보유하고 있다는 것입니다. 기술이 ZFS의 한계에 부딪히면 파일시스템 간 마이그레이션에 익숙하지 않은 사람들은 어려움을 겪을 것입니다. 다행히도 연습을 위해 FAT/UFS/extfs 롤오버가 몇 가지 남아있을 것입니다.

스토리지 풀

ZFS는 디스크가 아닌 스토리지 풀을 사용합니다. 스토리지 풀은 기본 스토리지 공급자의 상단에 있는 추상화로, 물리적 매체와 그 위에 있는 사용자가 볼 수 있는 파일 시스템을 분리할 수 있게 해줍니다. **zpool(8)**을 사용해 시스템의 스토리지 풀을 보고 관리할 수 있습니다. 다음은 기본 FreeBSD 시스템의 풀입니다.

```
$ zpool status
  pool: zroot
state: ONLINE
  scan: none requested
config:

NAME        STATE   READ WRITE CKSUM
zroot       ONLINE      0     0     0
 gpt/zfs0 ONLINE      0     0     0

errors: No known data errors
```

먼저 풀의 이름과 상태가 표시됩니다 . 시스템에는 둘 이상의 ZFS 풀이 있을 수 있으며 , 수십 개의 하드 드라이브가 있는 대규모 시스템에는 여러 개의 풀이 있는 경우가 많습니다 . 이 호스트에 여러 개의 스토리지 풀이 있는 경우 위의 샘플과 같이 각 풀이 별도의 설명으로 표시됩니다 .

ZFS는 스토리지 풀에 대해 다양한 종류의 무결성 검사를 수행할 수 있습니다 . **scan** 구문은 무결성 검사가 수행되고 있는지 여부와 가장 최근 검사 결과를 보여줍니다 .

풀 목록의 마지막 부분에는 풀에 있는 가상 장치의 레이아웃이 표시됩니다 .

가상 장치 (Virtual Devices)

스토리지 풀에는 하나 이상의 가상 장치 , 즉 VDEV가 포함되어 있습니다 . VDEV는 기존 RAID 장치와 유사합니다 . 시스템 관리자가 실제로는 여러 개의 작은 디스크라는 것을 알고 있지만 파일 시스템 계층에는 하나의 큰 장치로 표시됩니다 . 가상 장치를 사용하면 특정 장치를 특정 역할에 할당할 수 있습니다 . VDEV를 사용하면 필요에 따라 물리적 스토리지를 배치할 수 있습니다 .

가상 디바이스는 ZFS의 모든 마법이 일어나는 곳입니다 . RAID 방식의 중복성을 위해 풀을 배열할 수 있습니다 . 공급자를 전용 읽기 및 쓰기 캐시로 사용하여 가상 장치의 성능을 향상시킬 수 있습니다 . 2장에서는 가상 장치에 대해 더 자세히 다룹니다 .

ZFS의 데이터 중복성 및 자동 오류 수정도 VDEV 수준에서 이루어집니다 . ZFS의 모든 데이터는 무결성 검증을 위해 체크섬 처리됩니다 . 풀에 충분한 중복성이 있는 경우 , ZFS는 자가 복구 기능을 제공합니다 . 풀에 중복성이 부족하다면 적어도 데이터가 손상되었다는 것을 알 수 있고 백업에서 복원할 수 있습니다 .

> ZFS는 백업의 필요성을 없애지 않습니다 . 백업을 없애는 유일한 방법은 절대적인 무관심뿐입니다 .

풀의 상태를 표시하는 **zpool status** 명령은 해당 풀의 가상 장치도 표시합니다 . 이전 섹션의 예시를 보세요 . 이 매우 간단한 풀인 zroot에는 단일 스토리지 공급자인 **/dev/gpt/zfs0**이 포함되어 있습니다 . 이 공급자는 디스크가 아닌 GPT 파티션입니다 . 2장에서 설명한 대로 ZFS는 모든 종류의 기본 스토리지를 사용할 수 있습니다 . GPT 파티션을 사용하는 것이 매우 일반적이지만 , 다른 옵션으로는 전체 디스크 , 파일 및 기타 GEOM 공급자가 있습니다 . FreeBSD는 암호화와 같은 기능을 지원하기 위해 GEOM 공급자를 사용합니다 .

블록과 이노드 (Blocks and Inodes)

전통적인 파일시스템은 거의 항상 데이터를 저장하기 위해 다양한 데이터 블록을 사용하고 이러한 블록의 내용을 인덱스 노드로 매핑합니다. BSD의 UFS와 Linux의 extfs는 이러한 블록과 이노드를 호출합니다. Microsoft의 FAT 파일시스템에도 데이터 저장 블록과 인덱스 노드가 있습니다.

이러한 파일시스템과 마찬가지로 ZFS는 인덱스 블록과 데이터 블록을 사용합니다. 그러나 이전 파일시스템과 달리 ZFS는 필요에 따라 인덱스 노드를 생성합니다. ZFS는 가능할 때마다 데이터에 맞는 크기로 스토리지 블록을 생성합니다. 가변 크기 블록이 생성 가능한 모든 파일에 항상 맞는 것은 아니지만, 기존 파일시스템보다 확실히 더 유연합니다.

UFS 슈퍼블록과 달리 동적으로 생성된 인덱스 블록은 디스크의 알려진 위치에 배치할 수 없습니다. ZFS는 인덱스 블록이 손상될 가능성에 어떻게 대처할 수 있나요? ZFS는 중요한 인덱스 블록의 복사본을 알고리즘적으로 예측 가능한 위치에 여러 개 저장합니다. 이러한 블록은 디스크의 여러 위치에 복제됩니다. 3장에서는 ZFS 블록, 우버블록, 동일 블록, 트랜잭션 그룹 등에 대해 설명합니다.

이제 ZFS의 기본을 알았으니, 이 책의 나머지 부분에서는 수백 가지의 세부 사항을 설명합니다. 스택의 가장 아래쪽인 가상 장치부터 시작하겠습니다.

챕터 2 : 가상 장치 (Virtual Devices)

이 장에서는 소시지가 어떻게 만들어지는지 살펴보겠습니다 . 이것은 ... 돼지 , 즉 디스크입니다 . 디스크는 스토리지의 물리적 표현입니다 . 디스크는 사악합니다 . 디스크는 그 특성과 레이아웃에 대해 거짓말을 하고 , 오류를 숨기며 , 예상치 못한 방식으로 실패합니다 . ZFS를 사용하면 더 이상 디스크가 몰래 음모를 꾸미는 것은 아닌지 걱정할 필요가 없습니다 . 예 , 디스크가 음모를 꾸미고 있기는 하지만 , ZFS는 디스크의 배신을 폭로하고 이를 중단시킵니다 .

ZFS에서 사용 가능한 디스크를 가장 효과적으로 사용하려면 운영 체제가 ZFS에 디스크를 제공하는 방식과 ZFS가 해당 디스크의 데이터를 정렬하는 방식에 대한 기본적인 이해가 필요합니다 .

디스크 및 기타 저장 매체 (Disks and Other Storage Media)

ZFS는 디스크가 아닌 다른 저장 매체에서도 실행할 수 있습니다 . FreeBSD GEOM 저장소 제공자라면 무엇이든 ZFS 저장 매체가 될 수 있습니다 . ZFS는 파일을 백업 스토리지로 사용하는 기능도 지원하는데 , 이는 테스트용으로는 정말 좋지만 프로덕션용으로는 적합하지 않습니다 . ZFS는 모든 블록 장치를 물리적 저장소로 사용할 수 있지만 각 유형마다 장단점이 있습니다 .

원시 디스크 스토리지 (Raw Disk Storage)

전체 물리적 디스크를 사용하면 복잡성이 줄어듭니다 . 또한 , 파티셔닝에 대해 걱정할 필요가 없으며 , ZFS와 물리적 디스크 사이에 소프트웨어나 구성이 필요하지 않습니다 . 그러나 일반적으로 이러한 장점보다 단점이 더 큽니다 .

디스크에서 부팅하려면 디스크에 부트 로더가 있어야 합니다 . 부트 로더는 파티션된 디스크에서만 사용할 수 있습니다 . 원시 디스크는 부팅할 수 없습니다 . FreeBSD는 디스크에 유용한 레이블을 지정할 수 있도록 지원하지만 , 이러한 레이블은 파티션 정보 안에 있습니다 .

더 나쁜 점은 , 대체 디스크는 원본 디스크와 정확히 같은 크기이거나 더 커야 한다는 것입니다 . 6TB 디스크의 크기가 모두 같은 것은 아니며 , 공급업체마다 디스크 크기가 몇 메가바이트씩 다릅니다 . 시스템을 설정할 때는 이러한 차이가 중요하지 않지만 디스크를 교체할 때는 매우 중요합니다 . 대부분의 카탈로그에는 각 디스크의 섹터 수가 나열되어 있지 않고 크기만 나와 있기 때문에 사용 가능한 대체품을 찾으려면 여러 번 시도해야 할 수 있습니다 . 기존 512바이트 섹터를 사용하는 드라이브를 4096바이트 (4K, 고급 포맷이라고도 함) 섹터를 사용하는 드라이브로 교체하면 상황이 더욱 복잡해집니다 . 원래 드라이브에는 8로 균등하게 나눌 수 없는 수의 섹터가 있었을 가능성이 높습니다 . 디스크 드라이브에서 사용하는 특수 수학 덕분에 새 드라이브는 이전 드라이브보다 몇 바이트 더 크더라도 몇 바이트 더 작아 보일 수 있습니다 .

파티션 스토리지

전체 원시 디스크를 사용하는 대신 디스크를 파티션한 다음 그 파티션 중 하나를 ZFS용으로 사용할 수 있습니다. 이 방법의 가장 큰 장점은 별도의 부팅 장치가 필요 없이 작은 부팅 파티션을 만들어 ZFS 파티션이 포함된 디스크에서 부팅할 수 있다는 것입니다. 또한 파티션을 사용하면 원시 스왑 파티션이나 다른 파일 시스템과 같은 다른 용도로 디스크 공간의 일부를 사용할 수 있으며, 교체 디스크의 섹터 수가 일치하지 않아도 되도록 디스크 끝에 약간의 여유 공간을 남겨둘 수 있습니다. 또한 파티셔닝을 통해 드라이브를 '짧게' 만들어 성능을 향상시킬 수도 있습니다.

많은 원본 Solaris ZFS 관리 가이드에서는 성능상의 이유로 파티션(또는 Solaris 용어로는 슬라이스)을 사용하지 말 것을 권장합니다. Solaris에서 파일 시스템에 파티션을 사용하면 쓰기 캐시가 비활성화됩니다. FreeBSD에서 쓰기 캐시를 비활성화하는 것은 디스크 파티셔닝이나 파일 시스템과는 완전히 별개입니다. FreeBSD는 파티션에서 ZFS를 사용할 때 완전한 성능을 제공합니다.

FreeBSD는 여러 가지 파티셔닝 체계를 지원하지만, GPT를 강력히 권장합니다. 이전 파티셔닝 시스템인 MBR은 기본 파티션의 수를 4개로 제한한 반면, GPT는 최대 128개의 파티션을 지원합니다. MBR은 최대 2TB까지만 디스크를 관리할 수 있는 반면, GPT는 512바이트 섹터 디스크로 최대 8ZB, 4K-섹터 디스크로 최대 64ZB까지 관리할 수 있습니다. FreeBSD Mastery: Storage Essentials에서는 두 가지 파티셔닝 방법에 대한 FreeBSD의 지원을 다룹니다.

> 클레이 태블릿에 데이터를 저장하는 경우, bsdlabe(8) 파티션을 사용할 수 있습니다.

파티션 사용의 단점은 ZFS가 제공하는 일부 이식성을 잃을 수 있다는 것입니다. 한 시스템에서 다른 시스템으로 디스크를 이동하는 경우, 대상 시스템이 디스크 파티션을 인식할 수 있어야 합니다.

GEOM 장치 스토리지 (GEOM Device Storage)

ZFS는 다양한 FreeBSD GEOM 클래스를 백업 스토리지로 사용할 수도 있습니다. 이러한 클래스는 파일 시스템과 물리적 장치 사이에 위치하며 다양한 기능을 수행합니다. GEOM 클래스는 전체 디스크 암호화(GELI, GBDE), 고가용성, 레이블, 다중 경로 및 플러그형 스케줄러와 같은 기능을 제공합니다. GEOM 클래스는 전체 장치를 기반으로 만들거나 파티션, 다중 경로 장치 또는 암호화된 디스크와 같은 다른 GEOM 클래스 위에 만들 수 있습니다.

암호화된 ZFS 풀을 생성하는 가장 좋은 방법은 GELI(FreeBSD Disk Encryption Subsystem)입니다. GELI는 ZFS와 물리적 디스크 사이를 오갈 때 블록을 암호화하고 해독하므로 ZFS에서 다른 작업을 할 필요가 없습니다. GELI는 다양한 암호화 알고리즘을 지원하지만, 기본값인 AES-XTS는 특히 AES-NI(AES New Instructions)를 지원하는 최신 CPU에서 최상의 성능을 제공합니다. 이 하드웨어 오프로드 기능의 도움으로 GELI는 초당 1GB 이상의 속도로 데이터를 암호화하고 더 빠르게 해독할 수 있으므로, SSD에서도 암호화를 추가해도 성능이 저하되지 않습니다. 또한 GELI는 각 섹터에 해시 메시지 인증 코드(Heshed Message Authentification Code, HMAC)를 저장하는 데이터 인증(무결성 검증) 기능을 선택적으로 제공할 수 있습니다. 이 HMAC를 사용하여 데이터의 무결성(데이터가 변조되지 않았는지)과 진위성(이 데이터가 사용자가 작성한 것인지)을 확인합니다. 섹터를 다시 읽을 때 HMAC가 데이터를 확인하지 못하면 오류가 반환됩니다. HMAC 기능은 기본적으로 활성화되어 있지 않으며, ZFS는 각 데이터 블록에 대해 자체 체크섬을 제공하기 때문에 ZFS에 비해 과한 기능일 수 있습니다.

고가용성 스토리지 기술(High Availability Storage Technology, HAST)은 FreeBSD의 분산 스토리지 솔루션입니다. 네트워크를 통해 컴퓨터 간에 블록 장치를 미러링할 수 있습니다. HAST를 ZFS 풀의 백업

스토리지로 사용하면 각 백업 디스크를 두 번째 머신에 미러링할 수 있습니다. HAST의 장점은 실시간이라는 점입니다. 블록은 HAST 클러스터의 모든 호스트에 쓰여질 때까지 쓰여진 것으로 간주되지 않습니다. 반면에 ZFS 복제는 주기적인 스냅샷 동기화를 기반으로 합니다. 그러나 HAST를 사용하면 두 번째 머신은 첫 번째 머신과 동시에 풀을 가져오거나 마운트할 수 없습니다. 복제된 풀을 동시에 활성화(읽기 전용)할 수 있는 ZFS 복제와 비교했을 때, HAST는 몇 가지 경우에만 의미가 있습니다.

GEOM 레이블은 각 디스크나 파티션에 의미 있는 메모를 첨부할 수 있는 편리한 방법을 제공합니다. disk ident, gptid, GPT 레이블과 같은 표준과 GEOM-specific glabel을 포함한 다양한 레이블 유형이 있습니다. 드라이브 라벨링 모범 사례는 0장에 나와 있습니다.

GEOM은 고가용성을 위한 다중 경로도 지원합니다. 디스크뿐만 아니라 컨트롤러 카드, 백플레인 또는 케이블도 고장이 나는 경우가 있습니다. 다중 경로를 사용하면 '듀얼 포트'인 엔터프라이즈 드라이브를 둘 이상의 HBA(RAID 기능이 없는 디스크 컨트롤러 카드)에 연결할 수 있습니다. 각 드라이브에 두 개의 서로 다른 스토리지 컨트롤러에 대한 경로가 있는 경우, 해당 컨트롤러 중 하나가 손실되더라도 드라이브는 살아남을 수 있습니다. 그러나 각 디스크가 서로 다른 두 개의 컨트롤러에 연결되어 있으면 운영 체제는 각 디스크를 각 컨트롤러를 통해 한 번씩 두 번 인식합니다. GEOM 다중 경로 클래스를 사용하면 각 디스크에 레이블을 작성하여 동일한 디스크에 대한 연속적인 경로가 그렇게 감지되도록 할 수 있습니다. 이렇게 하면 각 디스크에 대한 하나의 표현을 얻을 수 있으며, 다른 컨트롤러를 통해 해당 디스크에 대한 여러 경로로 뒷받침됩니다. 다중 경로에 대해서는 FreeBSD Mastery: Advanced ZFS에서 다중 경로에 대해 설명합니다.

관리자는 GEOM 스케줄러 모듈을 사용하여 더 나은 성능을 달성하기 위해 다양한 I/O 스케줄링 알고리즘을 지정할 수 있습니다. 이 글을 쓰는 현재, 현재 사용 가능한 스케줄러는 하나의 큐만 사용하는 간단한 형태의 예상 스케줄링인 "as"와 각 클라이언트 큐에 걸쳐 라운드 로빈 서비스를 사용하는 예상 스케줄링인 "rr"입니다. 자세한 내용은 **gsched(8)**을 참조하세요. GEOM 시스템을 사용하면 특정 워크로드에 대한 추가 스케줄링 모듈을 비교적 쉽게 작성할 수 있습니다.

파일 백업 스토리지 (File-Backed Storage)

파일 백업 가상 디스크를 ZFS 저장 장치로 사용할 수 있습니다. 프로덕션 환경에서는 권장하지 않지만, 파일 백업 디스크는 테스트 및 실험용으로 유용할 수 있습니다.

프로바이더와 디스크 (Providers vs. Disks)

"프로바이더"는 FreeBSD의 기술 용어입니다. GEOM 스토리지 제공자는 데이터 스토리지를 제공하는 것을 말합니다. 디스크일 수도 있습니다. 어떤 식으로든 저장소를 변환하는 GEOM 클래스일 수도 있습니다. 기술적으로 말하자면, 이 책에서는 거의 모든 곳에서 디스크 대신 프로바이더라는 단어를 사용해야 합니다. 모든 GEOM 공급자를 ZFS의 백엔드로 사용할 수 있습니다. 문제는 하나의 물리적 디스크가 여러 개의 다른 공급자를 제공할 수 있다는 것입니다. 풀에 여러 개의 서로 다른 공급자가 있을 수 있지만, 이 모든 공급자가 하나의 디스크에 있다면 중복성이 머릿속을 가득 채운 것입니다.

> FreeBSD의 유연한 스토리지 시스템은 멍청한 짓을 할 수 있는 힘을 제공합니다. 하지 마세요.

이 책에서 "디스크"에 대해 설명하는 것은 "디스크 위에 있는 일종의 공급자"를 의미합니다. 디스크에 스왑 파티션과 ZFS 파티션이 있어도 문제없지만, 이 디스크가 전적으로 ZFS 전용일 필요는 없습니다. 하지만 하나의 물리적 디스크에 두 개의 ZFS 파티션을 만들어 미러링하고 물리적 중복성을 확보할 수는 없습니다.

VDEV: 가상 장치 (VDEVs: Virtual Devices)

가상 장치 또는 VDEV는 ZFS의 논리적 저장 장치입니다. 각 VDEV는 하나 이상의 GEOM 공급자로 구성됩니다. ZFS는 여러 가지 유형의 VDEV를 지원하며, 이는 VDEV가 제공하는 중복성 유형에 따라 구분됩니다. 각 디스크에 다른 디스크의 복사본이 들어 있는 일반적인 미러 디스크는 VDEV의 한 유형입니다. 중복성이 없는 플레인 디스크는 또 다른 유형의 VDEV입니다. 그리고 ZFS에는 RAID-Z라고 하는 세 가지 종류의 정교한 RAID가 있습니다.

이러한 VDEV는 3장에서 설명한 스토리지 풀로 배열됩니다. 실제 데이터는 4장에서 다루는 것처럼 풀 위에 저장됩니다. 하지만 가상 디바이스의 배열에 따라 풀의 성능과 물리적 손상에 대한 저항력이 결정됩니다. ZFS의 거의 모든 중복성은 가상 장치에서 비롯됩니다.

스토리지 풀은 하나 이상의 VDEV로 구성되며, 풀 데이터는 중복성 없이 해당 VDEV에 분산됩니다. (4장에서 설명한 대로 복사 속성을 사용하여 중복성을 추가할 수 있지만 전체 디스크 장애에 대한 보호는 제공하지 않습니다.) ZFS 풀은 VDEV를 스토리지 공간을 제공하는 단일 단위로 취급합니다. 스토리지 풀은 VDEV가 손실되더라도 살아남을 수 없으므로, 중복성이 있는 VDEV를 사용하거나 이 풀의 데이터를 손실해도 괜찮다고 미리 결정하는 것이 중요합니다.

한 풀에 여러 개의 VDEV를 사용하면 고급 RAID 어레이와 유사한 시스템이 만들어집니다. RAID-Z2 어레이는 RAID-6과 유사하지만, 두 개의 RAID-Z2 VDEV가 있는 ZFS 풀은 RAID-60과 유사합니다. 미러링된 VDEV는 RAID-1처럼 보이지만, 그룹은 RAID-10과 유사합니다. 이 두 경우 모두 ZFS는 중복성 없이 각 VDEV에 걸쳐 데이터를 스트라이핑합니다. 개별 VDEV가 중복성을 제공합니다.

VDEV 중복성 (VDEVs Redundancy)

둘 이상의 디스크가 포함된 VDEV는 여러 가지 중복성 체계를 사용하여 내결함성을 제공할 수 있습니다. 단일 디스크를 단독으로 이중화할 수 있는 방법은 없습니다. ZFS는 미러 디스크와 여러 패리티 기반 어레이 사용을 지원합니다.

ZFS는 중복성을 사용하여 자가 복구합니다. 중복성이 없는 VDEV는 자가 복구를 지원하지 않습니다. 데이터 세트 레이어에서 이 문제를 해결할 수 있지만 (**copies** 속성을 사용하여), 이중화 VDEV는 자동으로 자가 복구를 지원합니다.

Stripe (1개 공급자)

단일 디스크로 구성된 VDEV를 스트라이프라고 하며, 중복성이 없습니다. 예상할 수 있듯이 단일 공급자가 손실되면 디스크의 모든 데이터가 사라집니다. 스트라이프 풀에는 단일 디스크 VDEV만 포함됩니다.

ZFS 풀은 풀의 모든 VDEV에 걸쳐 데이터를 스트라이핑하고 중복성을 제공하기 위해 VDEV에 의존합니다. 하나의 스트라이프 디바이스에 장애가 발생하면 전체 풀이 장애가 발생합니다. 풀에 저장된 모든 데이터가 사라집니다. 스크래치 파티션의 경우 이 방식은 괜찮지만 데이터가 중요하다면 내결함성을 제공하는 VDEV 유형을 사용하세요.

Mirrors (2개 이상의 공급자)

미러 VDEV는 모든 디스크에 모든 데이터의 전체 사본을 저장합니다. 공급자의 드라이브 중 하나를 제외한 모든 드라이브를 잃어버려도 데이터에 액세스할 수 있습니다. 미러에는 원하는 수의 디스크를 사용할 수

있습니다.

미러는 모든 디스크에서 데이터를 한 번에 읽을 수 있기 때문에 랜덤 및 순차 읽기 속도가 매우 우수합니다. 모든 데이터를 모든 디스크에 써야 하고 가장 느린 디스크가 끝날 때까지 작업이 완료되지 않기 때문에 쓰기 성능이 저하됩니다.

RAID-Z1(3개 이상의 공급자)

ZFS에는 RAID-Z라고 하는 3개의 최신 RAID 스타일 중복 VDEV가 포함되어 있습니다. RAID-Z는 RAID-5와 비슷하지만 파일 무결성을 보장하기 위해 체크섬을 포함합니다. 체크섬과 ZFS의 쓰기 시 복사 기능 (7장) 사이에서 RAID-Z는 불완전한 쓰기로 인해 파일 시스템이 일관되지 않게 되지 않도록 보장합니다.

RAID-Z는 데이터 및 패리티 정보를 모든 디스크에 분산시킵니다. RAID-Z의 공급자가 죽거나 손상된 데이터를 제공하기 시작하면 RAID-Z는 패리티 정보를 사용하여 누락된 데이터를 다시 계산합니다. RAID-Z가 하나의 공급자를 사용하여 패리티 정보를 저장한다고 생각할 수 있지만, 단일 패리티 공급자가 아니라 여러 공급자가 패리티 역할을 순환하며 데이터를 분산합니다.

RAID-Z1 VDEV는 단일 스토리지 공급업체의 장애를 견딜 수 있습니다. 첫 번째 장애가 발생한 드라이브가 교체되기 전에 두 번째 공급자가 장애를 일으키면 모든 데이터가 손실됩니다. 패리티 데이터로 디스크 어레이를 재구축하는 데는 시간이 오래 걸릴 수 있습니다. 2TB 이상의 대용량 디스크를 사용하는 경우, 첫 번째 드라이브를 복구하는 동안 두 번째 드라이브에 장애가 발생할 가능성이 적지 않습니다. 더 큰 디스크의 경우 RAID-Z2를 고려해야 합니다.

RAID-Z2(4개 이상의 공급자

RAID-Z2는 RAID-Z1과 유사하지만, VDEV당 2개의 패리티 디스크가 있습니다. RAID-6과 마찬가지로 RAID-Z2는 두 개의 공급자가 실패하더라도 계속 작동할 수 있습니다. RAID-Z1보다 약간 느리지만 드라이브를 교체하는 데 다소 느긋할 수 있습니다.

RAID-Z3(5개 이상의 공급자)

RAID-Z의 가장 편집증적인 형태인 RAID-Z3는 VDEV당 3개의 패리티 디스크를 사용합니다. 즉, 5개의 디스크 어레이에 3개의 장애 디스크를 사용할 수 있습니다. RAID-Z2보다 약간 느립니다. 네 번째 디스크에 장애가 발생하면 전체 데이터가 손실됩니다.

RAID-Z 디스크 구성 (RAID-Z Disk Configurations)

모든 버전의 RAID-Z를 사용할 때 기억해야 할 중요한 사항 중 하나는 RAID-Z의 공급자 수가 완전히 고정되어 있다는 것입니다. RAID-Z VDEV에 드라이브를 추가하여 확장할 수 없습니다. VDEV를 추가하여 풀을 확장할 수는 있지만 디스크를 추가하여 VDEV를 확장할 수는 없습니다. 이 기능을 추가할 계획이 없습니다.

20개의 하드 드라이브를 수용할 수 있는 호스트가 있다고 가정해 보겠습니다. 나중에 필요에 따라 드라이브를 풀에 추가할 수 있다고 생각하고 12개의 드라이브를 설치하여 단일 RAID-Z2로 사용합니다. 이러한 새 드라이브는 별도의 RAID-Z2 VDEV로 연결해야 합니다. 게다가 VDEV의 밸런스가 맞지 않게 됩니다. 풀에는 단일 12-드라이브 VDEV와 두 번째 8-드라이브 VDEV가 있습니다. 하나는 다른 하나보다 느릴 것입니다. ZFS를 사용하면 이러한 장치를 강제로 함께 풀링할 수 있지만, 그렇게 하는 것은 정말 나쁜 생각입니다.

미리 계획하세요. 물리적 장비, 시작해야 하는 드라이브 수, 스토리지를 확장하는 방법을 고려하세요. 예제 서버는 하나의 RAID-Z2 VDEV를 포함하는 온풀과 나머지 8개의 디스크를 포함하는 완전히 분리된 풀을 원하는 배열로 구성해도 괜찮습니다. 시작도 하기 전에 스스로 목을 자르지 마세요!

RAID-Z 2의 규칙 (The RAID-Z Rule of 2s)

일반적으로 논의되는 구성 중 하나는 2의 배수만큼의 데이터 디스크 수와 주어진 RAID-Z 레벨에 필요한 패리티 디스크를 더하는 것입니다. 즉, 이 규칙에 따르면 RAID-Z1은 2n+1 디스크 또는 3, 5, 7, 9 등의 디스크를 사용해야 합니다. RAID-Z2는 2n+2 디스크(4개, 6개, 8개 등)를 사용해야 하고, RAID-Z3는 2n+3(5개, 7개, 9개 등)을 사용해야 합니다. 이 규칙은 데이터가 2의 거듭제곱인 크기의 작은 블록으로 구성된 경우에만 유효합니다. 하지만 다른 요인들이 훨씬 더 큰 차이를 만듭니다. 일반적으로 압축이 훨씬 더 효과적인 것으로 간주됩니다. 데이터를 압축하면 블록의 크기가 줄어들어 이러한 이점이 사라집니다.

VDEV 복구 (Repairing VDEVs)

중복 VDEV에 속한 공급자가 장애가 발생하면, 해당 공급자가 속한 VDEV는 "degraded(성능 저하)" 상태가 됩니다. 성능이 저하된 VDEV는 여전히 모든 데이터를 보유하지만 성능이 저하될 수 있습니다. 5장에서는 장애가 발생한 공급자를 교체하는 방법을 다룹니다.

공급자가 교체된 후에는 시스템이 새 공급자에 데이터를 저장해야 합니다. 미러를 사용하면 남은 디스크에서 데이터를 읽고 교체된 디스크에 쓰면 됩니다. RAID-Z의 경우, 데이터를 패리티에서 다시 계산해야 합니다.

ZFS가 RAID와 파일 시스템을 결합하는 방식은 ZFS가 어떤 블록에 데이터가 들어 있고 어떤 블록이 비어 있는지를 알고 있다는 것을 의미합니다. 새 드라이브에 모든 바이트를 기록할 필요 없이 ZFS는 실제로 사용 중인 블록만 기록하면 됩니다. 기존 RAID 컨트롤러는 파일 시스템 계층을 이해하거나 인식하지 못하기 때문에 무엇이 사용 중이고 무엇이 여유 공간인지 알 수 없습니다. RAID 컨트롤러는 디스크를 교체할 때 새 디스크의 모든 바이트를 복사해야 합니다. 즉, 손상된 ZFS RAID-Z가 훨씬 더 빨리 복구되어 데이터 손실을 유발할 수 있는 동시 장애의 가능성이 줄어듭니다. 5장에서 ZFS 복구에 대해 설명합니다.

RAID-Z와 기존 RAID 비교 (RAID-Z versus Traditional RAID)

RAID-Z는 기존 RAID에 비해 여러 가지 장점이 있지만, 가장 큰 장점은 ZFS가 디스크 중복성 계층 외에 볼륨 관리자이자 파일 시스템이라는 사실에서 비롯됩니다.

과거에는 파일시스템이 하나의 디스크에서만 작동할 수 있었습니다. 디스크가 두 개라면 두 개의 파일시스템이 필요했습니다. 기존 RAID를 사용하면 여러 디스크를 하나의 가상 디스크로 결합하여 100MB 이상의 대용량 디스크를 만들 수 있었습니다! 그런 다음 운영 체제는 물리적 디스크에 블록이 어떻게 배치될지 전혀 이해하지 못한 채 그 위에 자체 파일 시스템을 얹었습니다. 동시에 RAID는 내결함성을 제공할 수 있습니다. 당시 하드웨어와 소프트웨어의 한계를 고려할 때 RAID는 꽤 괜찮은 선택으로 보였습니다.

파일 시스템과 볼륨 관리자를 결합함으로써 ZFS는 모든 데이터가 어디에 있는지, 스토리지 계층과 데이터가 어떻게 상호 작용하는지 정확히 파악할 수 있습니다. 이를 통해 ZFS는 디토 블록(3장)과 같은 중요한 데이터의 추가 복사본을 별도의 디스크에 저장하는 것과 같은 여러 가지 중요한 결정을 내릴 수 있습니다. 하나의 기본 스토리지 공급업체에 중요한 데이터의 사본을 두세 개씩 저장해 두면 한 번의 하드웨어 장애로 인해 데이터가 모두 지워질 수 있기 때문입니다. ZFS는 인접한 디스크에 동일한 블록을 배치하는데, 이는 두 개의 디스크가 동시에 장애가 발생할 가능성이 통계적으로 낮기 때문입니다.

기존 RAID는 2단계 작업이 중간에 중단되는 "쓰기 구멍 (write hole)"이라는 단점이 있습니다. RAID 5 및 6 장치는 모든 데이터 디스크에 기록할 데이터를 청크 단위로 묶습니다. 이 작업이 완료되면 패리티 블록이 계산되어 패리티 디스크에 저장됩니다. 데이터가 기록된 후 패리티가 기록되기 전에 시스템이 충돌하거나 전원이 차단되면 디스크는 불확실한 상태가 됩니다. 시스템이 다시 시작되면 데이터가 패리티와 일치하지 않습니다. 미러링된 드라이브에서도 한 드라이브는 업데이트를 완료하고 다른 드라이브는 완료하지 못하면 같은 일이 발생할 수 있습니다.

쓰기 구멍 문제는 고장난 디스크를 교체하기 전까지는 발견되지 않습니다. 패리티가 잘못되었거나 미러가 잘못되면 RAID 장치가 파일시스템에 가비지 데이터를 반환하게 됩니다. 기존 파일시스템은 이 가비지 데이터를 파일의 콘텐츠로 반환합니다.

ZFS는 쓰기 시 복사 및 체크섬으로 이러한 문제를 해결합니다. 쓰기 시 덮어쓰기 (Copy-on-write, 7장)는 데이터를 제자리에서 덮어쓰지 않는다는 의미입니다. 각 업데이트는 트랜잭션 방식으로 진행되며, 완전히 완료되거나 수행되지 않아 시스템을 업데이트 전 상태로 되돌립니다. 또한 ZFS에는 체크섬이 있어 드라이브가 잘못된 데이터를 반환할 때 이를 감지할 수 있습니다. ZFS는 유효하지 않은 데이터를 감지하면 해당 데이터를 데이터의 추가 복사본, 미러링된 드라이브 또는 RAID-Z 패리티와 같은 다른 소스의 올바른 데이터로 대체합니다. 이러한 기능을 결합하여 ZFS의 자가 복구 속성을 생성합니다.

특수 VDEV (Special VDEVs)

풀은 풀의 성능을 향상시키기 위해 특수 목적의 VDEV를 사용할 수 있습니다. 이러한 특수 VDEV 유형은 데이터를 영구적으로 저장하는 데 사용되지 않고, 대신 더 빠른 장치에 데이터의 추가 사본을 임시로 보관합니다.

별도의 인텐트 로그 (Seperate Intent Log; SLOG, ZIL)

ZFS는 풀의 일부로 ZFS 인텐트 로그 (ZIL)를 유지합니다. 다른 파일시스템의 저널과 마찬가지로 진행 중인 작업을 기록하는 곳으로, 시스템 충돌이나 정전 시 작업을 완료하거나 롤백할 수 있습니다. ZIL은 디스크의 정상적인 작동 조건에 따라 달라집니다. 풀의 사용량이 갑자기 급증하거나 부하와 관련된 지연 시간이 발생하여 성능이 저하될 수 있습니다.

성능을 향상시키는 한 가지 방법은 ZIL을 일반 풀 작업에서 분리하는 것입니다. 풀의 일반 부분을 사용하는 대신 전용 장치를 별도의 인텐트 로그 (SLOG)로 사용할 수 있습니다. 전용 장치는 일반적으로 매우 높은 내구성의 SSD와 같이 작지만 매우 빠른 장치입니다.

ZFS는 데이터를 수신한 SLOG를 순서대로 풀의 메인 스토리지에 복사하는 대신 데이터를 합리적인 그룹으로 일괄 처리하여 보다 효율적으로 쓸 수 있습니다.

특정 소프트웨어는 디스크에 쓰는 데이터가 실제로 디스크에 있는지 확인을 받은 후에 작업을 진행합니다. 데이터베이스는 시스템 충돌이나 정전 시 손상을 방지하기 위해 이 작업을 수행하는 경우가 많습니다. 특정 NFS 작업도 마찬가지입니다. 이러한 요청을 더 빠른 로그 장치에 쓰고 "all done"를 보고함으로써 ZFS는 이러한 작업을 가속화합니다. 데이터베이스는 트랜잭션을 완료하고 계속 진행합니다. 저렴한 디스크를 저장 매체로 사용하면서 거의 SSD 수준의 쓰기 성능을 얻을 수 있습니다.

ZIL을 미러링하여 데이터 손실을 방지할 수 있습니다.

캐시 (Cache: L2ARC)

디스크에서 파일을 읽을 때 시스템은 다른 용도로 메모리가 필요할 때까지 파일을 메모리에 보관합니다. 이것은 초기 BSD 시절에도 사용된 오래된 기술입니다. UFS 기반 BSD 시스템의 상단 (1)을 보면 Buf라고 표시된 메모리 덩어리를 볼 수 있습니다. 이것이 바로 버퍼 캐시입니다.

하지만 전통적인 버퍼 캐시는 수십 년 전에 설계되었습니다. ZFS에는 최신 하드웨어용으로 설계된 적응형 대체 캐시 (ARC)가 있어 속도가 향상됩니다. ARC는 가장 최근에 자주 액세스한 파일을 유지합니다.

하지만 최신 시스템 중 원하는 만큼 캐시를 저장할 수 있는 충분한 RAM을 갖춘 시스템은 거의 없습니다. ZFS가 SLOG를 사용해 쓰기를 가속화하는 것처럼, 매우 빠른 디스크를 사용해 읽기를 가속할 수 있습니다. 이를 레벨 2 ARC 또는 L2ARC라고 합니다.

개체가 캐싱의 이점을 누릴 만큼 자주 사용되지만 RAM에 저장할 만큼 자주 사용되지는 않는 경우 ZFS는 캐시 장치에 개체를 저장할 수 있습니다. L2ARC는 일반적으로 매우 빠르고 내구성이 뛰어난 SSD 또는 NVMe 장치입니다. 이제 해당 데이터 블록이 요청되면 나머지 풀을 구성하는 느린 디스크가 아닌 더 빠른 SSD에서 데이터를 읽을 수 있습니다. ZFS는 백엔드 디스크에서 어떤 데이터가 변경되었는지 알기 때문에 읽기 캐시가 스토리지 풀의 데이터와 동기화되도록 할 수 있습니다.

VDEV가 성능에 미치는 영향 (How VDEVs Affect Performance)

각기 다른 유형의 VDEV는 서로 다른 성능을 발휘합니다. 디스크 성능을 벤치마킹하고 분석하는 것은 매우 복잡한 주제이므로, 누군가 귀찮아서 읽지 않는다면 커다란 교과서 한 권이 필요할 것입니다. 여기서 구체적인 조언을 하면 금방 지루해지므로 몇 가지 일반적인 용어에 대해서만 논의하겠습니다.

일반적인 측정 방법 중 하나는 초당 입출력 횟수, 즉 드라이브가 초당 수행할 수 있는 고유한 작업의 수를 나타내는 초당 입출력 횟수 (IOPS)입니다. 회전식 드라이브의 IOPS는 일반적으로 읽기/쓰기 헤드가 플래터 위에서 얼마나 빨리 이동할 수 있는지에 따라 물리적으로 제한됩니다. 솔리드 스테이트 디스크는 물리적으로 아무것도 움직일 필요가 없기 때문에 성능이 매우 뛰어납니다.

비패리티 스핀들의 수는 손상되지 않은 풀의 스트리밍 읽기 및 쓰기 성능을 제한합니다. "스트리밍" 성능은 드라이브가 읽거나 쓸 수 있는 초당 메가바이트 (MB/s)의 수로 요약됩니다. 드라이브가 데이터를 순차적으로 읽거나 쓸 때, 헤드가 서로 다른 위치를 왔다 갔다 할 필요가 없습니다. 이러한 조건에서 드라이브는 최상의 스트리밍 성능을 달성하여 최고의 처리량을 제공합니다. 스핀들 수는 랜덤 및 스트리밍 성능 모두에 영향을 미칩니다. 스핀들 및 헤드 수가 많을수록 IOPS와 스트리밍 성능이 모두 향상되므로 일반적으로 12개의 1테라바이트 (12 x 1TB) 드라이브 어레이가 6개의 2테라바이트 (6 x 2TB) 드라이브 어레이보다 성능이 더 뛰어납니다. 헤드 수가 많다는 것은 ZFS가 한 번에 디스크의 더 많은 다른 위치에서 읽거나 쓸 수 있다는 것을 의미하며, 그 결과 IOPS 성능이 향상됩니다. 스핀들이 많다는 것은 더 많은 디스크가 최대한 빠르게 작동하여 데이터를 읽고 쓸 수 있다는 것을 의미합니다. 그러나 드라이브 수가 많을수록 더 큰 셸프 또는 섀시, 더 많은 전력, 더 많은 컨트롤러가 필요합니다.

다른 일반적인 측정 항목으로는 읽기 대역폭, 쓰기 대역폭, 공간 효율성, 스트리밍 성능 등이 있습니다.

일반적으로 미러는 더 나은 IOPS와 읽기 대역폭을 제공할 수 있지만, RAID-Z는 더 나은 쓰기 대역폭과 훨씬 더 나은 공간 효율성을 제공할 수 있습니다.

여러 개의 VDEV가 있는 풀은 모든 VDEV에 걸쳐 데이터를 스트라이핑합니다 . 이렇게 하면 성능이 향상되지만 , 각 VDEV마다 중복 디스크가 있기 때문에 공간을 많이 차지할 수 있습니다 . 여러 개의 VDEV가 있는 풀은 안정성과 내결함성이 향상될 수 있습니다 . ZFS의 중복성은 모두 VDEV 수준이지만 , 중복 VDEV가 여러 개 있는 풀은 더 많은 디스크 장애를 견딜 수 있습니다 . 풀에 VDEV가 많을수록 풀의 성능은 더 좋아집니다 .

몇 가지 일반적인 VDEV 구성을 살펴보고 가능한 다양한 배열이 성능과 용량에 어떤 영향을 미치는지 알아보겠습니다 . 일반적인 1TB 스피닝 디스크 세트를 사용한다고 가정해 보겠습니다 . 각 디스크는 250 IOPS 와 100MB/s의 스트리밍 읽기 /쓰기가 가능합니다 .

디스크 하나 (One Disk)

디스크가 하나뿐인 경우 가능한 구성은 단일 ZFS 스트라이프 VDEV 하나뿐입니다 . 이 구성은 가장 기본적인 구성이며 내결함성을 제공하지 않습니다 . 디스크 하나가 죽으면 모든 데이터가 사라집니다 .

Table 1. Single Disk Virtual Device Configurations

Disks	Config	Read IOPS	Write IOPS	Read MB/s	Write MB/s	Usable Space	Fault Tolerance
1	Stripe	250	250	100	100	1 TB (100%)	none

단일 디스크 스트라이프 장치의 성능 특성은 기본 디스크의 특성과 의심스러울 정도로 비슷합니다 . 이상하지 않나요 ?

두 개의 디스크 (Two Disks)

시스템에 두 개의 디스크가 있는 경우 , 두 개의 스트라이프 VDEV 또는 하나의 미러 VDEV로 풀을 구성할 수 있습니다 .

스트라이프 VDEV는 사용 가능한 스토리지와 대역폭을 두 배로 늘리지만 장애 위험도 두 배로 증가합니다 . ZFS는 각 파일의 블록을 두 개의 디스크에 분산시킵니다 . 디스크 중 하나에 장애가 발생하면 모든 데이터를 사용할 수 없게 됩니다 .

미러링된 단일 VDEV를 사용하면 각 데이터 블록이 두 디스크에 모두 저장됩니다 . 이렇게 하면 두 디스크에서 한 번에 블록을 읽을 수 있으므로 읽기 성능이 향상됩니다 . 하지만 가장 작은 디스크의 용량만 사용할 수 있습니다 . 쓰기 성능은 가장 느린 디스크의 속도로 제한됩니다 . 그러나 하나의 디스크에 장애가 발생해도 풀은 계속 사용할 수 있습니다 .

Table 2: Two-Disk Virtual Device Configurations

Disks	Config	Read IOPS	Write IOPS	Read MB/s	Write MB/s	Usable Space	Fault Tolerance
2	2 x Stripe	500	500	200	200	2 TB (100%)	none
2	1 x 2 disk Mirror	500	250	200	100	1 TB (50%)	1

표에서 볼 수 있듯이 미러 풀은 스트라이프 풀의 쓰기 성능은 절반이고 공간은 절반입니다 .

디스크 3개 (Three Disks)

디스크 3개는 더 깊은 미러와 RAID-Z를 포함하여 더 많은 옵션을 의미합니다. 스트라이프 디스크 3개로 구성된 풀을 사용할 수도 있지만 실패 확률이 훨씬 높습니다.

더 깊은 미러에는 더 많은 디스크가 있으므로 더 많은 내결함성과 향상된 읽기 성능을 제공합니다. 스핀들과 헤드가 많다는 것은 VDEV가 세 개의 드라이브 중 가장 사용량이 적은 드라이브에서 데이터를 읽을 수 있어 랜덤 읽기를 더 빠르게 제공할 수 있다는 의미입니다. 미러의 쓰기 성능은 여전히 미러에 있는 드라이브 중 가장 느린 드라이브로 제한됩니다.

RAID-Z1은 내결함성을 위해 VDEV의 디스크 중 하나만 필요하므로 공간 효율성이 더 뛰어납니다. 데이터는 모든 드라이브에 분산되어 있으므로 읽기 및 쓰기를 수행하려면 함께 작동해야 합니다. 모든 드라이브에 데이터를 분산하면 스트리밍 쓰기 성능이 향상됩니다. 미러와 달리 RAID-Z에서는 각 드라이브가 동일한 데이터를 쓰는 대신 모든 드라이브가 동시에 자신의 몫의 데이터를 쓸 수 있습니다.

Table 3: Three-Disk Virtual Device Configurations

Disks	Config	Read IOPS	Write IOPS	Read MB/s	Write MB/s	Usable Space	Fault Tolerance
3	1 x 3 disk Mirror	750	250	300	100	1 TB (33%)	2
3	1 x 3 disk RAID-Z1	250	250	200	200	2 TB (66%)	1

여기서 주의할 점은 IOPS가 반드시 실제 읽기/쓰기 성능으로 확장되는 것은 아니라는 점입니다. 미러 VDEV는 각 드라이브의 헤드가 독립적으로 작동할 수 있는 반면, RAID-Z에서는 헤드가 함께 작동해야 하기 때문에 읽기 IOPS가 RAID-Z1의 3배에 달합니다. 초당 메가바이트 단위로 미러는 모든 디스크의 처리량을 데이터에 사용할 수 있다는 장점이 있는 반면, RAID-Z1은 패리티 데이터로 인해 디스크 한 개 분량의 처리량이 손실됩니다. 또한 3 디스크 미러는 모든 디스크에 동일한 데이터를 쓰기 때문에 쓰기 속도가 절반인 MB/s인 반면, RAID-Z1은 모든 디스크에 쓰기를 분산할 수 있지만 패리티로 인해 처리량이 일부 손실됩니다.

디스크 4개 또는 5개 디스크 (Four or Five Disks)

4개 또는 5개의 디스크를 사용하면 더 많은 옵션을 사용할 수 있습니다.

다중 미러 VDEV(기존 RAID 10과 유사)는 데이터베이스와 같은 랜덤 I/O 워크로드에 최상의 성능을 제공합니다. 4개의 디스크를 각각 2개의 디스크가 있는 2개의 미러 VDEV로 나누면 ZFS는 두 미러에 걸쳐 쓰기를 스트라이핑합니다. 한 미러는 데이터의 절반을, 다른 미러는 나머지 절반을 저장합니다. 이렇게 하면 미러의 쓰기 병목 현상을 완화하는 동시에 인상적인 읽기 성능을 제공할 수 있습니다.

디스크가 4개이면 RAID-Z2가 옵션이 됩니다. RAID-Z2의 패리티 디스크 2개는 디스크 2개가 손실되더라도 VDEV가 계속 작동할 수 있음을 의미합니다. 디스크 수가 같은 미러와 비교하면 성능은 떨어지지만, 디스크 두 개가 동시에 장애가 발생해도 더 이상 문제가 되지 않습니다.

5개의 디스크로 RAID-Z3를 구축할 수 있습니다. RAID-Z3 VDEV는 디스크 3개가 모두 손실되더라도 살아남을 수 있습니다. RAID-Z3는 성능과 내결함성을 교환합니다.

물론 RAID-Z1은 여전히 옵션으로 남아 있습니다 .

Table 4: Four- or Five-Disk Virtual Device Configurations

Disks	Config	Read IOPS	Write IOPS	Read MB/s	Write MB/s	Usable Space	Fault Tolerance
4	2 x 2 disk Mirror	1000	500	400	200	2 TB (50%)	2 (1/VDEV)
4	1 x 4 disk RAIDZ-Z1	250	250	300	300	3 TB (75%)	1
4	1 x 4 disk RAIDZ-Z2	250	250	200	200	2 TB (50%)	2
5	1 x 5 disk RAIDZ-Z1	250	250	400	400	4 TB (80%)	1
5	1 x 5 disk RAIDZ-Z2	250	250	300	300	3 TB (60%)	2
5	1 x 5 disk RAIDZ-Z3	250	250	200	200	2 TB (40%)	3

RAID-Z1의 스트리밍 (MB/s) 읽기 및 쓰기 성능이 RAID-Z2와 어떻게 비교되는지 , RAID-Z3의 성능은 둘 다와 어떻게 비교되는지 참고하세요 . 패리티 디스크를 추가하면 해당 디스크의 처리량이 저하될 수 있습니다 .

다중 미러 VDEV의 내결함성은 약간 까다롭습니다 . 중복성은 풀 단위가 아니라 VDEV 단위라는 점을 기억하세요 . 각 미러 VDEV는 여전히 n - 1의 내결함성을 제공합니다 . 각 미러 VDEV에서 하나의 드라이브가 계속 작동하는 한 , 모든 데이터에 액세스할 수 있습니다 . 풀에 두 개의 디스크 미러 VDEV가 있는 경우 , 각 VDEV에서 하나의 디스크가 손실되더라도 계속 실행할 수 있습니다 . 하지만 동일한 VDEV에서 디스크 두 개가 손실되면 풀이 죽고 모든 데이터가 손실됩니다 .

6~12개의 디스크 (Six to Twelve Disks)

디스크 수가 많으면 내결함성 , 공간 효율성 , 성능의 균형을 맞추는 것이 중요합니다 .

6개의 디스크를 3개의 2 디스크 미러 VDEV로 구성하면 상당한 공간과 우수한 쓰기 성능을 확보할 수 있습니다 . 한 쌍의 3-디스크 미러 VDEV를 선택하면 공간은 더 적지만 데이터 손실 위험 없이 각 디스크 세트 3 개 중 2개에 장애가 발생할 수 있습니다 . 또는 RAID-Z VDEV가 될 수도 있습니다 .

디스크를 6개 이상 확보하면 한 풀에 여러 개의 RAID-Z VDEV를 사용할 수 있습니다 . 12개의 디스크를 단일 VDEV로 함께 운영하여 가용 공간을 최대한 확보하거나 , 두 개의 개별 VDEV로 분할하여 가용 공간은 줄어들지만 성능과 내결함성은 향상시킬 수 있습니다 .

Table 5: Six- to Twelve-Disk Virtual Device Configurations

Disks	Config	Read IOPS	Write IOPS	Read MB/s	Write MB/s	Usable Space	Fault Tolerance
6	3 x 2 disk Mirror	1500	750	600	300	3 TB (50%)	3 (1/VDEV)

6	2 x 3 disk Mirror	1500	500	600	200	2 TB (33%)	4 (2/VDEV)
6	1 x 6 disk RAIDZ-Z1	250	250	500	500	5 TB (83%)	1
6	1 x 6 disk RAIDZ-Z2	250	250	400	400	4 TB (66%)	2
6	1 x 6 disk RAIDZ-Z3	250	250	300	300	3 TB (50%)	3
12	6 x 2 disk Mirror	3000	1500	1200	600	6 TB (50%)	6 (1/VDEV)
12	4 x 3 disk Mirror	3000	1000	1200	400	4 TB (33%)	8 (2/VDEV)
12	1 x 12 disk RAIDZ-Z1	250	250	1100	1100	11 TB (92%)	1
12	2 x 6 disk RAIDZ-Z1	500	500	1000	1000	10 TB (83%)	2 (1/VDEV)
12	3 x 4 disk RAIDZ-Z1	750	750	900	900	9 TB (75%)	3 (1/VDEV)
12	1 x 12 disk RAIDZ-Z2	250	250	1000	1000	10 TB (83%)	2
12	2 x 6 disk RAIDZ-Z2	500	500	800	800	8 TB (66%)	4 (2/VDEV)
12	1 x 12 disk RAIDZ-Z3	250	250	900	900	9 TB (75%)	3
12	2 x 6 disk RAIDZ-Z3	500	500	600	600	6 TB (50%)	6 (3/VDEV)

풀에서 여러 RAID-Z 장치를 사용하는 것은 풀에서 여러 미러 장치를 사용하는 것과 매우 유사합니다 . 디스크 장애에 대한 허용 오차는 풀이 아닌 VDEV별로 적용됩니다 . 6개의 디스크 RAID-Z2 VDEV 2개로 구성된 12 개의 디스크 어레이는 VDEV당 2개의 디스크만 손실되는 경우 4개의 디스크 손실을 처리할 수 있습니다 .

많은 디스크 (Many Disks)

일반적으로 VDEV당 9~12개 이하의 디스크를 사용하는 것이 좋습니다 . 더 많이 사용할 수도 있지만 ZFS는 그렇게 설계되지 않았습니다 . 36개의 디스크를 배열하여 몇 가지 가능한 배열과 성능에 미치는 영향을 살펴보겠습니다 .

Table 5: Six- to Twelve-Disk Virtual Device Configurations

Disks	Config	Read IOPS	Write IOPS	Read MB/s	Write MB/s	Usable Space	Fault Tolerance
36	18 x 2 disk Mirror	9000	4500	3600	1800	18 TB (50%)	18 (1/VDEV)

36	12 x 3 disk Mirror	9000	3000	3600	1200	12 TB (33%)	24 (2/VDEV)
36	1 x 36 disk RAID-Z2	250	250	3400	3400	34 TB (94%)	2
36	2 x 18 disk RAID-Z2	500	500	3200	3200	32 TB (89%)	4 (2/VDEV)
36	4 x 9 disk RAID-Z2	1000	1000	2800	2800	28 TB (78%)	8 (2/VDEV)
36	6 x 6 disk RAID-Z2	1500	1500	2400	2400	24 TB (66%)	12 (2/VDEV)

더 많은 VDEV를 사용하면 매우 빠른 풀을 만들 수 있습니다. 18개의 디스크 미러 VDEV로 구성된 풀은 그 어떤 것보다 더 빠르게 데이터를 읽을 수 있으며, 장애가 발생하기 전에 18개의 드라이브를 잃을 수도 있습니다! 물론 18개의 드라이브가 정확히 일치해야 하지만, 전원 공급 장치가 다른 두 개의 디스크 셸이 있다면 충분히 가능합니다. 반면에 해당 풀에 있는 디스크 두 개가 잘못되면 전체 풀이 죽습니다.

각 VDEV에 패리티 또는 미러를 추가하면 안정성이 향상됩니다. VDEV의 수가 많을수록 성능이 향상됩니다. 여러분의 역할은 이 두 가지 특성을 적절히 조합하여 환경을 지원하는 것입니다.

각 VDEV는 가장 느린 디스크의 랜덤 읽기/쓰기 성능으로 제한되므로 하나의 VDEV에 디스크가 너무 많으면 공간 효율성을 조금만 높이기 위해 성능을 포기하는 것입니다. 성능을 개선하기 위해 L2ARC 및 SLOG 장치를 추가할 수는 있지만, 이러한 문제를 완전히 피하는 것이 가장 좋습니다.

그렇다면 VDEV가 많으면 많을수록 좋은데, 왜 6 x 6 디스크 RAID-Z2 풀이 1 x 36 디스크 RAID-Z2 풀에 비해 읽기 및 쓰기 속도가 훨씬 느린 것일까요? 답은 내결함성 열에 있습니다. RAID-Z2 VDEV가 많을수록 중복성이 높아져 더 많은 장애를 견딜 수 있습니다. 디스크가 내결함성을 제공하는 경우, 디스크는 데이터의 여분의 복사본을 저장하므로 디스크에 장애가 발생했을 때 손실된 복사본을 대체할 수 있습니다. 시스템은 데이터가 변경될 때마다 패리티 데이터를 다시 계산하여 저장합니다. 패리티 데이터는 원본 사본이 없는 경우를 제외하고는 파일을 읽을 때 사용되지 않습니다. 패리티에 사용된 디스크는 더 이상 스트리밍 성능에 기여하지 않습니다. 디스크를 더 추가하여 해당 성능을 복원할 수 있습니다. 6 x 8 디스크 RAID-Z2 풀은 36개의 데이터 디스크와 12개의 패리티 디스크로 구성되며, 1 x 36 디스크 RAID-Z2 풀보다 뛰어난 성능을 발휘할 수 있습니다.

VDEV에 대해 알고 있는 내용을 바탕으로 실제 풀을 만들어 보겠습니다.

챕터 3. 풀 (Pools)

ZFS 풀 또는 zpool은 ZFS 스택의 중간을 형성하며, 하위 수준의 가상 장치를 사용자가 볼 수 있는 파일 시스템에 연결합니다. 풀 보기 시스템의 모든 풀을 보려면 스토리지 블록의 zpoolocating을 실행하세요. ZFS 풀 수준에서 ZFS 데이터 세트에 사용할 수 있는 공간을 늘리거나 특수 가상 장치를 추가하여 읽기 또는 쓰기 성능을 향상시킬 수 있습니다.

ZFS 블록

UFS 및 extfs와 같은 기존 파일시스템은 디스크에 데이터를 고정 크기 블록으로 저장합니다. 파일 시스템에는 아이노드(inode)라고 하는 특수 블록이 있으며, 이 블록은 어떤 블록이 어떤 파일에 속하는지 색인합니다. NTFS나 FAT와 같은 비유닉스 파일시스템도 비슷한 구조를 사용합니다. 이는 업계 전반의 표준입니다.

ZFS는 특별한 인덱스 블록을 미리 구성하지 않습니다. 스트라이프라고도 하는 스토리지 블록만 사용합니다. 각 블록에는 블록을 디스크의 다른 블록과 트리로 연결하기 위한 인덱스 정보가 포함되어 있습니다. ZFS는 블록의 모든 정보에 대한 해시를 계산하고 해당 정보를 블록과 상위 블록에 저장합니다. 각 블록은 그 자체로 완전한 단위입니다. 파일이 부분적으로 누락될 수 있지만 존재하는 내용은 일관성을 유지합니다.

전용 인덱스 블록이 없어도 좋지만, ZFS는 분명 어딘가에서 시작해야 합니다! 모든 데이터 트리에는 루트가 필요합니다. ZFS는 우버블록이라는 특수 블록을 사용해 파일 시스템 루트에 대한 포인터를 저장합니다. ZFS는 디스크의 데이터를 변경하지 않으며, 오히려 블록이 변경되면 수정된 데이터로 블록의 완전히 새로운 복사본을 씁니다. (이 쓰기 시 복사본 동작에 대해서는 6장에서 자세히 설명합니다.) 데이터 풀은 128개의 블록을 우버블록용으로 예약하고, 기본 풀이 변경될 때 순차적으로 사용합니다. 마지막 우버블록이 사용되면 ZFS는 처음으로 돌아갑니다.

우버블록만이 중요한 블록은 아닙니다. ZFS는 파일시스템 메타데이터 및 풀 데이터와 같은 중요한 정보가 포함된 블록을 여러 개의 동일한 블록에 복사합니다. 메인 블록이 손상되면 ZFS는 디토 블록(ditto block, 동일한 내용을 복사한)에서 백업 복사본을 확인합니다. 동일한 블록은 별도의 디스크 또는 단일 디스크의 별도 부분에 가능한 한 서로 멀리 떨어져 저장됩니다. (ZFS는 디스크 하드웨어의 레이아웃을 볼 수 있는 특별한 기능은 없지만 대충 짐작할 수 있습니다.)

ZFS는 트랜잭션 그룹 또는 txg의 스토리지 미디어에 변경 내용을 커밋합니다. 트랜잭션 그룹은 여러 개의 일괄 변경 내용을 포함하며 64비트 숫자가 증가합니다. 각 트랜잭션 그룹은 라인의 다음 우버블록을 사용합니다. ZFS는 128개의 트랜잭션 그룹 중에서 가장 높은 트랜잭션 번호를 가진 우버블록을 찾아서 가장 최신의 우버블록을 식별합니다.

ZFS는 인덱싱을 위해 일부 블록을 사용하지만, 이러한 z노드와 d노드는 풀에 있는 모든 스토리지 블록을 사용할 수 있습니다. 파일시스템을 만들 때 할당되는 UFS2 또는 extfs 인덱스 노드와는 다릅니다.

스트라이프, RAID 및 풀 (Stripes, RAID, and Pools)

스토리지와 관련하여 스트라이프라는 단어를 여러 번 들어보셨을 것입니다. ZFS 풀은 가상 장치에 걸쳐 데이터를 "스트라이프"합니다. 전통적인 RAID는 물리적 장치에 걸쳐 데이터를 "스트라이프"합니다.

스트라이프란 무엇이며, 풀에서 어떻게 작동할까요?

스트라이프는 단일 장치에 기록되는 데이터 덩어리입니다. 대부분의 기존 RAID는 128KB 스트라이프 크기를 사용합니다. 기존 RAID 장치에 파일을 쓸 때 RAID 소프트웨어는 보통 병렬로 128KB 단위로 각 드라이브에 씁니다. 마찬가지로 기존 RAID 어레이에서 읽는 작업도 스트라이프 크기 단위로 이루어집니다. 서버의 워크로드에 맞게 스트라이프 크기를 사용자가 지정할 수 있지만, 하드웨어의 용량과 소프트웨어의 한계로 인해 스트라이프 크기가 크게 제한됩니다.

스트라이프는 중복성을 제공하지 않습니다. 기존 RAID는 패리티 및/또는 미러링을 통해 중복성을 확보합니다. ZFS 풀은 기본 VDEV에서 중복성을 확보합니다.

ZFS는 로켓으로 구동되는 롤러스케이트에 줄무늬를 넣습니다. ZFS 데이터 세트는 기본 스트라이프 크기인 128KB를 사용하지만, ZFS는 장비와 워크로드에 맞게 이 스트라이프 크기를 동적으로 변경할 수 있을 만큼 똑똑합니다. 특정 데이터 청크에는 32KB 스트라이프 크기가 적합하지만 다른 데이터 조각에는 64KB가 적합할 경우, ZFS는 각 데이터에 적합한 크기를 사용합니다. ZFS 개발자들은 최대 1MB의 스트라이프 크기에 대한 지원을 완료했습니다. 이 기능은 이미 FreeBSD-CURRENT에서 사용할 수 있으며, FreeBSD 10.2 이상에 포함될 예정입니다.

ZFS 풀은 기존 RAID보다 훨씬 더 유연합니다. 전통적인 RAID는 고정적이고 유연하지 않은 데이터 레이아웃을 가지고 있습니다(일부 하드웨어 벤더는 유연성이 더 높은 자체 독점 RAID 시스템을 가지고 있습니다). RAID 소프트웨어는 결정론적 순서로 각 디스크에 씁니다. ZFS는 더 많은 유연성을 제공합니다. 5개의 디스크로 구성된 기존 RAID 어레이를 사용하는 경우, 해당 어레이는 항상 5개의 디스크로 구성됩니다. 디스크를 추가하여 어레이를 변경할 수 없습니다. 디스크를 더 큰 디스크로 교환할 수는 있지만, 그렇게 해도 어레이의 크기는 변경되지 않습니다. RAID 장치를 생성하면 어레이의 기본 특성이 석화됩니다.

ZFS 풀은 변경을 허용할 뿐만 아니라 추가도 쉽게 허용하도록 설계되었습니다. 다섯 개의 VDEV가 있는 ZFS 풀이 있는데 여섯 번째를 추가하려는 경우 괜찮습니다. ZFS는 해당 VDEV를 수락하고 깜박임 없이 해당 장치에서 데이터 스트라이핑을 시작합니다. RAID-Z VDEV에는 스토리지를 추가할 수 없고 풀에는 VDEV만 추가할 수 있습니다. RAID-Z VDEV의 공급자 수는 생성 시 고정되어 있습니다.

하지만 ZFS를 사용하면 해당 가상 장치는 ZFS가 지원하는 모든 유형의 VDEV가 될 수 있습니다. 미러 쌍인 두 개의 VDEV를 예로 들어보겠습니다. 하나의 zpool에 넣습니다. ZFS가 데이터를 스트라이핑합니다. 기존 RAID에서는 미러 상단의 스트라이프를 RAID-10이라고 합니다. 대부분의 사용 사례에서 RAID-10은 가장 높은 성능의 RAID입니다. 그러나 기존 RAID-10의 크기가 고정되어 있는 경우 풀에 VDEV를 추가할 수 있습니다. RAID-10을 확장한다는 것은 데이터를 백업하고, RAID 어레이에 디스크를 추가하고, 데이터를 복원하는 것을 의미합니다. zpool을 확장한다는 것은 풀에 더 많은 VDEV를 추가한다는 의미입니다. 또한 RAID-10은 최대 2개의 디스크 깊이를 허용하는 반면, ZFS는 최대 264개의 디스크 깊이를 허용합니다.

하지만 풀은 중복성을 제공하지 않는다는 점을 기억하세요. 모든 ZFS 중복성은 기본 VDEV에서 제공됩니다.

풀 보기 (Viewing Pools)

시스템의 모든 풀을 보려면 **zpool list**를 실행합니다.

```
$ zpool list
NAME  SIZE  ALLOC  FREE  EXPANDSZ  FRAG  CAP  DEDUP  HEALTH  ALTROOT
```

```
db    2.72T  1.16G  2.72T       -    0%   0%  1.00x  ONLINE  -
zroot  920G  17.3G  903G       -    2%   1%  1.00x  ONLINE  -
```

첫 번째 열은 풀 이름입니다 . 이 시스템에는 **db**와 **zroot**라는 두 개의 풀이 있습니다 .
다음 세 열은 각 풀의 크기와 사용량 정보를 제공합니다 . 크기 , 사용된 공간의 양 , 사용 가능한 공간의 양이 표시됩니다 .
EXPANDSZ 열은 기본 스토리지 공급업체에 여유 공간이 있는지 여부를 보여줍니다 . 5장에서 설명한 대로 이 풀의 공간을 확장할 수 있습니다 . 이 공간에는 패리티 정보로 이동하는 블록이 포함되어 있으므로 풀을 확장해도 사용 가능한 공간이 이만큼 확보되지는 않습니다 .
FRAG 아래에서 이 풀의 조각화 양을 볼 수 있습니다 . 조각화는 파일 시스템 성능을 저하시킵니다 .
CAP 열은 사용 가능한 공간의 몇 퍼센트가 사용되었는지 보여줍니다 .
DEDUP 항목은 파일 시스템에서 발생한 중복 제거의 양을 보여줍니다 . 6장에서는 중복 제거에 대해 다룹니다 .
풀의 **HEALTH** 열은 기본 VDEV의 상태를 반영합니다 . 스토리지 공급업체에 장애가 발생하면 첫 번째 힌트는 온라인 이외의 모든 상태입니다 . 5장에서는 풀 상태에 대해 설명합니다 .
마지막으로 , **ALTROOT**는 이 풀이 마운트된 위치 , 즉 "대체 루트 "를 보여줍니다 .

4장에서는 대체 루트에 대해 다룹니다 . 특정 풀에 대한 정보를 알고 싶으면 zpool list 뒤에 풀 이름을 나열합니다 .
이 명령은 스토리지 풀 **prod** 및 **test**의 출력만 표시합니다 .

```
$ zpool list prod test
```

기본 드라이브의 사용률을 포함하여 풀에 대한 자세한 정보를 원하면 **-v** 옵션을 추가합니다 . 이 옵션은 풀 이름 앞에 붙여야 합니다 .

```
$ zpool list -v zroot
```

-p 플래그는 사람이 읽기 쉬운 형식이 아닌 바이트 단위로 숫자를 인쇄하고 , **-H**는 열 헤더를 제거합니다 . 이러한 옵션은 자동화 및 관리 스크립트에 유용합니다 .

기본 VDEV 레이아웃을 포함하여 시스템 풀을 더 자세히 보려면 **zpool status**를 사용하세요 . 풀을 만들 때 zpool 상태의 많은 예를 보게 될 것입니다 .

풀을 간단히 확인하려면 zpool status -x를 실행합니다 .

```
$ zpool status -x
all pools are healthy
```

때로는 이것만으로 충분할 때도 있습니다 .

여러 개의 VDEV (Multiple VDEVs)

풀에는 여러 개의 VDEV가 포함될 수 있습니다 . VDEV를 추가하면 풀에서 사용 가능한 공간이 늘어날 뿐만 아니라 성능도 향상됩니다 . 풀은 모든 쓰기를 VDEV 간에 분할합니다 . 작은 파일은 단일 스트라이프만 있으면

되지만, 여러 개의 작은 파일을 쓰는 경우 ZFS는 VDEV 간에 쓰기를 분할합니다.

2장에서는 다양한 VDEV 유형의 성능에 대해 설명합니다. 이러한 성능은 풀 레벨로 올라갑니다. 여러 VDEV에서 대용량 파일을 읽는 경우, 파일 읽기는 마지막(일반적으로 가장 느린) 드라이브가 해당 데이터의 일부 호출을 완료하면 완료됩니다. 하지만 풀에 여러 대의 VDEV가 포함된 경우, 가장 느린 드라이브에는 파일의 일부만 포함되어 있으므로 파일에 액세스하는 데 필요한 시간이 다소 줄어듭니다. 스토리지 공급업체에서 데이터를 읽을 때 가장 느린 부분은 데이터를 호출할 올바른 디스크의 헤드를 찾는 것이므로 시간을 VDEV의 수로 나누는 것만큼 간단하지는 않지만, 풀에 VDEV를 추가하면 성능이 향상됩니다.

모범 사례에서는 풀에 동일한 스토리지 VDEV만 사용하는 것이 좋습니다. 풀에 미러링된 VDEV가 여러 개 있는 경우 풀에 RAID-Z3 장치를 추가하지 마세요. 혼합 스토리지 VDEV는 풀 성능을 크게 떨어뜨리고 장치 간에 데이터를 최적으로 분산하는 ZFS의 작업을 더 힘들게 만듭니다. 이 작업을 수행할 수는 있지만, 하지 않는 것이 좋습니다.

> ZFS는 유닉스의 전통에 따라 멍청한 일을 하는 것을 막지 않는데, 이는 영리한 일을 하는 것도 막는 것이기 때문입니다.

VDEV 제거 (Removing VDEVs)

현재 풀에서 VDEV를 제거할 수 없습니다. 각 VDEV에는 데이터가 있습니다. 특정 유형의 VDEV에서 디스크를 제거할 수 있지만, VDEV 전체에는 중요한 풀 데이터가 남아 있습니다. 예를 들어, 미러 VDEV에서 디스크를 제거할 수는 있지만 풀에서 전체 VDEV를 제거할 수는 없습니다. 일반적으로는 장애가 발생한 경우에만 VDEV에서 디스크를 제거합니다. 스토리지 공급업체를 끌어당기는 등 풀에서 강제로 VDEV를 제거하면 풀이 파괴됩니다. 스트라이프 또는 미러 풀에서 VDEV를 제거하는 기능은 2015년 말에 OpenZFS에 제공될 예정이지만 아직은 불가능합니다. RAID-Z 장치 제거에 대한 지원은 로드맵에 있지만 아직 작업이 시작되지 않았습니다. 즉, 풀을 축소할 수 없습니다.

풀을 더 작게 만들려면 해당 풀의 데이터를 새롭고 더 작은 풀로 옮긴 다음 원래 풀에서 디스크를 재활용해야 합니다.

풀 정렬 및 디스크 섹터 크기 (Pools Alignment and Disk Sector Size)

ZFS는 기본 제공업체의 섹터 크기를 포함하여 스토리지 매체에 대한 심층적인 지식이 있어야 합니다. 풀이 올바른 섹터 크기를 사용하지 않거나 ZFS의 섹터가 디스크의 물리적 섹터에 정렬되지 않으면 스토리지 성능이 절반 이상 저하됩니다. 이 두 가지 문제는 직교하는 문제이지만, 둘 중 하나라도 계획하지 않으면 시스템이 망가질 수 있습니다.

파티션 정렬과 ZFS 섹터 크기에 대해서는 별도로 설명하겠습니다.

파티션 정렬 (Partition Alignment)

디스크는 섹터 크기를 보고하므로 문제가 되는 경우를 제외하고는 문제가 되지 않습니다. 많은 디스크가 512바이트 섹터를 가지고 있다고 보고하지만, 실제로는 4096바이트(4K) 섹터를 가지고 있습니다. FreeBSD Mastery: Storage Essentials에서 이에 대해 자세히 다루고 있으므로 여기서는 자세히 다루지 않겠습니다.

유서 깊은 마스터 부트 레코드(MBR)와 같은 오래된 파티션 관리 체계는 디스크 파티션이 디스크의 물리적 특성에 부합하는지 확인하기 위해 온갖 종류의 복잡한 수학을 포함했습니다. GPT(GUID Partition Table)와 같은 최신 파티션 체계는 물리적 디스크가 서로 다른 언어를 사용한다는 것을 알고 있으며, 이러한 오래된 MBR 기반 제한이 완전히 엉터리라는 것을 알기 때문에 파티션이 완전한 섹터를 채우도록 요구합니다.

하지만 디스크가 섹터 크기에 대해 거짓말을 하는 경우, **gpart(8)**을 사용하면 물리적 섹터의 중간에서 시작하거나 끝나는 파티션을 만들 수 있습니다. 디스크를 읽거나 쓸 때마다 두 개의 물리적 섹터를 건드려야 합니다. 이는 성능에 큰 영향을 미칩니다.

특정 SSD는 파티션이 128KB 또는 1MB 경계를 따라 정렬될 것을 기대하기도 합니다.

정렬 문제를 피하는 쉬운 방법은 모든 GPT 파티션을 메가바이트 경계에서 시작하고 끝내도록 하는 것입니다. gpart 추가 명령에 **-a 1m** 인수를 추가하세요.

ZFS 섹터 크기 (ZFS Sector Size)

ZFS는 기본적으로 512바이트 섹터 크기를 가정합니다. 물리적 512바이트 섹터가 있는 디스크에서 512바이트 파일 시스템 섹터 크기를 사용하는 것은 전혀 문제가 없습니다. 4K-섹터 디스크에 512바이트 파일 시스템 섹터를 사용하면 하드웨어가 더 많이 작동합니다. 이러한 디스크에 4KB의 데이터를 쓰고 싶다고 가정해 보겠습니다. 하드 드라이브에 하나의 물리적 섹터를 쓰도록 지시하는 대신 섹터의 첫 번째 8분의 1을 수정하고, 두 번째 8분의 1을 수정하고, 세 번째 8분의 1을 수정하는 식으로 수정하라는 지시를 내립니다. 4KB 섹터에 512바이트 쓰기를 수행한다는 것은 4KB 전체를 읽고, 작은 부분을 수정한 다음 다시 쓰는 것을 의미합니다. 이는 전체 섹터를 덮어쓰는 것보다 훨씬 느립니다. 성능이 급감합니다. 512바이트 섹터가 있는 디스크에서 ZFS가 4K 섹터 크기를 사용하는 경우, 디스크 하드웨어는 액세스 요청을 물리적 섹터 크기로 분할하여 성능 비용을 거의 들이지 않습니다.
더 큰 섹터 크기를 사용한다고 해서 성능에 영향을 주지는 않지만, 작은 파일을 많이 저장할 경우 공간 효율성이 떨어집니다. 1KB 파일이 여러 개 있는 경우 각 파일이 하나의 섹터를 차지합니다.

ZFS 섹터 크기는 풀에 있는 각 가상 장치의 속성입니다. 풀을 내보내거나 고장난 드라이브를 교체할 때에도 가상 장치의 섹터 크기는 변경할 수 없습니다.

이 두 가지 사실을 종합하면, 기본 디스크에서 보고하는 섹터 크기에 관계없이 ZFS가 4K-섹터를 사용하도록 강제하는 것이 거의 항상 바람직하다는 것을 의미합니다. 더 큰 ZFS 섹터 크기를 사용해도 특정 데이터베이스 작업을 제외하고는 성능이 저하되지 않으며, 그 경우에도 실제로 512바이트 섹터를 사용하는 디스크를 사용하는 경우에만 성능이 저하되지 않습니다.

ZFS는 가장 큰 섹터 크기를 보고하는 장치의 섹터 크기를 사용합니다. 모든 장치가 512바이트 섹터를 사용한다고 주장하고 더 큰 섹터 크기를 설정하지 않으면 해당 장치로 구축된 가상 장치가 512바이트 섹터를 사용합니다. 4096바이트 섹터를 가진 단일 디바이스를 VDEV에 포함하면 ZFS가 4096바이트 섹터를 사용하도록 강제합니다.

4K-섹터 디바이스가 섹터 크기를 보고한다고 믿지 마세요. 항상 4K-섹터를 사용하도록 ZFS에 지시하세요.

ashift라는 풀 변수가 섹터 크기를 제어합니다. ashift가 9이면 ZFS가 512바이트 섹터를 사용하도록 지시합니다. ashift가 12이면 ZFS가 4096바이트 섹터를 사용하도록 지시합니다. (왜 9와 12일까요? 2^9는 512이고, 2^{12}는 4096입니다.) ashift를 설정하는 방법은 FreeBSD 릴리스에 따라 다릅니다.

> 모두가 '9'를 보고 2^9를 생각하잖아요 ?

FreeBSD 10.1 및 새로운 Ashift (FreeBSD 10.1 and Newer Ashift)

/etc/sysctl.conf 또는 명령줄에서 **sysctl vfs.zfs.min_auto_ashift**를 사용하여 시스템의 기본 ashift를 설정합니다 .

```
$ sysctl vfs.zfs.min_auto_ashift=12
```

설치 중에 명령줄을 사용하지만 , 새 풀을 만들 때 잊지 않도록 **/etc/sysctl.conf**에 영구적으로 설정할 수도 있습니다 .

이 책의 예제에서는 FreeBSD 10.1 이상을 사용한다고 가정합니다 . 이전 FreeBSD 버전의 경우 , sysctl을 설정하는 대신 매번 ashift를 설정해야 합니다 .

구형 FreeBSD Ashift (Older FreeBSD Ashift)

10.1 이전의 FreeBSD 버전에는 최신 FreeBSD 버전에 있는 ashift sysctl이 없기 때문에 , ZFS의 내부 섹터 크기 감지 코드에 의존해야 합니다 . 이 코드는 기본 스토리지 매체 , 즉 스토리지 공급자로부터 섹터 크기를 읽습니다 .

이 사례는 공급자와 디스크의 중요한 차이점을 강조합니다 . FreeBSD에서는 GEOM 모듈 **gnop(8)**을 사용하여 패스스루 장치를 만들 수 있습니다 . gnop 모듈을 사용하면 저장 장치 사이에 임의의 데이터를 삽입할 수 있습니다 (이 경우 섹터 크기를 강제 적용). "모든 것을 투명하게 통과시키되 , 4096바이트 섹터 크기를 고집합니다 ."라는 gnop 장치를 생성합니다 . 이 장치를 사용하여 zpool을 만듭니다 . 여기서는 **/dev/gpt/zfs0**이라는 파티션에 gnop 장치를 추가합니다 .

```
$ gnop create -S 4096 /dev/gpt/zfs0
```

이렇게 하면 **/dev/gpt/zfs0.nop** 장치가 생성됩니다 . 이 공급자를 VDEV의 한 멤버로 사용하면 ZFS가 해당 VDEV의 섹터 크기를 선택합니다 . 이 장의 나머지 부분에서는 다양한 ZFS 풀을 생성하는 방법에 대해 설명하지만 , 여기서는 미러 풀을 생성할 때 이 장치를 사용하는 예제를 살펴보겠습니다 .

```
$ zpool create compost mirror gpt/zfs0.nop gpt/zfs1
```

gnop(8)로 생성된 프로바이더는 임시적이며 재부팅 시 사라집니다 . 하지만 **gnop(8)**이 모든 것을 디바이스로 전달하기 때문에 ZFS는 기본 디바이스에서 메타데이터를 찾습니다 . 이미 섹터 크기가 설정되어 있으므로 ZFS는 더 이상 디스크의 섹터 크기를 감지하려고 시도하지 않습니다 .

풀과 가상 디바이스 만들기 (Creating Pools and VDEVs)

zpool(8)을 사용하여 풀과 가상 디바이스를 동시에 생성합니다 . 또한 **zpool(8)**을 사용하여 기존 풀에 VDEV를 추가하고 고장난 디바이스를 교체할 수도 있지만 , 이 모든 내용은 5장에서 다루겠습니다 . 여기서는 각 RAID-Z 장치에 스트라이프 풀 , 미러 및 풀을 생성하겠습니다 . 2장에서는 각 VDEV 유형에 대해 설명합니다 .

원하는 만큼의 풀을 생성하기 전에 ashift를 한 번만 설정하면 됩니다. 풀을 생성할 때마다 재설정할 필요는 없습니다. 하지만 대부분의 독자가 자신이 생성하는 풀 유형에 대한 항목을 찾을 때까지 이 책을 건너뛸 것으로 예상되므로 모든 항목에 "set ashift"을 나열했습니다.

샘플 드라이브 (Sample Drives)

0장에서는 고장난 하드웨어를 쉽게 식별할 수 있도록 물리적 위치와 일련 번호로 드라이브에 라벨을 붙일 것을 권장합니다. 프로덕션에서는 매우 유용합니다. 하지만 책에서는 장치 이름이 길면 이해하기 어렵습니다. 이 예제에서는 zfs와 숫자로 된 GPT 레이블을 사용합니다. 이 장에서는 각각 1GB 스왑 파티션과 **gpart(8)**로 만든 대형 ZFS 파티션이 있는 6개의 1TB 드라이브를 사용합니다.

```
$ gpart create -s gpt da0
$ gpart add -a 1m -s1g -l sw0 -t freebsd-swap da0
$ gpart add -a 1m -1 zfs0 -t freebsd-zfs da0
```

결과 디스크의 파티션은 다음과 같습니다.

```
$ gpart show -l da0
=>     40  1953525088  da0  GPT  (932G)
       40       2008    -  free -  (1.0M)
     2048    2097152    1  sw0  (1.0G)
  2099200  1951424512    2  zfs0  (931G)
1953523712       1416    -  free -  (708K)
```

우리는 GPT 레이블로 ZFS 풀을 관리하므로 예제에서는 gpt/zfs0부터 gpt/zfs5까지를 참조합니다. 프로덕션 환경에서는 디스크를 실제 위치에 매핑하는 의미 있는 레이블을 사용하세요.

스트라이프 풀 (Striped Pools)

일부 스토리지 풀은 중복성이 필요하지 않지만 많은 공간이 필요합니다. 엔지니어링 및 물리 계산을 위한 스크래치 파티션이 이러한 종류의 스토리지의 일반적인 사용 사례입니다. 풀 이름인 **zpool create**를 사용해 풀에 있는 장치를 나열합니다. 풀을 만들기 전에 애쉬시프트를 설정하는 것을 잊지 마세요. 여기서는 5개의 스토리지 공급자로 구성된 풀을 생성합니다.

```
$ sysctl vfs.zfs.min_auto_ashift=12
$ zpool create compost gpt/zfs0 gpt/zfs1 gpt/zfs2 gpt/zfs4 gpt/zfs4
```

명령이 성공하면 아무런 출력도 돌아오지 않습니다. 풀이 **zpool status**로 존재하는지 확인합니다.

```
$ zpool status
  pool: compost
 state: ONLINE
  scan: none requested
```

```
config:

NAME       STATE READ WRITE CKSUM
compost    ONLINE   0     0     0
 gpt/zfs0  ONLINE   0     0     0
 gpt/zfs1  ONLINE   0     0     0
 gpt/zfs2  ONLINE   0     0     0
 gpt/zfs3  ONLINE   0     0     0
 gpt/zfs4  ONLINE   0     0     0
```

5개의 공급자가 모두 표시됩니다. 각 공급자는 자체 VDEV입니다.

이 정도 규모의 시스템에서는 상당히 큰 풀입니다. 이 풀은 모든 구성원 VDEV에 걸쳐 데이터를 스트라이핑하지만, VDEV에는 중복성이 없습니다. 대부분의 실제 애플리케이션에는 중복성이 필요합니다. 가장 간단한 종류의 중복성은 미러입니다.

미러 풀 (Mirrored Pools)

미러링된 장치는 모든 데이터를 여러 스토리지 공급업체에 복사합니다. 미러의 한 공급업체에 장애가 발생해도 풀에는 여전히 다른 데이터 사본이 있습니다. 기존 미러에는 두 개의 디스크가 있지만, 그 이상도 가능합니다.

동일한 **zpool create** 명령과 풀 이름을 사용합니다. **mirror** 키워드를 사용해 시스템 시프트 저장 장치를 설정합니다. 풀을 생성하기 전에 시스템 ashift를 설정하세요.

```
$ sysctl vfs.zfs.min_auto_ashift=12
$ zpool create reflect mirror gpt/zfs0 gpt/zfs1
```

zpool status로 풀의 구성을 확인합니다.

```
$ zpool status
  pool: reflect
 state: ONLINE
  scan: none requested
config:

NAME        STATE READ WRITE CKSUM
reflect     ONLINE   0     0     0
 mirror-0   ONLINE   0     0     0
  gpt/zfs0  ONLINE   0     0     0
  gpt/zfs1  ONLINE   0     0     0
```

```
errors: No known data errors
```

zpool 명령은 여기에 mirror-0이라는 새 계층을 만들었습니다. mirror-0 항목은 VDEV입니다. 이 VDEV에는 두 개의 장치, 즉 **gpt/zfs0**과 **gpt/zfs1**이 포함되어 있습니다. 필요에 따라 여러 개의 디스크로 미러를 구성할 수 있습니다. 사본이 너무 많은 것이 없는 것보다 낫습니다.

```
$ zpool create reflect mirror gpt/zfs0 gpt/zfs1 gpt/zfs2 gpt/zfs3
```

그러나 이것은 너무 지나친 예가 될 수 있습니다 (FreeBSD Mastery: Advanced ZFS에서 미러를 여러 풀로 분할하는 것에 대해 설명합니다).

RAID-Z Pools

미러에서 얻을 수 있는 중복성은 빠르고 안정적이지만, 매우 복잡하거나 흥분되지는 않습니다. RAID-Z는 복잡성 대가로 더 큰 유연성을 제공하므로 더 흥미롭습니다. 다른 zpool과 마찬가지로 **zpool create**을 실행하고 풀 이름, 유형 및 저장 장치를 지정하여 RAID-Z 풀을 생성합니다. 여기서는 RAID-Z(또는 RAID-Z1) 풀을 생성합니다.

> 시스템 관리에서 '흥분'은 나쁜 단어입니다.

```
$ sysctl vfs.zfs.min_auto_ashift=12
$ zpool create raidz1 gpt/zfs0 gpt/zfs1 gpt/zfs2
```

새 풀의 상태에는 세 개의 공급자가 있는 raidz1-0이라는 새 VDEV가 표시됩니다.

```
$ zpool status bucket
  pool: bucket
 state: ONLINE
  scan: none requested
config:

NAME        STATE READ WRITE CKSUM
bucket      ONLINE   0     0     0
 raidz1-0   ONLINE   0     0     0
  gpt/zfs0  ONLINE   0     0     0
  gpt/zfs1  ONLINE   0     0     0
  gpt/zfs2  ONLINE   0     0     0
```

이 풀에 있는 디스크 중 하나에 장애가 발생해도 데이터는 그대로 유지됩니다. 다른 RAID-Z 레벨에는 훨씬 더 많은 중복성이 있습니다. 여기서는 6개의 공급자를 raidz3-Z3으로 끌어옵니다. RAID-Z3를 만들 때와 RAID-Z1을 만들 때의 유일한 차이점은 raidz3와 필요한 추가 장치의 사용 여부입니다.

```
$ zpool create bucket raidz3 gpt/zfs0 gpt/zfs1 gpt/zfs2 gpt/zfs3 gpt/zfs4 gpt/zfs5
```

지금쯤이면 짐작할 수 있겠지만, 풀의 상태에는 raidz3-0이라는 새 장치가 표시됩니다.

```
$ zpool status
  pool: bucket
 state: ONLINE
  scan: none requested
config:

NAME        STATE  READ WRITE CKSUM
bucket      ONLINE    0     0     0
 raidz3-0   ONLINE    0     0     0
  gpt/zfs0  ONLINE    0     0     0
  gpt/zfs1  ONLINE    0     0     0
...
```

이러한 풀에는 모두 단일 VDEV가 있습니다. 하지만 여러 개의 VDEV를 원한다면 어떻게 해야 할까요?

다중 VDEV 풀 (Multi-VDEV Pools)

여러 개의 VDEV가 있는 풀을 만들 수 있습니다. *mirror, raidz, raidz2, raidz3* 키워드는 모두 **zpool(8)**에 새 VDEV를 생성하도록 지시합니다. 이러한 키워드 중 하나 뒤에 나열된 모든 스토리지 공급자는 해당 VDEV의 새 인스턴스를 생성합니다. 키워드 중 하나가 다시 나타나면 **zpool(8)**은 새 VDEV로 시작합니다.

이 장의 시작 부분에서는 기존 RAID-10 설정을 시뮬레이션하여 여러 미러에 걸쳐 스트라이핑하는 방법을 다루었습니다. 여기서는 정확히 그렇게 합니다.

```
$ sysctl vfs.zfs.min_auto_ashift=12
$ zpool create barrel mirror gpt/zfs0 gpt/zfs1 mirror gpt/zfs2 gpt/zfs3
```

처음 세 단어인 **zpool create barrel**은 **zpool(8)**에 barrel이라는 이름의 새 풀을 인스턴스화하라고 지시합니다. **mirror** 키워드는 "미러 생성"을 의미합니다. 그러면 두 개의 스토리지 프로바이더, 즉 **gpt/zfs0**과 **gpt/zfs1**이 생깁니다. 이 스토리지 공급자는 첫 번째 미러로 이동합니다. **mirror**라는 단어가 다시 나타나면서 이전 VDEV가 완료되었고 새 VDEV에서 시작한다는 것을 **zpool(8)**에 알려줍니다. 두 번째 VDEV에도 두 개의 스토리지 프로바이더, 즉 **gpt/zfs2**와 **gpt/zfs3**가 있습니다. 이 풀의 상태는 이전에 보았던 것과는 다릅니다.

```
$ zpool status barrel
  pool: barrel
 state: ONLINE
  scan: none requested
```

```
config:

NAME        STATE READ WRITE CKSUM
barrel      ONLINE   0     0     0
 mirror-0  ONLINE   0     0     0
  gpt/zfs0 ONLINE   0     0     0
  gpt/zfs1 ONLINE   0     0     0
 mirror-1  ONLINE   0     0     0
  gpt/zfs2 ONLINE   0     0     0
  gpt/zfs3 ONLINE   0     0     0
```

풀에는 mirror-0과 mirror-1이라는 두 개의 VDEV가 있습니다. 각 VDEV에는 두 개의 저장 장치가 포함됩니다. ZFS는 모든 VDEV에 걸쳐 데이터를 스트라이핑합니다. 미러에 대한 스트라이핑은 RAID-10입니다. 일반적인 RAID와 동등한 방식이 없는 방식으로 다중 VDEV 풀을 배열할 수도 있습니다. FreeBSD의 일부 GEOM 클래스와 같은 소프트웨어 RAID 시스템을 사용하면 유사한 RAID를 구축할 수 있지만, 하드웨어 RAID 카드에서는 이러한 기능을 찾을 수 없습니다. 여기서는 두 개의 RAID-Z1 VDEV에 걸쳐 데이터를 스트라이핑하는 풀을 생성합니다.

```
$ zpool create vat raidz1 gpt/zfs0 gpt/zfs1 gpt/zfs2 raidz1 gpt/zfs3 gpt/zfs4 gpt/zfs5
```

첫 번째 RAID-Z1 VDEV에는 세 가지 스토리지 공급업체, 즉 **gpt/zfs0, gpt/zfs1, gpt/zfs2**가 포함됩니다. 두 번째는 **gpt/zfs3, gpt/zfs4, gpt/zfs5**를 포함합니다. zpool은 두 공급자에 걸쳐 데이터를 스트라이핑합니다. 이렇게 하면 두 개의 RAID-Z 장치가 포함된 풀이 생성됩니다.

```
$ zpool status vat
...
config:

NAME        STATE READ WRITE CKSUM
vat        ONLINE   0     0     0
 raidz1-0  ONLINE   0     0     0
  gpt/zfs0 ONLINE   0     0     0
  gpt/zfs1 ONLINE   0     0     0
  gpt/zfs2 ONLINE   0     0     0
 raidz1-1  ONLINE   0     0     0
  gpt/zfs3 ONLINE   0     0     0
  gpt/zfs4 ONLINE   0     0     0
  gpt/zfs5 ONLINE   0     0     0
```

각 VDEV에는 고유한 중복성이 있습니다.

미러가 RAIDZ보다 빠르긴 하지만 , 여러 대의 VDEV를 사용하면 속도가 빨라져 이 RAIDZ 기반 풀이 워크로드에 충분히 빠르고 훨씬 더 많은 공간을 제공할 수 있습니다 . 이를 확인할 수 있는 유일한 방법은 풀을 생성하고 워크로드를 테스트하는 것입니다 .

풀은 모든 쓰기 요청을 풀에 있는 VDEV 간에 분할한다는 점을 기억하세요 . 하나의 작은 파일은 하나의 VDEV 에만 전송될 수 있지만 , 전체적으로는 쓰기 요청이 VDEV 간에 분할됩니다 . 여러 개의 VDEV를 사용하면 IOPS 와 처리량 대역폭이 증가합니다 .

로그 장치 사용 (Using Log Devices)

2장에서 설명한 대로 , ZFS는 전용 쓰기 캐시 장치 및 /또는 전용 읽기 캐시 장치를 사용하여 성능을 향상시킬 수 있습니다 . 이러한 전용 장치는 일반적으로 매우 빠르고 내구성이 높은 SSD입니다 . **zpool(8)** 명령은 쓰기 캐시를 **log**, 읽기 캐시를 **cache**로 호출합니다 . 풀을 만들 때 **log** 및 **cache** 키워드를 사용하여 이러한 장치를 지정합니다 . 여기서는 읽기 및 쓰기 캐시가 모두 포함된 스크래치 라는 이름의 스트라이프 풀을 만듭니다 .

```
$ zpool create scratch gpt/zfs0 log gtp/zlog0 cache gpt/zcache1
```

로그 장치가 풀의 상태에 표시됩니다 .

```
$ zpool status scratch
...
config:

NAME         STATE READ WRITE CKSUM
scratch      ONLINE   0     0     0
 gpt/zfs0    ONLINE   0     0     0
logs
 gpt/zlog0   ONLINE   0     0     0
cache
 gpt/zcache1 ONLINE   0     0     0
```

고가용성이 필요한 시스템에서는 이러한 쓰기 캐시를 미러링할 수 있습니다 . 읽기 캐시를 미러링하는 것은 큰 의미가 없습니다 . 읽기 캐시가 손실되면 ZFS는 실제 풀에서 읽기로 돌아가기 때문입니다 . 그러나 ZIL 쓰기 로그가 손실되면 데이터 손실이 발생할 수 있으므로 미러링하는 것이 좋습니다 . 여기서는 *gpt/zfs0*부터 *gpt/zfs3*까지 장치를 사용해 두 개의 미러 스트라이프를 만들고 , 미러링된 로그 장치로 *gpt/zlog0*과 *gpt/zlog1*을 사용합니다 .

```
$ zpool create db mirror gpt/zfs0 gpt/zfs1 mirror gpt/zfs2 gpt/zfs3 log mirror gpt/zlog0 gpt/zlog1
```

기존 풀에 인텐트 로그 및 읽기 캐시 장치를 추가하거나 제거할 수 있습니다 . 이러한 장치의 성능 향상이 꼭 필요한지 확실하지 않은 경우 , 장치 없이 풀을 실행해 보세요 . 하지만 나중에 SSD 저장 장치를 추가할 수 있는 공간이 하드웨어에 있는지 확인하세요 !

일치하지 않는 VDEV (Mismatched VDEVs)

풀 내에서 서로 다른 VDEV 유형을 사용하는 것은 바람직하지 않으며, **zpool(8)**은 이러한 장애를 방지하려고 시도합니다.

```
$ zpool create daftie raidz gpt/zfs0 gpt/zfs1 gpt/zfs2 mirror gpt/zfs3 gpt/zfs4 gpt/zfs5
invalid vdev specification
use '-f' to override the following errors:
mismatched replication level: both raidz and mirror vdevs are present
```

zpool(8) 명령은 실수를 지적한 다음 어떻게 고쳐야 하는지 알려줍니다. 일반적으로 이런 종류의 오류는 시스템 관리자에게 카페인이 더 필요하다는 의미로 받아들이지만, 정말 의도한 것일 수도 있습니다. 지정된 VDEV 유형과 스토리지 공급자로 **zpool create -f**를 실행하면 ZFS는 사용자가 완전히 잘못된 풀을 만들려고 했다는 것을 알 수 있습니다. 이봐요, 이건 당신 시스템이고 당신이 책임자예요.

ZFS가 원하지 않는 작업을 수행하면 안 됩니다. **-f**를 사용하면 ZFS가 처리하도록 설계되지 않은 것을 만들게 됩니다. 제대로 작동하지 않고 복구할 수 없는 풀을 쉽게 만들 수 있습니다.

공급자 재사용 (Reusing Providers)

올바른 풀을 만들기 위해 풀을 두 번 이상 생성하고 파괴하는 경우가 있습니다. 한 머신에서 디스크를 가져와서 다른 머신에 마운트할 수도 있습니다. 이전에 사용했던 디스크를 발견할 때도 있습니다.

```
$ zpool create db gpt/zfs1 gpt/zfs2 gpt/zfs3 gpt/zfs4
invalid vdev specification
use '-f' to override the following errors:
/dev/gpt/zfs3 is part of exported pool 'db'
```

이 디스크를 나중에 내보낸 다른 풀에서 사용했습니다 (5장 참조). 문제가 발생한 디스크는 해당 풀에서 사용되었고 ZFS 레이블은 디스크에 그대로 남아있었습니다. 파티션 테이블을 지우고 다시 만들었지만, 새 파티션 테이블은 이전 파티션 테이블과 정확히 동일합니다. 이 경우 ZFS는 이전 메타데이터를 쉽게 찾을 수 있습니다.

이 공급업체에 중요한 정보가 없는 것이 확실하다면 지침에 따라 **-f**를 사용하여 새 풀을 강제로 생성하세요.

```
$ zpool create -f db gpt/zfs1 gpt/zfs2 gpt/zfs3 gpt/zfs4
```

ZFS 프로그램은 명령줄 플래그가 어디로 가는지 매우 까다로울 수 있으므로 **-f**가 create 바로 뒤에 오는지 확인하세요.

풀 무결성 (Pool Integrity)

ZFS에 대한 일반적인 불만 중 하나는 **fsck(8)**과 같은 파일 시스템 검사기가 없다는 것입니다. 온라인 풀 무결성 검사기가 **fsck(8)**이 검사하는 모든 것을 검증하기 때문에 오프라인 파일 검사기는 ZFS를 개선하지

못합니다. 또한 온라인 검사기는 기존 파일 시스템보다 훨씬 더 효과적입니다. 이제 ZFS가 파일 무결성을 보장하는 방법과 풀 스크러빙이 무결성을 유지하는 데 어떻게 도움이 되는지 알아보겠습니다.

ZFS 무결성 (ZFS Integrity)

저장 장치가 망가집니다. 어떤 종류의 디스크든 수조 개의 섹터가 있다면, 술에 취해 비틀거리며 돌아다니는 디스크에 우주선이 부딪힐 확률은 물론 쓰기 오류, 정전, 케이블 단락 등 다양한 문제가 발생할 확률이 훨씬 더 높아집니다. 어떤 파일 시스템도 기본 하드웨어의 오류를 막을 수는 없습니다.

ZFS는 거의 모든 곳에서 해시를 사용합니다. 해시는 데이터 덩어리를 가져와서 고정 길이의 문자열을 계산하는 수학적 알고리즘입니다. 해시의 흥미로운 점은 원본 데이터를 조금만 변경해도 데이터의 해시가 크게 달라진다는 것입니다. 각 스토리지 블록에는 상위 블록의 해시가 포함되며, 각 상위 블록에는 모든 하위 블록의 해시가 포함됩니다.

ZFS는 스토리지 공급자 오류를 방지할 수는 없지만, 이러한 해시를 사용하여 오류를 감지합니다. 시스템이 데이터에 액세스할 때마다 체크섬을 확인합니다. ZFS는 운영 체제에 수정된 파일을 제공하기 전에 풀의 중복성을 사용하여 오류를 복구합니다. 이를 자가 치유 **(self-healing)**라고 합니다.

기본 VDEV에 중복성이 있는 경우, ZFS는 RAID-Z에서 손상된 블록을 재구성하거나 미러에서 온전한 사본을 가져옵니다. 미러의 양쪽 모두에 오류가 있는 경우, 두 디스크에 동일한 데이터가 손상되지 않는 한 ZFS는 파일을 복구할 수 있습니다. VDEV에 중복성이 없지만 데이터 세트에 데이터의 여분의 복사본이 있는 경우 (4장 참조), ZFS는 그 여분의 복사본을 대신 사용합니다.

기본 VDEV에 중복성이 없고 데이터 세트에 여분의 복사본이 보관되어 있지 않은 경우, 풀은 파일이 손상되었음을 알리고 잘못된 데이터를 반환하는 대신 오류를 반환합니다. 백업에서 해당 파일을 복원하거나 버릴 수 있습니다.

ZFS는 파일 무결성 검사를 수행하는 동시에 스토리지 블록 간의 연결도 확인합니다. 이 작업은 기존 파일 시스템에서 **fsck(8)**가 수행하는 작업입니다. 이 작업은 데이터 검증의 작은 부분이며, ZFS는 정상 작동의 일부로 이 작업을 지속적으로 수행합니다. ZFS는 사용되거나 사용되지 않는 아이노드가 아니라 실제로 존재하는 블록만 검사한다는 점에서 **fsck(8)**에 비해 추가적인 이점이 있습니다. 풀의 모든 데이터에 대해 전체 무결성 검사를 수행하려면 풀을 스크러빙하세요.

해시 기반 무결성 검사의 좋은 점은 모든 종류의 오류, 심지어 예상치 못한 오류까지 잡아낸다는 것입니다. 정상적인 파일시스템은 모두 비슷하지만, 불행한 파일시스템은 모두 나름의 방식으로 불행하다는 사실을 기억하세요.

Scrubbing ZFS

ZFS 풀의 스크럽 (scrub)은 풀에 있는 모든 데이터 블록의 암호화 해시를 확인합니다. 스크럽이 오류를 감지하면 충분한 복원력이 있는 경우 오류를 복구합니다. 스크럽은 풀이 온라인 상태이고 사용 중일 때 수행됩니다.

풀에서 데이터 오류가 확인된 경우, 해당 데이터는 zpool의 상태에 표시됩니다. 이전에 스크럽을 실행한 적이 있다면 스캔 라인에 해당 정보도 표시됩니다.

```
...
scan: scrub repaired 0 in 15h57m with 0 errors on Sun Feb  8 15:57:55 2015
...
errors: No known data errors
...
```

이 풀은 액세스한 데이터에서 오류가 발생하지 않았습니다. 오류가 발견되었다면 자체적으로 복구했을 것입니다. 그러나 이 풀은 모든 데이터에서 오류를 검사한 것이 아니라 요청받은 데이터만 검사했습니다. 전체 풀에서 오류를 체계적으로 검색하려면 스크럽을 사용합니다. 풀 이름과 함께 zpool 스크럽을 실행합니다.

```
$ zpool scrub zroot
```

스크럽은 백그라운드에서 실행됩니다. **zpool status**를 실행하여 진행 상황을 확인할 수 있습니다.

```
$ zpool status
...
scan: scrub in progress since Tue Feb 24 11:52:23 2015
12.8G scanned out of 17.3G at 23.0M/s, 0h3m to go
0 repaired, 74.08% done
...
```

스토리지를 스크러빙하는 ZFS 풀은 평소보다 느리게 실행됩니다. 시스템이 이미 성능 한계에 도달한 경우에는 사용량이 많지 않은 시간에만 풀을 스크럽하세요. 진행 중인 스크럽을 취소해야 하는 경우, **zpool scrub -s**를 실행합니다.

```
$ zpool scrub -s zroot
```

돌아가서 가능한 한 빨리 시스템에서 스크럽을 완료하도록 하세요.

스크럽 빈도 (Scrub Frequency)

ZFS에 내장된 무결성 테스트와 복원력 덕분에 대부분의 오류는 복원력이 작동할 수 있을 만큼 조기에 발견된다면 수정할 수 있습니다. 즉, 하드웨어의 품질에 따라 호스트의 풀을 얼마나 자주 스크러빙해야 하는지가 결정됩니다. 소위 "서버급" 장비와 같이 신뢰할 수 있는 하드웨어를 사용하는 경우 분기별로 스크러빙하는 것으로 충분합니다. 값싼 하드웨어를 남용하는 경우 한 달에 한 번 정도 스크러빙해야 합니다.

이 장의 뒷부분에 있는 "ZFS 유지 관리 자동화"에서 설명한 대로 FreeBSD는 정기적인 스크럽을 수행할 수 있습니다.

풀 속성 (Pool Properties)

ZFS는 속성을 사용하여 풀의 특성을 표현합니다. zpool 속성은 데이터 집합의 속성과 매우 유사하게 보이고 작동하며 많은 속성이 둘 사이에 겹치는 것처럼 보이지만, 데이터 집합 속성은 풀 속성과 아무런 관계가

없습니다 . 풀 속성에는 풀의 상태 , 크기 , 용량 및 풀별 기능과 같은 사실이 포함됩니다 .

풀의 속성은 전체 풀에 영향을 줍니다 . 풀의 일부에만 속성을 설정하려면 필요에 맞는 데이터 세트별 속성을 확인하세요 .

풀 속성 보기 (Viewing Pool Properties)

시스템에 있는 모든 풀의 모든 속성을 보려면 **zpool get all**을 실행합니다 . 특정 풀의 속성만 보려면 끝에 풀 이름을 추가하면 됩니다 . 여기서는 **zroot** 풀에 대한 속성을 봅니다 .

```
$ zpool get all zroot
NAME   PROPERTY  VALUE   SOURCE
zroot  size      920G    -
zroot  capacity  1%      -
zroot  altroot   -       default
zroot  health    ONLINE  -
...
```

처음 두 열은 풀 이름과 속성 이름을 나타냅니다 .

세 번째 열에는 속성 값이 나열됩니다 . 이 값은 사용 또는 사용 안 함 , 켜짐 또는 꺼짐 , 활성 또는 비활성 등과 같은 값일 수도 있고 값일 수도 있습니다 . 이 풀의 크기 속성은 920G로 , 이 풀에는 920GB의 공간이 있습니다 .

SOURCE 열에는 이 속성이 설정된 위치가 표시됩니다 . 이 열은 대시 하나 또는 기본 또는 로컬이라는 단어로 표시될 수 있습니다 . 대시는 이 속성이 자체적으로 설정된 것이 아니라 어떻게든 풀에서 읽혀진 것임을 의미합니다 . 풀의 크기나 해당 공간의 사용량에 대한 값은 설정하지 않습니다 . FreeBSD는 풀에서 이러한 값을 계산합니다 . 기본값의 출처는 이 속성이 기본값으로 설정되어 있음을 나타내며 , 로컬은 이 속성이 이 풀에 특별히 설정되었음을 의미합니다 .

단일 속성을 가져오려면 속성 이름과 함께 **zpool get**을 실행합니다 .

```
$ zpool get size
NAME   PROPERTY  VALUE  SOURCE
db     size      2.72T  -
zroot  size      920G   -
```

마지막에 풀 이름을 지정하여 범위를 좁히세요 .

풀 속성 변경 (Changing Pool Properties)

이 책 전체에서 풀 동작을 변경하기 위해 속성을 설정하겠습니다 . **zpool set** 명령을 사용하여 풀의 속성을 변경합니다 . 여기서는 풀의 **comment** 속성을 설정합니다 .

```
$ zpool set comment="Main OS files" zroot
```

이제 이 코멘트가 속성 목록에 표시됩니다.

```
$ zpool get comment
NAME   PROPERTY  VALUE          SOURCE
db     comment   -              default
zroot  comment   Main OS files  local
```

여기서 SOURCE 열에 주목하세요. 기본적으로 풀에는 주석이 없습니다. 하지만 코멘트를 설정했으므로 소스가 로컬로 변경됩니다. 속성의 소스가 기본값에서 로컬로 변경되면 영원히 로컬로 유지됩니다. 속성을 기본값으로 설정해도 소스는 변경되지 않습니다.

```
$ zpool set comment="-" zroot
# zpool get comment
NAME   PROPERTY  VALUE  SOURCE
db     comment   -      default
zroot  comment   -      local
```

주석을 기본값으로 로컬 설정하므로 값의 소스는 로컬로 유지됩니다.

풀을 만들 때 **-o**를 사용하여 풀의 속성을 설정할 수 있습니다. 풀의 루트 데이터 세트에 대한 속성은 **-O**로 설정할 수 있습니다.

```
$ zpool create -o altroot=/mnt -O canmount=off -m none zroot /dev/gpt/disk0
```

이 풀의 **altroot** 속성은 */mnt*로 설정되어 있고, 이 풀의 루트 데이터 세트는 **canmount** 속성이 ***off***로 설정되어 있습니다. 속성으로 데이터 기록 방식이 변경되면 속성 변경 후 기록된 데이터만 영향을 받습니다. ZFS는 속성 변경에 따라 기존 데이터를 다시 쓰지 않습니다.

풀 히스토리 (Pool History)

모든 zpool은 풀이 생성된 시점부터 지금까지 풀에 적용된 모든 변경 사항의 사본을 보관합니다. 이 기록에는 시스템 전원 켜기 및 끄기와 같은 일상적인 이벤트는 포함되지 않지만, 속성 설정, 풀 업그레이드, 데이터 세트 생성은 포함됩니다.

기록에 액세스하려면 **zpool history**를 실행하고 풀 이름을 입력합니다.

```
$ zpool history zroot
History for 'zroot':
2014-01-07.04:12:05 zpool create -o altroot=/mnt -O canmount=off -m none zroot mirror /
dev/gpt/disk0.nop /dev/gpt/disk1.nop
2014-01-07.04:12:50 zfs set checksum=fletcher4 zroot
2014-01-07.04:13:00 zfs set atime=off zroot
...
```

숙련된 FreeBSD 사용자라면 FreeBSD 문서와 포럼에 있는 수많은 ZFS 튜토리얼을 통해 이 기능을 알아볼 수 있을 것입니다 .

기록은 다음과 같이 끝납니다 :

```
...
2015-03-12.14:36:35 zpool set comment=Main OS files zroot
2015-03-12.14:43:45 zpool set comment=- zroot
```

comment 속성을 변경했으므로 기록에 남습니다 . 영원히요 .

안타깝게도 풀 기록에서는 누가 각 변경을 수행했는지 추적할 수 없지만 , 변경 사항을 영구적으로 기록해 두면 문제 분석에 도움이 됩니다 .

Zpool 유지 관리 자동화 (Zpool Maintenance Automation)

FreeBSD는 **periodic(8)**로 실행되는 일일 유지 관리 작업의 일부로 각 시스템의 파일 시스템을 검사합니다 . 이 검사에 ZFS 풀 정보를 추가하면 풀 상태에 대한 정보를 얻을 수 있습니다 . *periodic.conf*의 **daily_status_zfs_enable** 옵션은 풀 검사를 활성화합니다 .

```
daily_status_zfs_enable="YES"
```

이제 매일 **periodic(8)** 출력에 일반적으로 "all pools are healthy."라는 한 줄의 출력인 **zpool status -x** 가 포함됩니다 .

풀에 대한 더 자세한 정보를 원할 경우 일일 보고서에 zpool 목록 출력을 포함할 수도 있습니다 . 목록을 가져오려면 **daily_status_zfs_zpool_list**를 **yes**로 설정하세요 . 특정 풀의 상태만 표시하여 해당 출력을 잘라내려면 *periodic.conf*의 **daily_status_zpool** 변수에 원하는 풀을 나열하세요 .

FreeBSD가 풀 스크럽을 수행하도록 할 수도 있습니다 . 스크러빙 옵션을 설정하면 FreeBSD는 풀에 스크러빙이 필요한지 매일 확인하지만 , 설정된 간격으로만 스크러빙을 수행합니다 . 35일마다 모든 풀을 자동으로 스크러빙하려면 , *periodic.conf*에서 **daily_scrub_zfs_enable**을 **YES**로 설정하세요 .

```
daily_scrub_zfs_enable="YES"
```

FreeBSD는 기본적으로 모든 풀을 스크러빙합니다 . 일일 스크럽 검사에서 특정 풀을 명시적으로 제외할 수는 없습니다 . 그러나 **daily_scrub_zfs_pools**에 검사할 풀을 명시적으로 나열할 수 있습니다 . 목록에 없는 풀은 스크럽되지 않습니다 .

```
daily_scrub_zfs_pools="zroot prod test"
```

스크럽 사이의 일 수를 변경하려면 **daily_scrub_zfs_default_threshold**를 원하는 일 수로 설정합니다 .

```
daily_scrub_zfs_default_threshold="10"
```

특정 풀을 다른 일정에 따라 스크러빙하려면 **daily_scrub_zfs_${poolname}_threshold**를 원하는 일수로 설정하세요. 여기서는 7일마다 풀 프로드를 스크러빙합니다.

```
daily_scrub_zfs_prod_threshold="7"
```

자체 임계값이 없는 풀은 기본 임계값을 사용합니다.

풀 제거 (Removing Pools)

풀을 제거하려면 **zpool destroy** 명령과 풀 이름을 사용합니다.

```
$ zpool destroy test
```

파괴는 풀의 기본 공급자를 파괴된 풀의 일부로 표시하여 다른 풀에 재사용할 수 있도록 합니다. 디스크를 지우지는 않으며, 5장을 읽은 사람이라면 누구나 풀을 복원하고 데이터에 액세스할 수 있습니다.

공급업체의 데이터를 안전하게 삭제하거나 덮어써야 하는 경우 디스크 덮어쓰기 또는 파쇄 프로그램이 필요합니다.

Zpool 기능 플래그 (Zpool Feature Flags)

ZFS와 풀에는 원래 풀에서 지원하는 기능을 나타내는 버전 번호가 있습니다. 시스템에서 익숙하지 않은 스토리지 공급업체를 보고 "이 풀은 ZFS 버전 20이므로 중복 제거나 기본 암호화를 지원하지 않습니다."라고 말할 수 있습니다. 풀을 릴리스에서 지원하는 최신 버전으로 업그레이드할 수도 있고 그렇지 않을 수도 있습니다.

그 후 오라클은 ZFS 소스를 폐쇄했고, 다양한 사람들이 마지막 오픈 소스 ZFS 릴리스를 가져와서 직접 유지 관리하도록 했습니다. 마지막 오픈 소스 오라클 ZFS 버전은 28이었습니다. 다양한 그룹이 자체 풀 기능을 구현함에 따라 서로 다른 그룹의 버전 번호가 상호 호환되지 않게 될 위험이 있었습니다. 각기 다른 ZFS 팀이 선택한 새로운 기능을 구현할 수 있었기 때문에, 예를 들어 FooZFS 버전 30은 BarZFS 버전 30과 호환되지 않을 수 있었습니다. ZFS의 주요 목표는 상호 운용성입니다.

OpenZFS 팀은 버전 번호로 기능을 추적하는 방식에서 벗어나는 것이 가장 좋은 방법이라고 판단했습니다. 이들은 OpenZFS 버전을 최대 5000개까지 늘려서 오라클이 새로운 버전을 추가할 수 있는 충분한 공간을 확보했습니다. 다양한 플랫폼의 다양한 OpenZFS 개발자를 모두 수용하기 위해 개발자들은 버전 번호를 기능 플래그로 효과적으로 대체하기로 결정했습니다.

FreeBSD를 포함하여 OpenZFS를 실행하는 모든 플랫폼에는 이 특정 설치가 지원하는 풀 기능을 나열하는 **zpool-features(7)** 매뉴얼 페이지가 포함되어 있어야 합니다. 최신 버전의 FreeBSD는 아마도 새로운 풀 기능을 지원할 것입니다.

기능을 사용하면 일반적으로 어떤 식으로든 온디스크 형식이 변경됩니다. 예를 들어, 스냅샷 지원을 추가한다는 것은 "이것은 스냅샷입니다."라는 새 필드와 메타데이터를 추가하는 것을 의미합니다. 해당 기능을 지원하지 않는 시스템에서는 이 풀을 보고 "이런, 이 데이터 구조를 인식하지 못합니다. 난 이걸 건드리지 않을 거야!"라고 말할 것입니다. 시스템 간에 디스크를 정기적으로 교체하는 경우, 새로운 기능 플래그를 업그레이드하거나 활성화하기 전에 여러 호스트에서 지원되는 기능 플래그를 주의 깊게 확인해야 합니다.

기능 플래그 보기 (Viewing Feature Flags)

풀에서 지원하는 기능 플래그와 해당 설정을 보려면 **"feature"**이라는 단어가 포함된 풀 속성을 찾습니다.

```
$ zpool get all zroot | grep feature
zroot  feature@async_destroy  enabled  local
zroot  feature@empty_bpobj    active   local
zroot  feature@lz4_compress   active   local
...
```

활성화된 풀 기능은 사용할 수 있지만 실제로는 아직 사용되지 않습니다. 시스템이 새로운 유형의 압축을 지원할 수 있지만 실제로 새 알고리즘을 사용하여 풀에 데이터를 쓰지 않은 경우입니다. 온디스크 형식이 해당 기능을 수용하도록 변경되지 않았기 때문에 이 풀은 해당 기능을 지원하지 않는 시스템에서 가져올 수 있습니다. 새 호스트는 놀라게 하는 어떤 것도 볼 수 없습니다.

비활성화된 풀 기능은 운영 체제에서 사용할 수 있지만 활성화되어 있지 않습니다. 풀에 이러한 기능을 사용할 수 있다고 표시되어 있지 않습니다. 비활성화된 기능이 있다는 것은 운영 체제에서 해당 기능을 사용할 수 있다는 의미입니다. 이 풀은 이 기능을 지원하지 않는 호스트에서도 확실히 사용할 수 있습니다.

이 기능이 활성화되어 있으면 해당 기능이 사용 중이므로 디스크 형식이 변경된 것입니다. 대부분의 경우 이 기능을 지원하지 않는 시스템에서는 이 풀을 가져올 수 없습니다. 기능이 활성화되어 있지만 해당 기능을 사용하는 모든 데이터 세트가 삭제된 경우 풀은 기능 설정을 활성화로 되돌립니다.

일부 기능은 "read-only compatible"입니다. 해당 기능을 사용 중인 경우, 해당 기능을 지원하지 않는 시스템으로 풀을 부분적으로 가져올 수 있습니다. 새 호스트는 풀에 있는 일부 데이터 집합을 볼 수 없고 풀에 데이터를 쓸 수 없지만, 데이터 집합에서 일부 데이터를 추출할 수는 있습니다.

풀을 만들면 해당 운영 체제의 ZFS 구현에서 지원하는 모든 기능이 활성화됩니다. **zpool create**와 함께 **-d** 플래그를 사용하여 새 풀의 모든 기능을 비활성화한 다음 기능을 보다 선택적으로 활성화할 수 있습니다.

이제 풀의 작동 원리를 이해했으니 실제 데이터를 풀에 입력해 보겠습니다.

챕터 4. ZFS 데이터셋 (ZFS Datasets)

일반 파일시스템에서는 파티션을 만들어 다양한 유형의 데이터를 분리하고, 파티션에 다양한 최적화를 적용하고, 파티션이 사용할 수 있는 공간의 양을 제한할 수 있습니다. 각 파티션은 디스크에서 특정 양의 공간을 할당받습니다. 우리 모두 그런 경험이 있습니다. 우리는 다음 달, 내년, 5년 후 이 시스템의 각 파티션에 얼마나 많은 디스크 공간이 필요할지 추측합니다. 그런데 미래를 내다보면 각 파티션에 할당하기로 결정한 공간의 양이 잘못되었을 가능성이 높습니다. 모든 데이터를 저장할 공간이 충분하지 않은 파티션은 디스크를 추가하거나 데이터를 이동해야 하므로 시스템 관리가 복잡해집니다. 파티션에 공간이 너무 많으면 다른 곳에 두는 것이 더 나을 데이터를 파티션에 버리게 됩니다. 루카스의 UFS2 시스템 중 하나 이상은 */home*의 어딘가에 대한 심볼릭 링크로 */usr/*포트를 가지고 있습니다. Jude는 보통 */usr/local/var*에 */var*의 일부를 저장합니다.

ZFS는 여유 공간을 풀링하여 일반적인 파일 시스템에서는 불가능한 파티션의 유연성을 제공함으로써 이 문제를 해결합니다. 생성하는 각 ZFS 데이터 세트는 그 안에 파일을 저장하는 데 필요한 공간만 사용합니다. 각 데이터 세트는 풀의 모든 여유 공간에 액세스할 수 있으므로 파티션 크기에 대한 걱정을 덜 수 있습니다. 6장에서 설명한 대로 할당량으로 데이터 세트의 크기를 제한하거나 예약을 통해 최소한의 공간을 보장할 수 있습니다.

일반 파일시스템은 별도의 파티션을 사용하여 다양한 유형의 데이터에 대해 서로 다른 정책과 최적화를 설정합니다. */var*에는 로그와 데이터베이스처럼 자주 변경되는 파일이 들어 있습니다. 루트 파일시스템은 성능보다 일관성과 안전성이 중요합니다. */home*에서는 무엇이든 가능합니다. 하지만 기존 파일시스템에 대한 정책을 한 번 설정하면 변경하기가 정말 어렵습니다. UFS용 **tunefs(8)** 유틸리티를 사용하려면 파일시스템을 마운트 해제해야 변경할 수 있습니다. 아이노드 수와 같은 일부 특성은 파일시스템이 생성된 후에는 변경할 수 없습니다.

기존 파일시스템의 핵심 문제는 유연성 부족으로 귀결됩니다. ZFS 데이터 세트는 거의 무한대로 유연합니다.

데이터세트 (Datasets)

데이터 세트는 이름이 지정된 데이터 덩어리입니다. 이 데이터는 파일, 디렉터리, 권한 등 기존 파일 시스템과 비슷할 수 있습니다. 원시 블록 장치, 다른 데이터의 사본 또는 디스크에 넣을 수 있는 모든 것일 수 있습니다.

ZFS는 기존 파일시스템이 파티션을 사용하는 것과 마찬가지로 데이터세트를 사용합니다. */usr*에 대한 정책과 */home*에 대한 별도의 정책이 필요하신가요? 각각 데이터세트를 만들면 됩니다. iSCSI 대상에 대한 차단 장치가 필요하신가요? 바로 데이터 세트입니다. 데이터 세트의 사본을 원하시나요? 그것도 또 다른 데이터 세트입니다.

데이터 세트에는 계층적 관계가 있습니다. 하나의 스토리지 풀이 각 최상위 데이터 세트의 부모입니다. 각 데이터 세트는 하위 데이터 세트를 가질 수 있습니다. 이 장에서 살펴보겠지만 데이터 세트는 부모로부터 많은

특성을 상속받습니다.

모든 데이터 세트 작업은 **zfs(8)** 명령으로 수행합니다. 이 명령에는 다양한 종류의 하위 명령이 있습니다.

데이터세트 유형 (Dataset Types)

현재 ZFS에는 파일시스템 (filesystems), 볼륨 (volumes), 스냅샷 (snapshots), 복제본 (clones), 북마크 (bookmarks) 등 5가지 유형의 데이터 세트가 있습니다.

파일 시스템 데이터 세트는 기존 파일 시스템과 유사합니다. 파일과 디렉터리를 저장합니다. ZFS 파일시스템에는 마운트 지점이 있으며 읽기 전용, setuid 바이너리 제한 등과 같은 기존 파일시스템의 특성을 지원합니다. 파일시스템 데이터세트에는 권한, 파일 생성 및 수정을 위한 타임스탬프, NFSv4 액세스 제어 플래그, chflags(2) 등의 기타 정보도 포함됩니다.

ZFS 볼륨 또는 **zvol**은 블록 장치입니다. 일반 파일 시스템에서는 iSCSI용 파일 백업 파일 시스템이나 특수 목적의 UFS 파티션을 만들 수 있습니다. ZFS에서 이러한 블록 디바이스는 파일과 디렉터리의 모든 오버헤드를 우회하여 기본 풀에 직접 상주합니다. Zvol은 디스크 이미지를 마운트하는 데 사용되는 FreeBSD 메모리 장치를 건너뛰고 장치 노드를 가져옵니다.

스냅샷은 특정 시점의 데이터 세트에 대한 읽기 전용 사본입니다. 스냅샷을 사용하면 나중에 사용할 수 있도록 이전 버전의 파일시스템과 그 안에 있는 파일을 보존할 수 있습니다. 스냅샷은 현재 파일 시스템과 스냅샷에 있는 파일 간의 차이에 따라 일정량의 공간을 사용합니다.

클론은 기존 데이터 세트의 스냅샷을 기반으로 하는 새로운 데이터 세트로, 파일 시스템을 포크할 수 있습니다. 데이터 세트의 모든 항목에 대한 추가 복사본을 얻을 수 있습니다. 프로덕션 웹 사이트가 포함된 데이터세트를 복제하여 프로덕션 사이트를 건드리지 않고도 해킹할 수 있는 사이트 사본을 만들 수 있습니다. 복제본은 생성된 원본 스냅샷과의 차이점을 저장하는 데만 공간을 사용합니다. 7장에서는 스냅샷, 클론 및 북마크에 대해 다룹니다.

데이터 세트가 필요한 이유는 무엇인가요? (Why Do I Want Datasets?)

당연히 데이터 세트가 필요합니다. 디스크에 파일을 저장하려면 파일 시스템 데이터 세트가 필요합니다. 그리고 */usr* 및 */var*와 같은 각 기존 유닉스 파티션에 대한 데이터 세트가 필요할 것입니다. 하지만 ZFS를 사용하면 많은 데이터 세트가 필요합니다. 아주 많은 데이터 세트가 필요합니다. 파티션 수에 대한 하드코딩된 제한과 파티션의 유연성이 없는 기존 파일 시스템에서는 이런 일이 벌어질 수 없습니다. 하지만 많은 데이터세트를 사용하면 데이터에 대한 통제력이 높아집니다.

각 ZFS 데이터 세트에는 작동을 제어하는 일련의 속성이 있어 관리자가 데이터 세트의 작동 방식과 데이터 보호 수준을 제어할 수 있습니다. 기존 파일 시스템에서와 마찬가지로 각 데이터세트를 정확하게 조정할 수 있습니다. 데이터세트 속성은 풀 속성과 매우 유사하게 작동합니다.

시스템 관리자는 개별 데이터 세트에 대한 제어 권한을 다른 사용자에게 위임하고, 해당 사용자가 루트 권한 없이도 데이터 세트를 관리할 수 있도록 할 수 있습니다. 조직에 여러 프로젝트 팀이 있는 경우, 각 프로젝트 관리자에게 자신만의 공간을 주고 "여기, 원하는 대로 정리해 보세요."라고 말할 수 있습니다. 업무량을 줄여주는 것은 무엇이든 좋은 일입니다.

복제 및 스냅샷과 같은 많은 ZFS 기능은 데이터 세트 단위로 작동합니다. 데이터를 논리적 그룹으로 분리하면 조직을 지원하기 위해 이러한 ZFS 기능을 더 쉽게 사용할 수 있습니다.

각각 다른 팀에서 관리하는 수십 개의 사이트가 있는 웹 서버를 예로 들어 보겠습니다. 어떤 팀은 여러 사이트를 담당하는 반면, 어떤 팀은 한 사이트만 담당합니다. 어떤 사람들은 여러 팀에 소속되어 있습니다. 전통적인 파일 시스템 모델을 따른다면 */webserver* 데이터 집합을 만들어 모든 것을 그 안에 넣고 그룹 권한과 **sudo(8)**로 액세스를 제어할 수 있습니다. 수십 년 동안 이런 식으로 살아왔고 잘 작동하는데 왜 바꾸어야 할까요?

하지만 각 팀에 대한 데이터 집합을 만들고, 그 상위 데이터 집합 내에 각 사이트에 고유한 데이터 집합을 부여하면 가능성은 배가됩니다.

팀에서 테스트를 위해 웹 사이트의 사본이 필요하신가요? 복제하세요. 기존 파일 시스템에서는 사이트 디렉터리 전체를 복사해야 하므로 사이트에 필요한 디스크의 양이 두 배로 늘어나고 시간도 훨씬 더 오래 걸립니다. 복제본은 사이트 간의 차이점에 해당하는 공간만 사용하며 즉시 나타납니다.

팀에서 사이트의 새 버전을 배포하려고 하는데 이전 사이트의 백업이 필요하신가요? 스냅샷을 만드세요. 이 새 사이트는 이전 사이트와 동일한 파일을 많이 사용하므로 디스크 공간 사용량을 줄일 수 있습니다. 또한 배포가 심하게 잘못되었을 때 스냅샷으로 롤백하여 이전 버전을 복원할 수 있습니다.

특정 웹 사이트에 파일 시스템 수준의 성능 조정이나 압축 또는 로컬에서 생성된 일부 속성이 필요하신가요? 해당 사이트에 맞게 설정하세요.

각 팀에 대한 데이터 집합을 만든 다음, 각 팀에서 각자의 사이트에 대한 하위 데이터 집합을 만들도록 할 수 있습니다. 기술에 맞게 인력을 구성하는 것이 아니라 인력에 맞게 데이터 집합을 구성할 수 있습니다.

모든 사이트에서 파일 시스템 설정(속성)을 변경해야 하는 경우, 상위 데이터 집합을 변경하고 하위 데이터 집합이 이를 상속하도록 합니다.

사용자 홈 디렉토리에도 동일한 속성이 적용됩니다.

컴퓨터 간에 데이터 세트를 이동할 수도 있습니다. 웹 사이트가 웹 서버를 넘치나요? 데이터 세트의 절반을 사용자 지정 설정, 모든 복제본 및 스냅샷과 함께 새 서버로 전송하세요.

많은 파일시스템 데이터세트를 사용하는 데는 한 가지 단점이 있습니다. 파일시스템 내에서 파일을 이동하면 파일 이름이 변경됩니다. 서로 다른 파일시스템 간에 파일을 이동하려면 이름만 바꾸는 것이 아니라 파일을 새 위치로 복사하고 이전 위치에서 삭제해야 합니다. 데이터 세트 간 파일 복사는 시간이 더 걸리고 더 많은 여유 공간이 필요합니다. 하지만 ZFS가 여러 데이터 세트에 제공하는 모든 이점에 비하면 사소한 문제입니다. 이 문제는 다른 파일시스템에도 존재하지만, 대부분의 다른 파일시스템을 사용하는 호스트는 파티션이 몇 개 밖에 없기 때문에 잘 드러나지 않습니다.

데이터 세트 보기 (Viewing Datasets)

zfs list 명령은 모든 데이터 세트와 그에 대한 몇 가지 기본 정보를 표시합니다.

```
$ zfs list
NAME                 USED  AVAIL  REFER  MOUNTPOINT
mypool               420M  17.9G    96K  none
mypool/ROOT          418M  17.9G    96K  none
mypool/ROOT/default 418M  17.9G   418M  /
...
```

첫 번째 필드에는 데이터 세트의 이름이 표시됩니다 .

USED 및 REFER 아래에서 데이터 집합이 사용하는 디스크 공간의 양에 대한 정보를 확인할 수 있습니다 . ZFS 의 놀라운 유연성과 효율성에 대한 한 가지 단점은 디스크 공간 사용량에 대한 해석을 이해하지 못하면 다소 비현실적으로 보인다는 것입니다 . 6장에서는 디스크 공간과 이를 사용하는 전략에 대해 설명합니다 .

AVAIL 열은 풀 또는 데이터 집합에 남아 있는 여유 공간을 표시합니다 .

마지막으로 마운트포인트는 데이터세트를 마운트해야 하는 위치를 보여줍니다 . 이는 데이터 세트가 마운트된다는 의미는 아니며 , 단지 마운트될 경우 이 위치로 이동한다는 의미일 뿐입니다 . 마운트된 모든 ZFS 파일시스템을 보려면 **zfs mount**를 사용하세요 .

데이터세트를 인수로 제공하면 **zfs list**에 해당 특정 데이터 집합만 표시됩니다 .

```
$ zfs list mypool/lamb
NAME         USED  AVAIL  REFER  MOUNTPOINT
mypool/lamb  192K  17.9G    96K  /lamb
```

-t 플래그와 유형으로 표시되는 데이터세트의 유형을 제한합니다 . 파일 시스템 , 볼륨 또는 스냅샷을 표시할 수 있습니다 . 여기서는 스냅샷만 표시합니다 .

```
$ zfs list -t snapshot
NAME                    USED  AVAIL  REFER  MOUNTPOINT
zroot/var/log/db@backup    0      -  10.0G  -
```

이제 파일시스템을 볼 수 있게 되었으니 직접 만들어 보겠습니다 .

데이터세트 생성 , 이동 및 삭제하기 (Creating, Moving, and Destroying Datasets)

zfs create 명령을 사용해 데이터세트를 생성합니다 . 스냅샷 , 복제본 , 북마크는 7장에서 살펴보겠지만 , 지금은 파일시스템과 볼륨에 대해 알아보겠습니다 .

파일시스템 만들기 (Creating Filesystems)

파일시스템은 대부분의 시스템에서 가장 일반적인 데이터세트 유형입니다 . 누구나 파일을 저장하고 정리할 공간이 필요합니다 . 풀과 파일 시스템 이름을 지정하여 파일 시스템 데이터집합을 만듭니다 .

```
$ zfs create mypool/lamb
```

이렇게 하면 mypool이라는 ZFS 풀에 새 데이터 세트인 lamb이 생성됩니다 . 풀에 기본 마운트 지점이 있는 경우 새 데이터 세트가 기본적으로 마운트됩니다 (이 장 뒷부분의 "ZFS 파일 시스템 마운트 " 참조).

```
$ mount | grep lamb
mypool/lamb on /lamb (zfs, local, noatime, nfsv4acls)
```

괄호 안의 마운트 설정은 일반적으로 부모 데이터세트에서 상속된 ZFS 속성입니다 . 하위 파일시스템을 만들려면 상위 파일시스템의 전체 경로를 입력합니다 .

```
$ zfs create mypool/lamb/baby
```

이 장의 뒷부분에 나오는 '부모 /자식 관계 '에서 살펴보겠지만 , 데이터세트는 마운트 지점을 비롯한 많은 특성을 부모로부터 상속받습니다 .

볼륨 만들기 (Creating Volumes)

-V 플래그와 볼륨 크기를 사용하여 볼륨을 만들려는 볼륨을 **zfs create**에 알려줍니다 . 볼륨 데이터세트의 전체 경로를 입력합니다 .

```
$ zfs create -V 4G mypool/avolume
```

Zvols는 다른 데이터세트와 마찬가지로 데이터세트 목록에 표시됩니다 . **-t volume** 옵션을 추가하여 **zfs list**에 zvol만 표시하도록 할 수 있습니다 .

```
$ zfs list mypool/avolume
NAME            USED  AVAIL  REFER  MOUNTPOINT
ypool/avolume  4.13G  17.9G    64K  -
```

Z볼은 볼륨의 크기와 ZFS 메타데이터를 위한 만큼의 공간을 자동으로 예약합니다 . 이 4GB zvol은 4.13GB의 공간을 사용합니다 .

블록 디바이스로서 zvol에는 마운트 지점이 없습니다 . 하지만 */dev/zvol* 아래에 디바이스 노드가 있으므로 다른 블록 디바이스와 마찬가지로 액세스할 수 있습니다 .

```
$ ls -al /dev/zvol/mypool/avolume
crw-r-----  1 root  operator  0x4d Mar 27 20:22 /dev/zvol/mypool/avolume
```

이 디바이스 노드에서 **newfs(8)**를 실행하고 디스크 이미지를 복사한 후 일반적으로 다른 블록 디바이스처럼 사용할 수 있습니다 .

데이터세트 이름 변경 (Renaming Datasets)

데이터세트의 이름을 바꾸려면 이상하게도 **zfs rename** 명령을 사용하면 됩니다 . 데이터세트의 현재 이름을 첫 번째 인수로 지정하고 새 위치를 두 번째 인수로 지정합니다 .

```
$ zfs rename db/production db/old
$ zfs rename db/testing db/production
```

데이터 세트의 이름을 강제로 바꾸려면 **-f** 플래그를 사용합니다 . 프로세스가 실행 중인 파일시스템은 마운트 해제할 수 없지만 **-f** 플래그를 사용하면 강제로 마운트 해제할 수 있습니다 . 데이터세트를 사용 중인 모든 프로세스는 사용 중이던 데이터에 대한 액세스 권한을 잃고 사용자가 원하는 대로 반응합니다 .

> 아마 심할 겁니다 .

데이터세트 이동하기 (Moving Datasets)

데이터세트를 ZFS 트리의 일부에서 다른 부분으로 이동하여 데이터세트를 새 부모의 하위 집합으로 만들 수 있습니다 . 자식은 부모로부터 속성을 상속하므로 데이터세트의 많은 속성이 변경될 수 있습니다 . 데이터세트에 특별히 설정된 속성은 변경되지 않습니다 .

여기서는 내결함성을 개선하기 위해 몇 가지 속성을 설정한 새 상위 데이터세트인 *zroot/var/db* 데이터세트 아래에서 데이터베이스를 이동합니다 .

```
$ zfs rename zroot/var/db/mysql zroot/important/mysql
```

마운트 지점은 상속되므로 , 이렇게 하면 데이터 집합의 마운트 지점이 변경될 수 있습니다 . **rename** 명령에 **-u** 플래그를 추가하면 ZFS가 마운트 지점을 즉시 변경하지 않으므로 프로퍼티를 의도한 값으로 재설정할 시간을 벌 수 있습니다 . 컴퓨터를 다시 시작하거나 데이터세트를 수동으로 다시 마운트하면 새 마운트 지점을 사용한다는 점을 기억하세요 .

스냅샷의 이름을 변경할 수는 있지만 상위 데이터 세트에서 스냅샷을 이동할 수는 없습니다 . 스냅샷은 7장에서 자세히 다룹니다 .

데이터세트 삭제하기 (Destroying Datasets)

데이터 세트가 지겨우신가요 ? 헛간 뒤로 끌어다 놓고 **zfs destroy**를 통해 고통에서 벗어나세요 .

```
$ zfs destroy db/old
```

-r 플래그를 추가하면 데이터세트의 모든 자식 (데이터 세트 , 스냅샷 등)을 재귀적으로 파기합니다 . 복제된 데이터세트를 모두 파기하려면 **-R**을 사용합니다 . 데이터세트의 자식이 정확히 무엇인지 알 수 없는 경우가 종종 있으므로 데이터세트를 재귀적으로 파기할 때는 매우 주의하세요 .

데이터세트를 파기할 때 어떤 일이 발생하는지 정확히 확인하려면 **-v** 및 **-n** 플래그를 사용할 수 있습니다 . **-v** 플래그는 소멸되는 항목에 대한 자세한 정보를 출력하고 , **-n**은 **zfs(8)**에 드라이런을 수행하도록 지시합니다 . 이 두 플래그는 트리거를 실행하기 전에 이 명령이 실제로 무엇을 파괴하는지 보여줍니다 .

ZFS 속성 (ZFS Properties)

ZFS 데이터세트에는 데이터세트 작동 방식을 제어하는 속성이라고 하는 여러 가지 설정이 있습니다 . 이 중 몇 가지는 데이터세트를 만들 때만 설정할 수 있지만 , 대부분은 데이터세트가 라이브 상태일 때 조정할 수 있습니다 . ZFS는 또한 데이터세트가 사용하는 공간의 양 , 압축 또는 중복 제거 비율 , 데이터세트의 생성 시간 등의 정보를 제공하는 여러 읽기 전용 속성을 제공합니다 .

각 데이터세트는 해당 데이터세트에 속성이 특별히 설정되어 있지 않는 한 부모로부터 속성을 상속받습니다 .

속성 보기 (Viewing Properties)

zfs(8) 도구는 특정 속성 또는 데이터 세트의 모든 속성을 검색할 수 있습니다 . **zfs get** 명령과 원하는 속성 , 원하는 경우 데이터 세트 이름을 사용합니다 .

```
$ zfs get compression mypool/lamb
NAME         PROPERTY     VALUE    SOURCE
mypool/lamb  compression  lz4      inherited from mypool
```

NAME 아래에는 요청한 데이터 집합이 표시되고 , PROPERTY에는 요청한 속성이 표시됩니다 . VALUE는 속성이 설정된 값입니다 .

SOURCE는 조금 더 복잡합니다 . 기본 소스는 이 속성이 ZFS의 기본값으로 설정되어 있음을 의미합니다 . 로컬 소스는 누군가가 이 데이터세트에 이 속성을 의도적으로 설정했음을 의미합니다 . 임시 속성은 데이터세트가 마운트될 때 설정되었으며 , 데이터세트가 마운트 해제되면 이 속성은 일반적인 값으로 되돌아갑니다 . 상속된 속성은 이 장의 뒷부분에 있는 "상위 /하위 관계 "에서 설명하는 대로 상위 데이터세트에서 가져옵니다 .

일부 속성은 소스가 관련이 없거나 본질적으로 명백하기 때문에 소스가 없습니다 . 데이터세트가 생성된 날짜와 시간을 기록하는 생성 속성에는 소스가 없습니다 . 이 값은 시스템 시계에서 가져온 것입니다 .

데이터세트 이름을 지정하지 않으면 **zfs get**은 모든 데이터세트에 대해 이 속성의 값을 표시합니다 . 특수 속성 키워드는 모두 데이터세트의 모든 속성을 검색합니다 .

```
$ zfs get all mypool/lamb
NAME         PROPERTY   VALUE                  SOURCE
mypool/lamb  type       filesystem             -
mypool/lamb  creation   Fri Mar 27 20:05 2015  -
mypool/lamb  used       192K                   -
...
```

all를 사용하고 데이터세트 이름을 지정하지 않으면 모든 데이터세트에 대한 모든 속성을 가져옵니다 . 이것은 많은 정보입니다 . 속성 이름을 쉼표로 구분하여 여러 속성을 표시합니다 .

```
$ zfs get quota,reservation zroot/home
NAME        PROPERTY     VALUE   SOURCE
zroot/home  quota        none    local
zroot/home  reservation  none    default
```

zfs list **-o** .

```
$ zfs list -o name,quota,reservation
NAME              QUOTA  RESERV
db                none    none
zroot             none    none
zroot/ROOT        none    none
zroot/ROOT/default  none    none
...
zroot/var/log      100G    20G
...
```

(Changing Properties)

zfs set . , , .

compression **off** .

```
$ zfs set compression=off mypool/lamb/baby
```

zfs get .

```
$ zfs get compression mypool/lamb/baby
NAME            PROPERTY     VALUE  SOURCE
mypool/lamb/baby  compression  off    local
```

. **compression** ZFS . 6 . zfs .

(Read-Only Properties)

ZFS는 읽기 전용 속성을 사용하여 데이터 집합에 대한 기본 정보를 제공합니다. 디스크 공간 사용량은 속성으로 표현됩니다. "디스크가 반쯤 찼습니다."라는 속성을 변경하여 사용 중인 데이터의 양을 변경할 수는 없습니다. (6장에서는 ZFS 디스크 공간 사용량에 대해 다룹니다.) **creation** 속성은 이 데이터 집합이 언제 만들어졌는지 기록합니다. 디스크에 데이터를 추가하거나 제거하여 많은 읽기 전용 속성을 변경할 수 있지만 이러한 속성을 직접 쓸 수는 없습니다.

파일 시스템 속성 (Filesystem Properties)

기존 파일시스템의 성능과 동작을 관리하기 위한 주요 도구 중 하나는 마운트 옵션입니다. 기존 파일시스템을 읽기 전용으로 마운트하거나 **noexec** 플래그를 사용하여 파일시스템에서 실행 중인 프로그램을 비활성화할 수 있습니다. ZFS는 속성을 사용하여 동일한 효과를 얻을 수 있습니다. 다음은 이러한 익숙한 목표를 달성하는 데 사용되는 속성입니다.

atime

파일의 atime은 파일이 마지막으로 액세스된 시간을 나타냅니다. ZFS의 **atime** 속성은 데이터 세트가 액세스 시간을 추적할지 여부를 제어합니다. 기본값인 on은 파일에 액세스할 때마다 파일의 atime 메타데이터를 업데이트합니다. **atime**을 사용한다는 것은 디스크를 읽을 때마다 디스크에 기록한다는 의미입니다.

이 속성을 끄면 (off) 파일을 읽을 때 디스크에 쓰기가 방지되므로 성능이 크게 향상될 수 있습니다. 파일을 마지막으로 읽은 시점을 파악하는 데 의존하는 메일러 및 기타 유사한 유틸리티와 혼동을 일으킬 수 있습니다.

atime을 그대로 두면 스냅샷 크기가 커집니다. 파일에 처음 액세스할 때 해당 파일의 타임이 업데이트됩니다. 스냅샷에는 원래 액세스 시간이 유지되고, 라이브 파일시스템에는 새로 업데이트된 액세스 시간이 포함됩니다. 이것이 기본값입니다.

exec

exec 속성은 누구나 이 파일 시스템에서 바이너리와 명령을 실행할 수 있는지 여부를 결정합니다. 기본값은 실행을 허용하는 **on**입니다. 일부 환경에서는 사용자가 개인 또는 임시 디렉터리에서 프로그램을 실행하는 것을 허용하지 않습니다. 파일 시스템에서 프로그램을 실행할 수 없도록 하려면 **exec** 속성을 **off**로 설정하세요.

그러나 **exec** 속성은 사용자가 해석된 스크립트를 실행하는 것을 금지하지 않습니다. 사용자가 */bin/sh*를 실행할 수 있다면 */bin/sh /home/mydir/script.sh*를 실행할 수 있습니다. 셸은 실제로 실행되는 것으로, 스크립트의 명령만 받습니다.

readonly

이 데이터 세트에 아무 것도 쓰지 못하게 하려면 **readonly** 속성을 **on**으로 설정하세요. 기본값인 **off**로 설정하면 사용자가 관리 권한 내에서 데이터 집합을 수정할 수 있습니다.

setuid

많은 사람들이 setuid 프로그램을 위험하다고 생각합니다. **passwd(1), login(1)** 등 일부 setuid 프로그램은 반드시 setuid여야 하지만, */home* 및 */tmp*와 같은 파일시스템에 setuid 프로그램을 설치할

필요는 거의 없습니다. 대부분의 시스템 관리자는 특정 파일시스템을 제외하고는 setuid 프로그램을 허용하지 않습니다.

> 올바르게 작성된 setuid 프로그램은 위험하지 않습니다. 그렇기 때문에 실제 setuid 프로그램은 위험합니다.

ZFS의 **setuid** 속성은 setuid 지원을 토글합니다. **on**으로 설정하면 파일시스템이 setuid를 지원합니다. **off**로 설정하면 setuid 플래그가 무시됩니다.

사용자 정의 속성 (User-Defined Properties)

ZFS 속성은 정말 훌륭하고 충분히 사용할 수 있지 않나요? 이제 직접 추가해 보세요. 데이터 세트와 함께 자신만의 메타데이터를 저장할 수 있는 기능을 통해 완전히 새로운 자동화 영역을 개발할 수 있습니다. 자식들이 이러한 속성을 자동으로 상속한다는 사실은 삶을 훨씬 더 쉽게 만들어 줍니다.

사용자 정의 속성을 나만의 것으로 유지하고 다른 사람의 사용자 정의 속성과 충돌하지 않게 하려면 네임스페이스를 만드세요. 대부분의 사람들은 사용자 정의 속성 앞에 조직 식별자와 콜론을 붙입니다. 예를 들어 FreeBSD 관련 프로퍼티는 "org.freebsd:속성명" 형식 (예: **org.freebsd:swap**)을 갖습니다. 일루미오스 프로젝트에서 스왑이라는 이름의 자체 프로퍼티를 생성하는 경우, 이 프로퍼티를 **org.illumos:swap**이라고 부릅니다. 두 값은 충돌하지 않습니다.

예를 들어, Jude가 데이터세트 속성을 통해 어떤 데이터세트를 백업할지 제어하고 싶다고 가정해 보겠습니다. 그는 네임스페이스 **com.allanjude**를 생성하고, 이 네임스페이스 내에 **backup_ignore** 속성을 생성합니다.

> 자신의 이름을 딴 ZFS 속성의 이름을 지정하면 자신의 작업이 영원히 남게 됩니다. 이것이 좋은지 나쁜지는 여러분의 작업에 달려 있습니다.

```
$ zfs set com.allanjude:backup_ignore=on mypool/lamb
```

Jude의 백업 스크립트는 이 속성 값을 확인합니다. true로 설정되어 있으면 백업 프로세스에서 이 데이터 세트를 건너뜁니다.

부모/자식 관계 (Parent/Child Relationships)

데이터세트는 상위 데이터세트에서 속성을 상속합니다. 데이터세트에서 속성을 설정하면 해당 속성은 해당 데이터세트와 모든 하위 데이터세트에 적용됩니다. 편의를 위해 **-r** 플래그를 추가하여 데이터세트와 그 모든 하위 데이터세트에 대해 **zfs(8)** 명령을 실행할 수 있습니다. 여기서는 데이터세트와 모든 하위 데이터세트의 압축 속성을 쿼리합니다.

```
$ zfs get -r compression mypool/lamb
NAME           PROPERTY     VALUE  SOURCE
mypool/lamb    compression  lz4    inherited from mypool
```

```
mypool/lamb/baby  compression  off    local
```

소스 값을 살펴봅니다. 첫 번째 데이터 집합인 *mypool/lamb*은 상위 풀에서 이 속성을 상속받았습니다. 두 번째 데이터 집합에서 이 속성은 다른 값을 갖습니다. 소스가 로컬이므로 이 속성이 이 데이터 집합에 특별히 설정되었음을 의미합니다.

zfs inherit 명령으로 원래 설정을 복원할 수 있습니다.

```
$ zfs inherit compression mypool/lamb/baby
$ zfs get -r compression mypool/lamb
NAME              PROPERTY     VALUE SOURCE
mypool/lamb       compression  lz4   inherited from mypool
mypool/lamb/baby  compression  lz4   inherited from mypool
```

이제 자식은 부모로부터 **compression** 속성을 상속받으며, 부모는 조부모로부터 **compression** 속성을 상속받습니다.

부모의 속성을 변경하면 새 속성이 자동으로 자식에게 전파됩니다.

```
$ zfs set compression=gzip-9 mypool/lamb
$ zfs get -r compression mypool/lamb
NAME              PROPERTY     VALUE  SOURCE
mypool/lamb       compression  gzip-9 local
mypool/lamb/baby  compression  gzip-9 inherited from mypool/lamb
```

부모 데이터 집합에 *gzip-9* 압축을 사용하도록 지시했습니다. 이는 자식에게까지 영향을 미쳤습니다.

상속 및 이름 변경 (Inheritance and Renaming)

데이터세트를 이동하거나 이름을 변경하여 새 부모를 갖게 되면 부모의 속성이 자동으로 자식에게 전파됩니다. 로컬로 설정된 속성은 변경되지 않지만 상속된 속성은 새 부모의 속성으로 전환됩니다.

여기서는 새 상위 데이터세트를 만들고 압축 속성을 확인합니다.

```
$ zfs create mypool/second
$ zfs get compress mypool/second
NAME           PROPERTY     VALUE SOURCE
mypool/second  compression  lz4   inherited from mypool
```

baby 데이터세트는 *gzip-9* 압축을 사용합니다. 이 속성은 *mypool/lamb*에서 상속받았습니다. 이제 *baby*를 second의 자식으로 이동하고 압축 속성이 어떻게 되는지 살펴봅시다.

```
$ zfs rename mypool/lamb/baby mypool/second/baby
$ zfs get -r compression mypool/second
NAME                 PROPERTY     VALUE  SOURCE
mypool/second        compression  lz4    inherited from mypool
mypool/second/baby   compression  lz4    inherited from mypool
```

이제 자식 데이터 집합은 다른 부모에 속하며 새 부모로부터 해당 속성을 상속받습니다. 자식은 모든 로컬 속성을 유지합니다.

그러나 아기 데이터 세트의 데이터는 약간 엉켜 있습니다. 압축을 켜기 전에 작성된 데이터는 압축되지 않습니다. 데이터 세트가 gzip-9 압축을 사용하는 동안 작성된 데이터는 gzip-9로 압축됩니다. 지금 작성된 모든 데이터는 lz4로 압축됩니다. ZFS는 이 모든 것을 자동으로 분류해 주지만, 생각만 해도 머리가 아픕니다.

속성 제거 (Removing Properties)

속성을 기본값으로 다시 설정할 수는 있지만, 소스를 상속 또는 기본값으로 다시 변경하는 방법이나 사용자 정의 속성을 설정한 후 제거하는 방법은 명확하지 않습니다.

사용자 정의 속성을 제거하려면 상속하세요.

```
$ zfs inherit com.allanjude:backup_ignore mypool/lamb
```

이는 루트 데이터세트에 속성을 설정한 경우에도 작동합니다.

데이터세트와 모든 하위 데이터세트에서 속성을 기본값으로 재설정하거나 사용자 정의 속성을 완전히 제거하려면 풀의 루트 데이터세트에서 **zfs inherit** 명령을 사용합니다.

```
$ zfs inherit -r compression mypool
```

직관적이지 않지만 루트 데이터세트에서 사용자 정의 설정을 제거합니다.

ZFS 파일시스템 마운트하기 (Mounting ZFS Filesystems)

기존 파일시스템에서는 각 파티션과 그 유형, 마운트 위치를 */etc/fstab*에 나열했습니다. 편의를 위해 플로피나 CD-ROM 드라이브와 같은 임시 마운트도 나열했습니다. ZFS를 사용하면 너무 많은 수의 파일시스템을 생성할 수 있어 금방 비실용적이 됩니다.

각 ZFS 파일시스템에는 마운트할 위치를 정의하는 **mountpoint** 속성이 있습니다. 기본 **mountpoint**는 풀의 **mountpoint**에서 빌드됩니다. 풀에 마운트 포인트가 없는 경우, 마운트하려는 데이터 세트에 마운트 포인트를 할당해야 합니다.

```
$ zfs get mountpoint zroot/usr/home
NAME            PROPERTY    VALUE      SOURCE
zroot/usr/home  mountpoint  /usr/home  inherited from zroot/usr
```

파일시스템은 일반적으로 */usr/home*에 마운트됩니다 . 파일시스템을 수동으로 마운트할 때 이 위치를 재정의할 수 있습니다 .

기본 FreeBSD 설치에 사용되는 *zroot* 풀에는 마운트 지점이 설정되어 있지 않습니다 . *zroot* 바로 아래에 새 데이터세트를 생성하는 경우 , 해당 데이터세트에는 마운트 지점이 없습니다 . 예를 들어 */usr* 아래에 있는 *zroot*에서 만든 데이터 세트는 부모 데이터 세트에서 마운트 지점을 상속합니다 .

루트 파일시스템이 있는 풀 이외의 모든 풀은 일반적으로 풀의 이름을 딴 마운트 지점을 갖습니다 . *db*라는 이름의 풀을 만들면 */db*에 마운트됩니다 . 모든 자식은 변경하지 않는 한 해당 풀에서 마운트 지점을 상속받습니다 .

파일시스템의 **mountpoint** 속성을 변경하면 파일시스템과 마운트 지점을 상속하는 모든 하위 파일시스템이 마운트 해제됩니다 . 새 값이 레거시 값이면 마운트 해제된 상태로 유지됩니다 . 그렇지 않고 이전에 *legacy* 속성이었거나 속성이 *none*이거나 속성이 변경되기 전에 마운트된 경우 새 위치에 자동으로 다시 마운트됩니다 . 또한 모든 공유 파일시스템은 공유가 해제되고 새 위치에서 공유됩니다 .

일반 파일시스템과 마찬가지로 ZFS 파일시스템도 반드시 마운트할 필요는 없습니다 . **canmount** 속성은 파일시스템의 마운트 동작을 제어합니다 . **canmount**가 **yes**로 설정되어 있으면 , 마운트 **-a**와 마찬가지로 **zfs mount -a**를 실행하면 파일시스템이 마운트됩니다 . *etc/rc.conf*에서 ZFS를 활성화하면 FreeBSD는 시작할 때 **zfs mount -a**를 실행합니다 .

canmount 속성이 **noauto**로 설정된 경우 , 데이터세트는 명시적으로만 마운트 및 마운트 해제할 수 있습니다 . 데이터 집합을 만들거나 가져올 때 데이터세트가 자동으로 마운트되지 않으며 , **zfs mount -a** 명령으로 마운트하거나 **zfs unmount -a**로 마운트 해제하지도 않습니다 .

canmount를 **off**으로 설정하면 상황이 흥미로워질 수 있습니다 . 마운트 지점이 같은 마운트 불가능한 데이터 세트가 두 개 있을 수 있습니다 . 아래에서 보듯이 데이터 세트는 향후 데이터 세트의 부모가 되기 위한 목적으로만 존재할 수 있지만 실제로 파일을 저장하지는 않습니다 .

하위 데이터 세트는 **canmount** 속성을 상속하지 않습니다 .

canmount 속성을 변경해도 파일시스템이 자동으로 마운트 해제되거나 마운트되지 않습니다 . 마운트된 파일시스템에서 마운트를 비활성화하면 파일시스템을 수동으로 마운트 해제하거나 재부팅해야 합니다 .

마운트 포인트가 없는 데이터세트 (Datasets without Mount Points)

ZFS 데이터 세트는 계층 구조입니다 . 파일을 전혀 포함하지 않는 데이터 집합을 만들어 다른 여러 데이터 집합의 공통 부모가 될 수 있도록 해야 할 수도 있습니다 . FreeBSD 10.1 이상을 기본 설치한다고 가정해 보겠습니다 .

```
$ zfs mount
zroot/ROOT/default   /
zroot/tmp            /tmp
zroot/usr/home       /usr/home
zroot/usr/ports      /usr/ports z
root/usr/src         /usr/src
```

```
...
```

/usr 아래에 모든 종류의 데이터 세트가 있지만 */usr* 데이터 세트가 마운트되어 있지 않습니다. 무슨 일인가요?

zfs list는 데이터세트가 존재하고 */usr*의 마운트 지점을 가지고 있음을 보여줍니다. 하지만 *zroot/usr*과 그 모든 자식들의 마운트 포인트와 **canmount** 속성을 확인해 보겠습니다.

```
$ zfs list -o name,canmount,mountpoint -r zroot/usr
NAME          CANMOUNT  MOUNTPOINT
zroot/usr          off  /usr
zroot/usr/home     on   /usr/home
zroot/usr/ports    on   /usr/ports
zroot/usr/src      on   /usr/src
```

canmount를 *off*으로 설정하면 *zroot/usr* 데이터 세트는 마운트되지 않습니다. *usr/bin*의 명령어와 */usr/local*의 패키지 등 */usr*에 작성된 모든 파일은 루트 파일시스템으로 이동합니다. *usr/src*와 같은 하위 레벨 마운트 지점에는 자체 데이터세트가 있으며, 이 데이터세트가 마운트됩니다.

데이터세트는 하위 데이터세트의 부모로만 존재합니다. */var* 파티션에서도 비슷한 것을 볼 수 있습니다.

동일한 마운트 지점을 가진 여러 데이터세트 (Multiple Datasets with the Same Mount Point)

canmount를 *off*로 설정하면 데이터 집합을 속성 상속을 위한 메커니즘으로만 사용할 수 있습니다.
canmount를 *off*로 설정하는 한 가지 이유는 마운트 지점이 같은 두 데이터 집합을 사용하여 두 데이터 집합의 하위 집합이 동일한 디렉터리에 표시되지만 상속된 특성이 다를 수 있도록 하기 위해서입니다.

FreeBSD의 설치 관리자에는 기본 풀인 zroot에 마운트 지점이 없습니다. 새 데이터세트를 만들 때는 마운트 포인트를 할당해야 합니다.

풀 바로 아래에 생성하는 모든 데이터세트에 **mountpoint**를 할당하고 싶지 않다면, *zroot* 풀에 */*의 마운트 포인트를 할당하고 **canmount**를 *off*로 설정할 수 있습니다. 이렇게 하면 새 데이터세트를 만들 때 상속할 **mountpoint**를 갖게 됩니다. 이것은 동일한 마운트 포인트를 가진 여러 데이터세트를 사용하는 매우 간단한 예입니다.

두 개의 하위 디렉터리 세트가 있는 */opt* 디렉터리가 있다고 가정해 보겠습니다. 이러한 디렉터리 중 일부는 프로그램이 포함되어 있으므로 설치 후에는 절대로 쓰면 안 됩니다. 다른 디렉터리에는 데이터가 들어 있습니다. 파일 시스템 수준에서 프로그램을 실행하는 기능을 잠가야 합니다.

```
$ zfs create db/programs # zfs create db/data
```

이제 이 두 데이터세트에 모두 */opt*의 마운트 지점을 지정하고 마운트할 수 없음을 알려줍니다.

```
$ zfs set canmount=off db/programs
$ zfs set mountpoint=/opt db/programs
```

데이터세트에 프로그램을 설치한 다음 읽기 전용으로 설정합니다 .

```
$ zfs set readonly=on db/programs
```

db/data 데이터세트에서는 프로그램을 실행할 수 없으므로 **exec** 및 **setuid**를 해제하세요 . 하지만 이 디렉터리에 데이터를 써야 합니다 .

```
$ zfs set canmount=off db/data
$ zfs set mountpoint=/opt db/data
$ zfs set setuid=off db/data
$ zfs set exec=off db/data
```

이제 몇 가지 자식 데이터세트를 만듭니다 . *db/programs* 데이터세트의 하위 데이터 세트는 해당 데이터 세트의 속성을 상속하고 , *db/data* 데이터 세트의 하위 데이터세트는 다른 속성 집합을 상속합니다 .

```
$ zfs create db/programs/bin
$ zfs create db/programs/sbin
$ zfs create db/data/test
$ zfs create db/data/production
```

이제 */opt*에 바이너리용 2개와 데이터용 2개 , 총 4개의 데이터세트가 마운트되어 있습니다 . 사용자가 알다시피 이 디렉터리는 정상적인 디렉터리입니다 . 하지만 파일 권한이 어떻게 되어 있든 이 디렉터리 중 두 개는 아무도 쓸 수 없습니다 . 사람들이 어떤 속임수를 쓰더라도 시스템은 나머지 두 디렉터리에 있는 실행 파일과 setuid 파일을 인식하지 못합니다 . 데이터나 프로그램을 위해 다른 데이터세트가 필요한 경우 , 원하는 설정으로 데이터세트의 하위 데이터세트로 생성하세요 . 상위 데이터세트에 대한 변경 사항은 모든 하위 데이터세트에 즉시 전파됩니다 .

마운트 포인트가 없는 풀 (Pools without Mount Points)

풀은 일반적으로 풀의 이름을 딴 디렉터리에 마운트되지만 , 반드시 그런 것은 아닙니다 .

```
$ zfs set mountpoint=none mypool
```

이 풀은 더 이상 마운트되지 않습니다 . 마운트 지점을 지정하지 않으면 풀에 있는 데이터 세트도 마운트되지 않습니다 . 이것이 FreeBSD 설치 관리자가 OS용 풀을 생성하는 방식입니다 .

```
$ zfs set mountpoint=/someplace mypool/lamb
```

필요한 경우 디렉터리가 생성되고 파일 시스템이 마운트됩니다 .

수동으로 파일 시스템 마운트 및 마운트 해제하기 (Manually Mounting and Unmounting Filesystems)

파일시스템을 수동으로 마운트하려면 **zfs mount**와 데이터세트 이름을 사용합니다. 이 방법은 **canmount**가 *noauto*으로 설정된 파일시스템에 가장 일반적으로 사용됩니다.

```
$ zfs mount mypool/usr/src
```

파일시스템과 그 하위 파일시스템을 모두 마운트 해제하려면 **zfs unmount**를 사용합니다.

```
$ zfs unmount mypool/second
```

데이터 세트를 다른 위치에 임시로 마운트하려면 **-o** 플래그를 사용하여 새 마운트 지점을 지정합니다. 이 마운트 지점은 데이터 세트를 마운트 해제할 때까지만 지속됩니다.

```
$ zfs mount -o mountpoint=/mnt mypool/lamb
```

데이터 세트에 **mountpoint**가 정의되어 있는 경우에만 데이터세트를 마운트할 수 있습니다. 데이터세트에 마운트 지점이 없는데 임시 마운트 지점을 정의하면 오류가 발생합니다.

ZFS 와 /etc/fstab (ZFS and /etc/fstab)

원하는 경우 */etc/fstab*으로 ZFS 파일시스템 마운트 지점의 일부 또는 전부를 관리하도록 선택할 수 있습니다. 데이터 집합의 **mountpoint** 속성을 *legacy*로 설정합니다. 그러면 파일시스템이 마운트 해제됩니다.

```
$ zfs set mountpoint=legacy mypool/second
```

이제 **mount(8)** 명령으로 이 데이터세트를 마운트할 수 있습니다:

```
$ mount -t zfs mypool/second /tmp/second
```

시스템의 */etc/fstab*에 ZFS 데이터세트를 추가할 수도 있습니다. 전체 데이터세트 이름을 장치 노드로 사용합니다. 유형을 *zfs*로 설정합니다. 표준 파일 시스템 옵션인 *noatime, noexec, readonly* 또는 *ro, nosuid*를 사용할 수 있습니다. (명시적으로 *atime, exec, rw, suid*의 기본 동작을 지정할 수도 있지만 이는 ZFS의 기본값입니다). 마운트 순서는 정상이지만 fsck 필드는 무시됩니다. 다음은 */tmp*에 데이터세트 *scratch/junk* nosuid를 마운트하는 */etc/fstab* 항목입니다.

```
scratch/junk /tmp nosuid 2 0
```

그러나 마운트를 관리할 때는 ZFS 속성을 사용하는 것이 좋습니다. 속성은 */etc/fstab*이 하는 거의 모든 작업을 수행할 수 있습니다.

ZFS 볼륨 조정 (Tweaking ZFS Volumes)

Zvol은 매우 간단합니다. 공간 덩어리를 블록 디바이스로 설정하고 이를 사용하면 됩니다. 볼륨이 공간을 사용하는 방식과 제공하는 디바이스 노드의 종류를 조정할 수 있습니다.

공간 확보 (Space Reservations)

zvol의 **volsize** 속성은 볼륨의 논리적 크기를 지정합니다. 기본적으로 볼륨을 생성하면 볼륨 크기와 동일한 양의 데이터세트 공간이 예약됩니다. (6장 앞부분에서 보면 동일한 크기의 예약이 설정되어 있습니다.). **volsize**를 변경하면 예약이 변경됩니다. volsize는 **volblocksize** 속성의 배수로만 설정할 수 있으며 0은 될 수 없습니다.

예약을 하지 않으면 볼륨 사용 방식에 따라 볼륨의 공간이 부족하여 정의되지 않은 동작이나 데이터 손상이 발생할 수 있습니다. 이러한 영향은 사용 중에 볼륨 크기를 변경할 때, 특히 크기를 축소할 때 발생할 수 있습니다. 볼륨 크기를 조정하면 차단 장치를 사용하는 애플리케이션에 혼란을 줄 수 있습니다.

Zvol은 씬 프로비저닝 *(thin provisioning)*이라고도 하는 스파스 볼륨 *(sparse volumes)*도 지원합니다. 스파스 볼륨은 예약이 볼륨 크기보다 작은 볼륨입니다. 기본적으로 스파스 볼륨을 사용하면 데이터세트가 사용할 수 있는 공간보다 더 많은 공간을 할당할 수 있습니다. 예를 들어, 스파스 프로비저닝을 사용하면 5TB 데이터 세트에 1TB 스파스 볼륨 10개를 생성할 수 있습니다. 볼륨을 많이 사용하지 않는 한, 아무도 볼륨이 과도하게 커밋되었다는 사실을 알아차리지 못합니다.

스파스 볼륨은 권장하지 않습니다. 희소 볼륨에 대한 쓰기는 볼륨 자체가 부분적으로만 가득 차 있어도 '공간 부족' 오류와 함께 실패할 수 있습니다.

zfs create -V 명령에 **-s** 옵션을 지정하여 생성 시 스파스 볼륨을 지정합니다. 볼륨 크기 변경은 예약에 반영되지 않습니다. 볼륨이 생성된 후 예약을 줄일 수도 있습니다.

Zvol 모드 (Zvol Mode)

FreeBSD는 일반적으로 운영체제에 zvol을 **geom(4)** 공급자로 노출하여 최대한의 유연성을 제공합니다. **volmode** 속성으로 이를 변경할 수 있습니다.

볼륨의 **volmode**를 *dev*로 설정하면 볼륨이 */dev*에 문자 장치로만 노출됩니다. 이러한 볼륨은 원시 디스크 장치 파일로만 액세스할 수 있습니다. 파티션하거나 마운트할 수 없으며, RAID 또는 기타 GEOM 기능에 참여할 수 없습니다. 더 빠릅니다. 볼륨을 사용하는 장치를 신뢰할 수 없는 경우에는 개발 모드가 더 안전할 수 있습니다.

volmode를 *none*으로 설정하면 볼륨이 ZFS 외부에 노출되지 않습니다. 그러나 이러한 볼륨은 스냅샷, 복제 및 복제가 가능합니다. 이러한 볼륨은 백업 용도로 적합할 수 있습니다.

volmode를 *default*로 설정하면 볼륨 노출이 **sysctl vfs.zfs.vol.mode**에 의해 제어됩니다. 시스템 전체에 기본 zvol 모드를 설정할 수 있습니다. 값이 1이면 기본값은 geom, 2이면 dev, 3이면 none을 의미합니다.

라이브 볼륨에서 속성을 변경할 수는 있지만, 이 변경은 아무런 영향을 미치지 않습니다. 이 속성은 볼륨 생성 및 풀 가져오기 중에만 처리됩니다. **zfs rename**으로 볼륨의 이름을 바꾸면 zvol 장치를 다시 만들 수 있습니다.

데이터세트 무결성 (Dataset Integrity)

ZFS의 보호 기능은 대부분 VDEV 계층에서 작동합니다. 결국 블록과 디스크가 손상되는 곳이 바로 이 계층입니다. 하지만 일부 하드웨어는 풀 중복성을 제한합니다. RAID-Z는 말할 것도 없고 미러링을 사용할 수 있는 하드 드라이브가 충분한 노트북은 거의 없습니다. 그러나 데이터 세트 계층에서 체크섬, 메타데이터 중복성 및 복사본을 사용하여 일부 중복성을 제공할 수 있습니다. 대부분의 사용자는 처음 두 개는 절대 건드리지 말아야 하며, 중복 가상 장치를 사용하는 사용자는 세 개를 모두 그대로 두는 것이 좋습니다.

체크섬 (Checksums)

ZFS는 기록하는 모든 블록에 대해 체크섬을 계산하고 저장합니다. 이렇게 하면 블록을 다시 읽을 때 ZFS가 블록이 쓰여졌을 때와 동일한지, 어떤 식으로든 조용히 손상되지 않았는지 확인할 수 있습니다. 체크섬 속성은 데이터 세트가 사용하는 체크섬 알고리즘을 제어합니다. 유효한 설정은 *on, fletcher2, fletcher4, sha256, off, noparity*입니다.

기본값인 *on*은 OpenZFS 개발자가 선택한 알고리즘을 사용합니다. 2015년에는 이 알고리즘이 *fletcher4*이지만 향후 릴리스에서는 변경될 수 있습니다.

표준 알고리즘인 *fletcher4*가 기본 체크섬 알고리즘입니다. 이 알고리즘은 대부분의 용도에 적합하며 매우 빠릅니다. 영원히 *fletcher4*를 사용하려면 이 속성을 *fletcher4*로 설정하면 됩니다. 그러나 기본값을 *on*으로 유지하고 때가 되면 ZFS가 풀의 체크섬 알고리즘을 업그레이드하도록 하는 것이 좋습니다.

off 값은 사용자 데이터에 대한 무결성 검사를 비활성화합니다.

*noparity*는 무결성을 비활성화할 뿐만 아니라 사용자 데이터의 패리티 유지도 비활성화합니다. 이 설정은 RAID-Z 풀에 있는 덤프 장치에서 내부적으로 사용되며 다른 데이터 세트에서 사용해서는 안 됩니다. 체크섬을 비활성화하는 것은 권장하지 않습니다.

이전 버전의 ZFS는 *fletcher2* 알고리즘을 사용했습니다. 이 알고리즘은 이전 풀에서 지원되기는 하지만, 권장되는 알고리즘은 아닙니다. *sha256* 알고리즘은 *fletcher4*보다 느리지만 충돌이 발생할 가능성이 적습니다. 대부분의 경우 충돌은 해롭지 않습니다.

중복 제거 (deduplication)를 수행할 때 *sha256* 알고리즘을 자주 권장합니다.

복사본 (Copies)

ZFS는 중요한 메타데이터의 복사본을 두세 개 저장하며, 중요한 사용자 데이터에 대해서도 동일한 처리를 할 수 있습니다. **copies** 속성은 보관할 사용자 데이터의 사본 수를 ZFS에 알려줍니다. ZFS는 하드웨어 장애에 대비하기 위해 이러한 복사본을 다른 디스크에 저장하거나, 저장하지 못할 경우 물리적 디스크에 가능한 한 멀리 떨어져 저장하려고 시도합니다. **copies** 속성을 늘리면 해당 데이터 집합에 대한 메타데이터 복사본 수도 최대 **3**개 까지 증가합니다.

풀이 2개의 미러 디스크에서 실행되고 **copies**를 3개로 설정한 경우, 데이터 사본이 6개가 됩니다. 이 중 하나는 원시 공급자 장치에서 **dd(1)**을 잘못 사용하거나 지붕에서 떨어져도 살아남아야 합니다.

사본을 늘리거나 줄이면 설정 변경 후에 기록된 데이터에만 영향을 줍니다. **copies**를 1개에서 2개로 변경한다고 해서 모든 데이터의 사본이 갑자기 중복 생성되지는 않습니다. 10MB 용량의 임의 데이터 파일을 만듭니다:

```
$ dd if=/dev/random of=/lamb/random1 bs=1m count=10
10+0 records in
10+0 records out
10485760 bytes transferred in 0.144787 secs (72421935 bytes/sec)
$ zfs set copies=2 mypool/lamb
```

이제 모든 블록이 두 번 저장됩니다. 복사본 중 하나가 손상되더라도 ZFS는 여전히 파일을 읽을 수 있습니다. 체크섬이 일치하지 않기 때문에 어떤 블록이 손상되었는지 알 수 있습니다. 하지만 풀에서 사용되는 공간(풀 목록의 REFER 공간)을 보세요.

```
$ zfs list mypool/lamb
NAME         USED  AVAIL  REFER  MOUNTPOINT
mypool/lamb  10.2M  13.7G  10.1M  /lamb
```

우리가 쓴 10MB만 사용되었습니다. 이 파일은 복사본 속성을 변경하기 전에 썼기 때문에 추가 복사본이 만들어지지 않았습니다. 그러나 복사본이 2로 설정된 상태에서 다른 파일을 쓰거나 원본 파일을 덮어쓰면 디스크 사용량이 달라지는 것을 볼 수 있습니다.

```
$ dd if=/dev/random of=/lamb/random2 bs=1m count=10
10+0 records in
10+0 records out
10485760 bytes transferred in 0.141795 secs (73950181 bytes/sec)
```

지금 디스크 사용량을 확인하세요.

```
$ zfs list mypool/lamb
NAME        USED  AVAIL  REFER  MOUNTPOINT
mypool/lamb  30.2M  13.7G  30.1M  /lamb
```

총 공간 사용량은 30MB로, 첫 번째 무작위 데이터 파일에 10, 두 번째 10MB 파일 사본 2개에 20입니다. **ls(1)**로 파일을 보면 실제 크기만 표시됩니다:

```
$ ls -l /lamb/random*
-rw-r--r--  1 root  wheel  10485760 Apr  6 15:27 /lamb/random1
-rw-r--r--  1 root  wheel  10485760 Apr  6 15:29 /lamb/random2
```

데이터 세트의 복원력을 제대로 활용하고 싶다면 메타데이터 중복성을 살펴보세요.

메타데이터 중복성 (Metadata Redundancy)

각 데이터 세트는 내부 메타데이터의 추가 복사본을 저장하므로 단일 블록이 손상되더라도 손실되는 사용자 데이터의 양이 제한됩니다. 이 추가 복사본은 VDEV 수준에서 제공되는 모든 중복성(예: 미러링 또는 RAID-Z)

에 추가됩니다. 또한 복사본 속성(아래)에 지정된 모든 추가 복사본에 추가되며, 최대 총 3개의 복사본이 생성됩니다.

redundant_metadata 속성을 사용하면 데이터 세트 메타데이터를 얼마나 중복시킬지 결정할 수 있습니다. 대부분의 사용자는 이 속성을 변경하지 않아야 합니다.

redundant_metadata를 *all*(기본값)로 설정하면 ZFS는 모든 메타데이터의 추가 복사본을 저장합니다. 하나의 디스크 블록이 손상되면 최악의 경우 사용자 데이터의 단일 블록이 손실될 수 있습니다.

redundant_metadata를 *most*로 설정하면 ZFS는 대부분의 메타데이터 유형에 대해서만 추가 복사본을 저장합니다. 이렇게 하면 쓰기해야 하는 메타데이터가 줄어들기 때문에 랜덤 쓰기 성능이 향상될 수 있습니다. 대부분의 메타데이터만 중복 저장하는 경우, 하나의 온디스크 블록이 손상되면 최악의 경우 약 100개의 사용자 데이터 블록이 손실될 수 있습니다. 중복 저장되는 메타데이터 블록의 정확한 동작은 향후 릴리스에서 변경될 수 있습니다.

redundant_metadata를 *most*로 설정하고 **copies**를 3으로 설정하고 데이터 세트가 미러 풀에 있는 경우, ZFS는 대부분의 메타데이터 사본 6개와 데이터 및 일부 메타데이터 사본 4개를 저장합니다.

이 속성은 데이터베이스와 같이 메타데이터를 자주 업데이트하는 특정 사용 사례를 위해 설계되었습니다. 데이터가 이미 충분히 강력한 내결함성으로 보호되고 있다면 데이터베이스가 변경될 때마다 작성해야 하는 메타데이터의 복사본 수를 줄이면 성능이 향상될 수 있습니다. 이 값은 자신이 무엇을 하고 있는지 알고 있는 경우에만 변경하세요.

이제 데이터 집합에 대해 이해했으니 풀 유지 관리에 대해 알아보겠습니다.