9️⃣

S3와 유사한 객체 저장소

💡 **AWS S3란 무엇인가?**

- 2006년 6월에 서비스를 시작함
- 2010년부터는 버전 관리, 버킷 정책, 멀티 파일 업로드 기능을 제공하기 시작함
- 2011년부터는 서버 측 암호화, 여러 객체 삭제, 객체 만료 등을 지원하기 시작함
- 2013년에 아마존은 S3에 저장된 객체가 2조 개에 달한다고 보고함
- 2014~2015년에는 수명 주기 정책, 이벤트 알림, 지역 간 복제 기능이 도입됨
- 2021년에 아마존은 S3에 저장된 객체가 100조 개가 넘는다고 보고함

저장소 시스템 101

블록 저장소

- 1960년대에 처음 등장
- HDD나 SDD처럼 서버에 물리적으로 연결되는 형태의 드라이브가 가장 흔한 형태
- 가장 유연하고 융통성 높은 저장소
- 원시 블록을 서버에 볼륨 형태로 제공함
    - 서버는 원시 블록을 포맷한 다음에 파일 시스템으로 이용하거나 애플리케이션에 블록 제어권을 넘겨버릴 수도 있음
    - 데이터베이스나 가상 머신 엔진 같은 애플리케이션은 원시 블록을 직접 제어하여 최대한의 성능을 끌어냄
- 서버에 물리적으로 직접 연결되는 저장소에 국한되지 않음

    - 고속 네트워크를 통해 연결 가능
    - 업계 표준 연결 프로토콜인 FC나 iSCSI를 통해 연결될 수도 있음

⇒ 네트워크를 통해 연결되는 블록 저장소도 원시 블록을 제공한다는 점에서 다르지 않음

⇒ 서버 입장에서는 물리적으로 연결된 블록 저장소와 동일하게 동작

파일 저장소

- 블록 저장소 위에 구현됨
- 파일과 디렉터리를 손쉽게 다룰 때 필요한 더 높은 수준의 추상화를 제공
- 데이터는 계층적으로 구성되는 디렉터리 안에 보관됨
- 가장 널리 사용되는 범용 저장소 솔루션
- 하나의 저장소를 여러 서버에 동시에 붙이기도 가능함
    - SMB/CIFS나 NFS 같은 파일 수준 네트워크 프로토콜 사용
- 폴더나 파일을 같은 조직 구성원에 공유하는 솔루션으로 사용하기 좋음

객체 저장소

- 데이터 영속성을 높이고 대규모 애플리케이션을 지원하며 비용을 낮추기 위해 의도적으로 성능을 희생함
- 실시간으로 갱신할 필요가 없는 데이터 보관에 초점을 맞추며 데이터 아카이브나 백업에 주로 쓰임
- 모든 데이터를 수평적 구조 내에 객체로 보관
- 계층적 디렉터리 구조는 제공하지 않음
- 데이터 접근은 보통 RESTful API를 통함
- 다른 유형의 저장소에 비해 상대적으로 느림
- AWS S3, Azure Blob Storage, ...

용어 정리

- 버킷: 객체를 보관하는 논리적 컨테이너
- 객체: 버킷에 저장하는 개별 데이터
- 버전: 한 객체의 여러 버전을 같은 버킷 안에 둘 수 있도록 하는 기능 / 버킷마다 별도 설정 가능
- URI: 버킷과 객체에 접근할 수 있도록 하는 API URI
- SLA: 서비스 수준 협약 / 서비스 제공자와 클라이언트 사이에 맺어지는 계약

문제 이해 및 설계 범위 확정

비기능 요구사항

- 100PB 데이터
- 식스 나인 수준의 데이터 내구성
- 포 나인 수준의 서비스 가용성
- 저장소 효율성
    - 높은 수준의 안정성과 성능은 보증하되 저장소 비용은 최대한 낮추어야 함

대략적인 규모 추정

객체 저장소는 디스크 용량이나 조당 디스크 IO가 병목될 가능성이 높음

1. 계산을 쉽게 하기 위해 객체 유형별 중앙값을 사용할 것
2. 디스크 용량과 IOPS는 아래와 같은 상황이라고 가정할 것
    - 디스크 용량
        - 객체 가운데 20%는 그 크기가 1MB 미만의 작은 객체
        - 60% 정도의 객체는 1MB~64MB 정도 크기의 중간 크기 객체
        - 나머지 20% 정도는 64MB 이상의 대형 객체

- IOPS
    - SATA 인터페이스를 탑재하고 7200rpm을 지원하는 하드 디스크 하나가 초당 100~150회의 임의 데이터 탐색을 지원 가능

3. 40%의 저장 공간 사용률을 유지하는 경우

- 100PB = 100 * 1000 * 1000 * 1000MB = 10^11MB
- (10^11 * 0.4) / (0.2 * 0.5MB + 0.6 * 332MB + 0.2 * 200MB) = 6억 8천만 개 (0.68billion) 객체
- 모든 객체의 메타데이터 크기가 대략 1KB 정도라고 가정

⇒ 모든 데이터 정보를 저장하기 위해서는 0.68TB가 필요함

개략적 설계안 제시 및 동의 구하기

객체 저장소의 속성

객체 불변성

- 객체 저장소에 보관되는 객체들은 변경이 불가능함
- 삭제한 다음 새 버전 객체로 완전히 대체할 수는 있어도 그 값을 점진적으로 변경할 수는 없음

키-값 저장소

- 객체 저장소를 사용하는 경우 해당 객체의 URI를 사용하여 데이터를 가져올 수 있으므
- URI는 키이고 데이터는 값임 = 키-값 저장소

저장은 1회, 읽기는 여러 번

- 데이터 접금 패턴 = 쓰기 1회, 읽기 여러 번
- 객체 저장소에 대한 요청 가운데 95%가 읽기 요청

소형 및 대형 객체 동시 지원

- 다양한 크기의 객체를 문제 없이 저장할 수 있음
- 메타데이터 저장소에 네트워크를 통해 데이터 저장소에 보관된 객체를 요청하는 데 필요한 식별자가 저장됨
- 데이터 저장소에 데이터가 저장됨
- 메타데이터와 객체의 실제 데이터를 분리하면 설계가 단순해짐
    - 데이터 저장소에 보관되는 데이터는 불변
    - 메타데이터 저장소에 보관되는 데이터는 변경 가능
    - 두 컴포넌트를 독립적으로 구현하고 최정화 가능

개략적 설계안

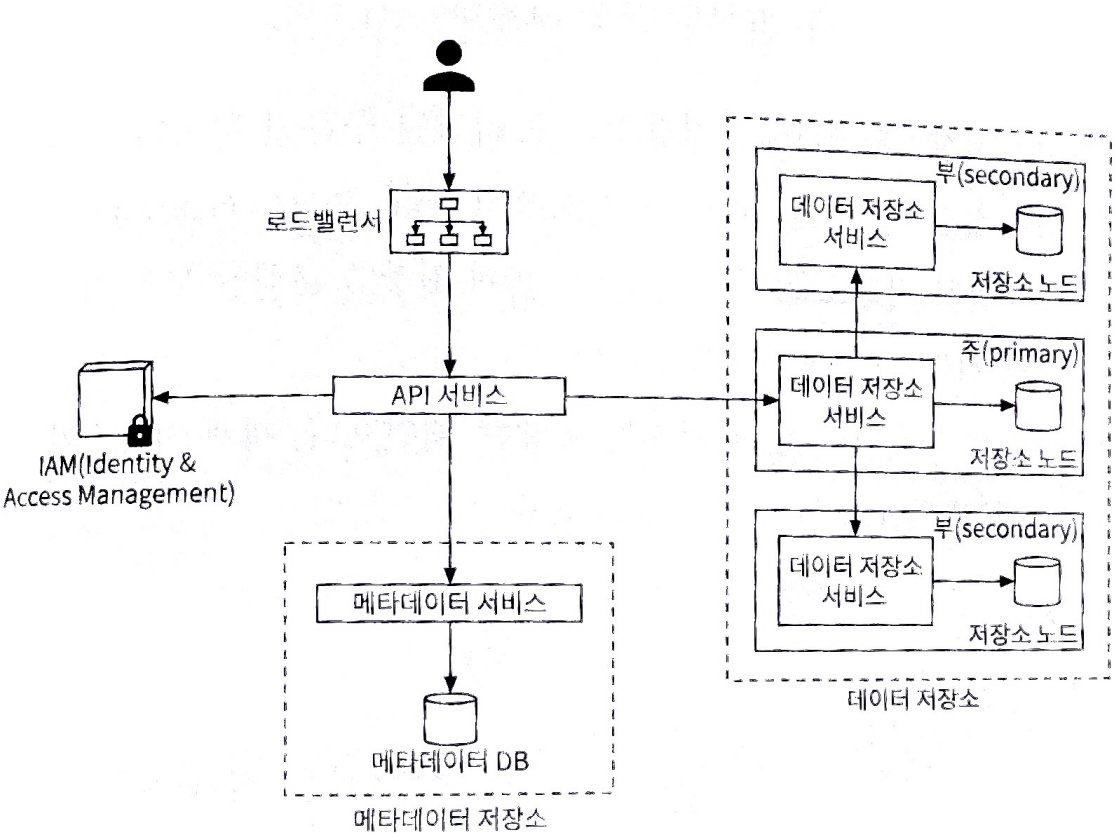


그림 9.4 개략적 설계안

- 로드밸런서: RESTful API에 대한 요청을 API 서버들에 분산
- API 서비스: IAM 서비스, 메타데이터 서비스, 저장소 서비스에 대한 호출을 조율 / 무상태 서비스
- IAM 서비스: 인증, 권한 부여, 접근 제어 등을 중앙에서 맡아 처리
- 데이터 저장소: 실제 데이터를 보관하고 필요할 때마다 읽어가는 장소
- 메타데이터 저장소: 객체 메타데이터를 보관하는 장소

객체 업로드

1. 클라이언트는 API 서비스로 버킷을 생성하기 위한 HTTP PUT 요청을 보냄
2. API 서비스는 IAM을 호출하여 해당 사용자가 WRITE 권한을 가졌는지 확인

3. API 서비스는 메타데이터 데이터베이스에 버킷 정보를 등록하기 위해 메타데이터 저장소를 호출
4. 버킷 정보가 만들어지면 그 사실을 알리는 메시지가 클라이언트에 전송됨
5. 버킷이 만들어지고 나면 클라이언트는 캑체를 생성하기 위한 HTTP PUT 요청을 보냄
6. APU 서비스는 권한 여부 확인
7. 권한이 있으면 HTTP PUT 요청 body에 실린 객체 데이터를 저장소로 전달
8. 데이터 저장소는 해당 데이터를 객체로 저장 후 객체의 UUID 반환
9. API 서비스는 메타데이터 저장소를 호출하여 새로운 항목을 등록

객체 다운로드

1. 클라이언트는 GET 요청을 로드밸런서로 전송하고, 로드밸런스는 이 요청을 API 서버로 전송
2. API 서비스는 IAM을 호출하여 해당 사용자가 READ 권한을 가졌는지 확인
3. 권한이 있으면 API 서비스는 해당 객체의 UUID를 메타데이터 저장소에서 가져옴
4. API 서비스는 해당 UUID를 사용해 데이터 저장소에서 객체 데이터를 가져옴
5. API 서비스는 HTTP GET 요청에 대한 응답으로 해당 객체 데이터를 반환함

상세 설계

데이터 저장소

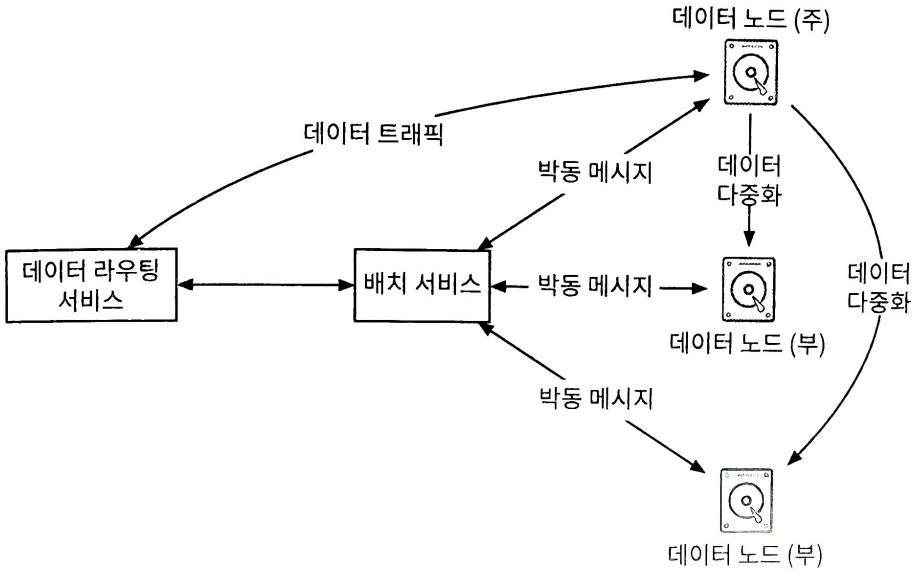


그림 9.8 데이터 저장소 컴포넌트

데이터 라우팅 서비스

- 노드 클러스트에 접금하기 위한 RESTful 또는 gRPC 서비스를 제공
- 무상태 서비스
- 배치 서비스를 호출하여 데이터를 저장할 최적의 데이터 노드를 판단
- 데이터 노드에서 데이터를 읽어 API 서비스에 반환
- 데이터 노드에 데이터 기록

배치 서비스

- 어느 데이터 노드에 데이터를 저장할지 결정하는 역할
- 데이터 노드
  - 주 데이터 노드
  - 부 네이터 노드
- 내부적으로 가상 클러스터 지도를 유지함

    - 클러스터의 물리적 형상 정보가 보관됨
- 가상 클러스터 지도에 보관되는 데이터 노드의 위치 정보를 이용하여 데이터 사본이 물리적으로 다른 위치에 놓이도록 함
    - 높은 데이터 내구성을 달성하는 핵심 요소
- 모든 데이터 노드와 지속적으로 박동 메시지를 주고받으며 상태를 모니터링함
    - 15초의 유예 기간동안 박동 메시지에 응답하지 않는 데이터 노드는 지도에 죽은 노드로 표시함
- 아주 중요한 서비스이기 때문에 5~7개의 노드를 갖는 배치 서비스 클러스터를 팩서스나 래프트 같은 합의 프로토콜을 사용해서 구축할 것을 권장
    - 일부 노드에 장애가 생겨도 건강한 노드 수가 절반 이상이면 서비스를 지속할 수 있도록 보장

데이터 노드

- 실제 객체 데이터가 보관되는 곳
- 여러 노드에 데이터를 복제함으로써 데이터의 안정성과 내구성을 보증 = 다중화 그룹
- 데이터 노드에는 배치 서비스에 주기적으로 박동 메시지를 보내는 서비스 데몬이 돌고 있음
    - 해당 데이터 노드에 부착된 디스크 드라이브의 수
    - 각 드라이브에 저장된 데이터의 양
- 배치 서비스는 못 보던 데이터 노드에서 박동 메시지를 처음 받으면 해당 노드에 ID를 부여하고 가상 클러스터 지도에 추가한 후 특정 정보를 반환
    - 해당 데이터 노드에 부여한 고유 식별자
    - 가상 클러스터 지도
    - 데이터 사본을 보관할 위치

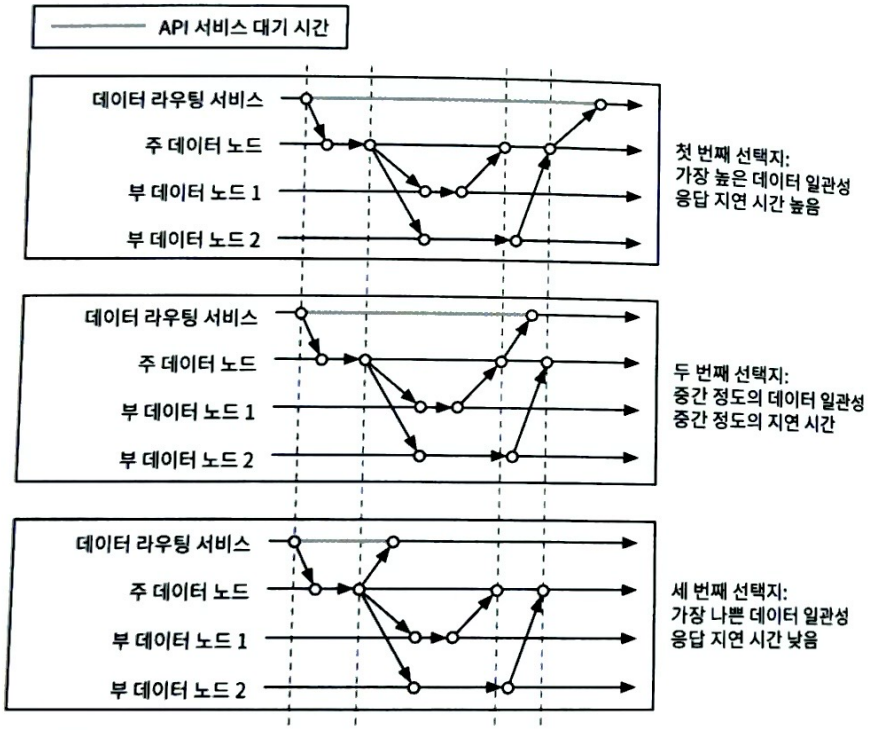
데이터 저장 흐름

데이터를 영속적으로 보관하는 흐름

1. API 서비스는 객체 데이터를 데이터 저장소로 포워딩
2. 데이터 라우팅 서비스는 해당 객체에 UUID를 할당하고 배치 서비스에 해당 객체를 보관할 데이터 노드를 질의
3. 배치 서비스는 가상 클러스터 지도를 확인하여 데이터를 보관할 주 데이터 노드를 반환
4. 데이터 라우팅 서비스는 저장할 데이터를 UUID와 함께 주 데이터 노드에 직접 전송
5. 주 데이터 노드는 데이터를 자기 노드에 지역적으로 저장하고, 두 개의 부 데이터 노드에 다중화
6. 주 데이터 노드는 다중화가 끝나면 데이터 라우팅 서비스에 응답을 보냄
7. 객체의 UUID를 API 서비스에 반환

- 배치 서비스에 UUID를 입력으로 주고 질의하면 해당 객체에 대한 다중화 그룹이 반환됨
    - 배치 서비스의 계산 결과는 결정적이어야 하고, 다중화 그룹이 추가되거나 삭제되는 경우에도 유지되어야 함
        - 보통 안정 해시를 많이 사용함
- 응답을 반환하기 전에 데이터를 모든 부 노드에 다중화하기 때문에 모든 데이터 노드에 강력한 데이터 일관성을 보장

데이터 일관성과 지연 시간 사이의 타협적 관계



- 선택지 1
  - 데이터를 세 노드에 전부 보관하면 성공적으로 보관하였다고 간주
  - 데이터 일관성 측면에서는 최선이지만 응답 지연은 가장 높음
- 선택지 2
  - 데이터를 주 데이터 및 두 개 부 노드 가운데 하나에 성공적으로 보관하면 성공적으로 저장했다고 간주
  - 중간 정도의 데이터 일관성 및 응답 지연을 제공
- 선택지 3
  - 데이터를 주 데이터에 보관하고 나면 성공적으로 저장했다고 간주
  - 데이터 일관성 측면에서는 최악이지만 응답 지연은 가장 낮음

⇒ 선택지 2, 3 모두 결과적 일관성의 한 형태로 볼 수 있음

데이터는 어떻게 저장되는가

- 각각의 객체를 개별 파일로 저장

    - 가장 단순한 방안이지만 작은 파일이 많아지면 성능이 떨어짐
    - 낭비되는 데이터 블록 수가 늘어남
        - 파일 시스템은 파일을 별도의 디스크 블록으로 저장
        - 디스크 블록의 크기는 전부 같기 때문에 작은 파일이 많아지면 낭비되는 블록도 많아짐
    - 시스템의 아이노드 용량 한계를 초과
        - 파일 시스템은 파일 위치 등의 정보를 아이노드라는 특별한 유형의 블록에 저장
        - 대부분의 파일 시스템의 사용 가능한 아이노드의 수는 디스크가 초기화되는 순간 결정됨
        - 작은 파일의 수가 수백만에 달하게 되면 아이노드가 전부 소진될 가능성이 생김
        - 운영체제는 파일 시스템 메타데이터를 공격적으로 캐싱하는 전략을 취하더라도 아주 많은 양의 아이노드를 효과적으로 처리하지 못함
- 작은 객체들을 큰 파일 하나로 모아서 해결
    - 개념적으로는 WAL(Write-Ahead Log)처럼 객체를 저장할 때 이미 존재하는 파일에 추가하는 방식
    - 용량 임계치에 도달한 파일은 읽기 전용 파일로 변경하고 새로운 파일을 만듦
    - 읽기 전용으로 변경된 파일은 오직 읽기 요청만 처리함
    - 읽기-쓰기 파일에 대한 쓰기 연산은 순차적으로 이루어져야 한다는 것을 유의
    - 객체는 파일에 일렬로 저장됨
        - 여러 CPU 코어가 쓰기 연산을 병렬로 진행하더라도 객체 내용이 뒤섞이는 일은 없어야 함
        - 파일에 객체를 기록하기 위해서는 자기 순서를 기다려야 한다는 뜻
        - 많은 코어를 갖는 현대적 서버 시스템의 경우에는 쓰기 대역폭이 심각하게 줄어든다는 문제가 있음
        - 서버에 오는 요청을 처리하는 코어별로 전담 읽기-쓰기 파일을 두어야 함

객체 소재 확인

각각의 데이터 파일 안에 많은 작은 객체가 들어 있는 경우 데이터 노드는 어떻게 UUID로 객체 위치를 찾을까?

- 필요한 정보
  - 객체가 보관된 데이터 파일
  - 데이터 파일 내 객체 오프셋
  - 객체 크기
- 데이터베이스 스키마

object_mapping	
object_id	객체의 UUID
file_name	객체를 보관하는 파일의 이름
start_offset	파일 내 객체의 시작 주소
object_size	객체의 바이트 단위 크기

  - RocksDB 같은 파일 기반 키-값 저장소
    - 쓰기 성능은 좋으나 읽기 성능이 느림
  - 관계형 데이터베이스
    - 읽기 성능은 좋으나 쓰기 성능이 느림
    - 이 데이터는 한 번 기록된 후에는 변경되지 않고, 읽기 연산이 빈번하게 발생
- 데이터베이스 구성
  - 데이터 노드에 저장되는 위치 데이터를 다른 데이터 노드와 공유할 필요가 없음
  - 데이터 노드마다 관계형 데이터베이스를 설치하는 방안이 가능
  - SQLite: 파일 기반 관계형 데이터베이스로 평이 좋음

개선된 데이터 저장 흐름

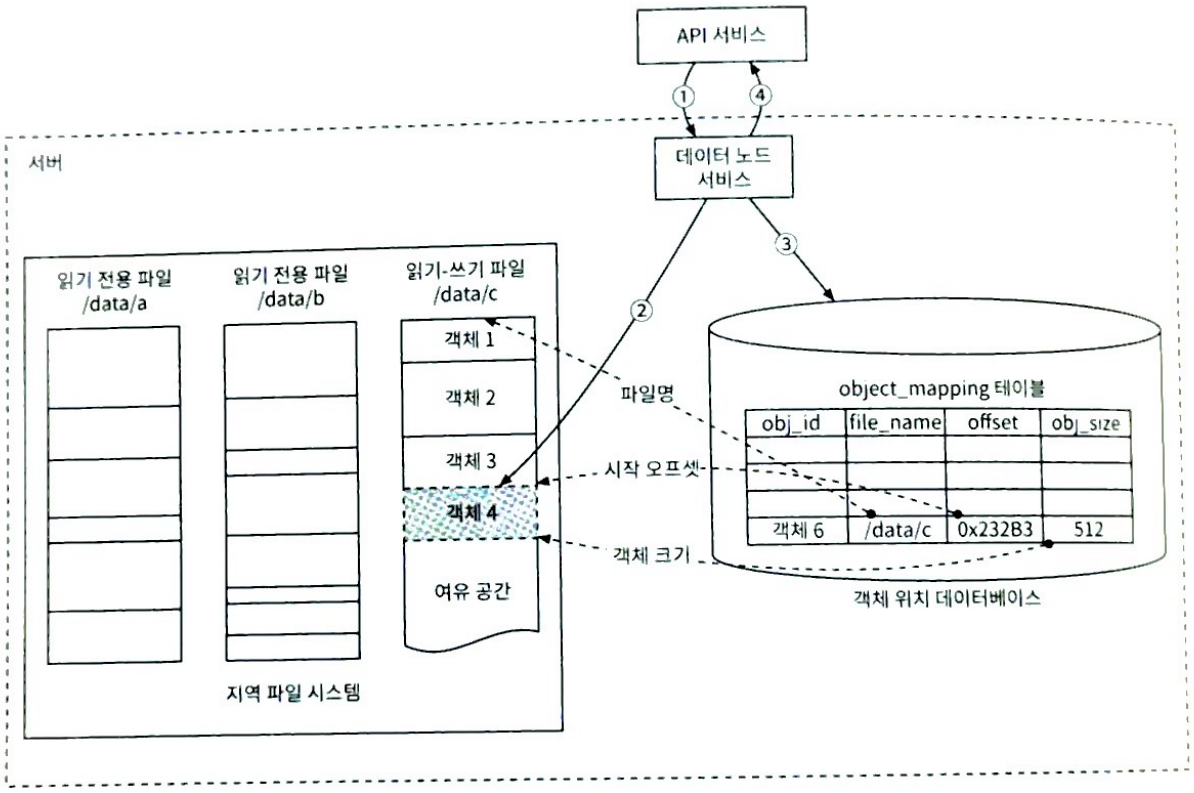


그림 9.13 개선된 데이터 저장 흐름

새로운 객체를 데이터 노드에 저장한는 절차

1. API 서비스는 새로운 객체를 저장하는 요청을 데이터 노드 서비스에 전송
2. 데이터 노드 서비스는 새로운 객체를 읽기-쓰기 파일 /data/c의 마지막 부분에 추가
3. 해당 객체에 대한 새로운 레코드를 object_mapping 테이블에 추가
4. 데이터 노드 서비스는 API 서비스에 해당 객체의 UUID를 반환

데이터 내구성

하드웨어 장애와 장애 도메인

- 기록 매체 종류와 관계없이, 하드 디스크 장애는 피할 수 없음
- 드라이브 한 대로 원하는 내구성 목표를 달성할 수 없음

- 내구성을 높이는 검증된 방법은 데이터를 여러 대의 하드 드라이브에 복제하여 어떤 드라이브에서 발생한 장애가 전체 데이터 가용성에 영향을 주지 않도록 하는 것

⇒ 본 설계안에서는 데이터를 3중 복제함

ex) 회전식 드라이브의 연간 장애율이 0.81%라고 할 때,

데이터를 3중 복제하면 내구성은 ₩-0.0081^3 = ~ 0.999999

- 완전한 내구성 평가를 위해서는 여러 장애 도메인의 영향을 복합적으로 고려해야 함
- 현대적인 데이터센터에서 서버는 보통 랙에 설치됨
  - 랙은 특정한 열/층/방에 위치함
  - 각각의 랙 내 모든 서버는 해당 랙에 부설된 네트워크 스위치와 파워 서플라이를 공유하기 때문에 해당 랙이 나타내는 장애 도메인 안에 있음
  - 현대적인 서버 장비 내 컴포넌트들은 마더보드, CPU, 파워 서플라이, HDD 드라이브 등을 공유하기 때문에 장애 도메인에 속함

대규모의 장애 도메인 사례

- 데이터센터의 가용성 구역
  - 가용성 구역: 보통 다른 데이터센터와 물리적 인프라를 공유하지 않는 독립적 데이터센터 하나
  - 데이터를 여러 AZ에 복제해 놓으면 장애 여파를 최소화할 수 있음
  - 어떤 수준의 장애 도메인을 선택하느냐가 데이터 내구성에 직접적 영향을 끼치지는 않지만, 대규모의 정전이나 냉방 설비 장애, 자연 재해 등의 극단적 문제에 대한 안정성을 높인다는 점에 유의

소거 코드

- 데이터를 3중으로 다중화하면 대략 99.9999%의 내구성을 달성할 수 있음
- 내구성을 달성하는 또 다른 방안
- 데이터 내구성을 다른 관점에서 달성하려 시도함

- 데이터를 작은 단위로 분할하여 다른 서버에 배치
- 그 가운데 일부가 소실되었을 때 복구하기 위한 패리티라는 정보를 만들어 중복성을 확보하는 것
- 장애가 생기면 남은 데이터와 패리티를 조합하여 소실된 부분을 복구

- ex) (4+2) 소거 코드 사례

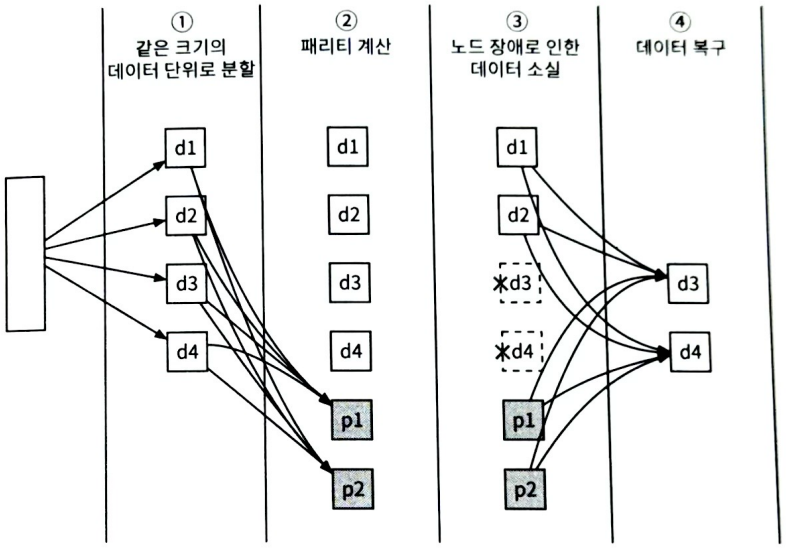


그림 9.15 소거 코드를 통한 데이터 복구

1. 데이터를 네 개의 같은 크기 단위로 분할
2. 수학 공식을 사용하여 패리티 p1, p2를 계산
    a. p1 = d1 + 2 * d2 - d3 + 4 * d4
    b. p2 = -d1 + 5 *d2 + d3 - 3 * d^4
3. 데이터 d3와 d4가 노드 장애로 소실되었다고 가장
4. 남은 값 d1, d2, p1, p2와 패리티 계산에 쓰인 수식을 결합하면 d3와 d4를 복원할 수 있음

소거 코드가 장애 도메인과 어떻게 결부될 수 있는지

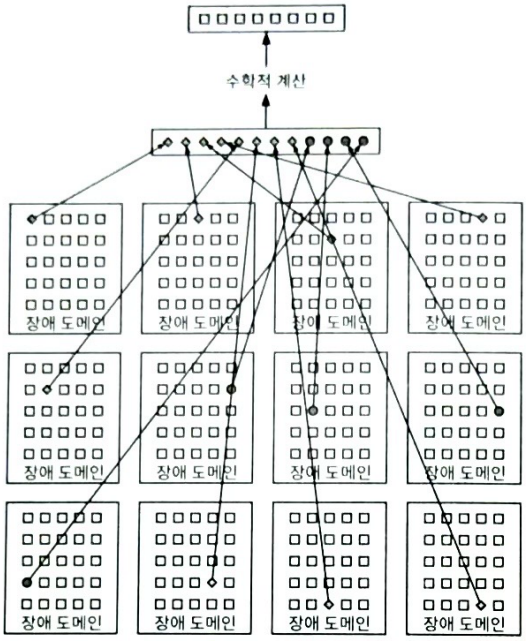


그림 9.16 (8 + 4) 소거 코드

- 8조각 분할 + 4개 패리티를 계산 = 12조각의 데이터는 전부 같은 크기로, 12개의 장애 도메인에 분산
- 최대 4대 노드에 장애가 동시에 발생하더라도 원본 데이터를 복원해 낼 수 있음

소거 코드의 구조적 단점

- 데이터를 다중화 할 경우 데이터 라우터는 객체 데이터를 하나의 건강한 노드에서 읽으면 충분
- 소거 코드를 사용하면 최대 8개의 건강한 노드에서 데이터를 가져와야 함

⇒ 응답 지연은 높아지는 대신 내구성 향상

⇒ 저장소 비용 낮아짐

⇒ 객체 저장소는 저장 비용이 대부분이기 때문에 이런 타협적 측면은 고려할 가치가 있음

소거 코드를 사용하면 추가로 어느 정도의 저장 용량이 필요한가?

- 2개 데이터 블록에 하나의 패리티 블록이 필요하기 때문에 오버헤드는 50%

- 3중 복제 다중화 방안을 채택하는 경우에는 200%

소거 코드가 정말로 데이터 내구성을 높이는가?

- 노드의 연간 장애 발생률이 0.81%라고 했을 때 백블레이즈의 계산 결과에 따르면 소거 코드는 99.999999999%의 내구성을 달성할 수 있음
- 계산이 복잡함

데이터 다중화 vs 소거 코드

	다중화	소거 코드
내구성	99.9999% (3중 복제의 경우)	99.999999999% (8+4 소거 코드를 사용하는 경우) 소거 코드가 우월
저장소 효율성	200%의 저장 용량 오버헤드	50%의 저장 용량 오버헤드 소거 코드가 우월
계산 자원	계산이 필요 없음 다중화가 우월	패리티 계산에 많은 계산 자원 소모
쓰기 성능	데이터를 여러 노드에 복제 추가로 필요한 계산은 없음 다중화가 우월	데이터를 디스크에 기록하기 전에 패리티 계산이 필요하기 때문에 쓰기 연산의 응답 지연 증가
읽기 성능	장애가 발생하지 않은 노드에서 데이터를 읽음 다중화가 우월	데이터를 읽어야 할 때마다 클러스터 내의 여러 노드에서 데이터를 가져와야 함 장애가 발생한 경우 빠진 데이터를 먼저 복원해야 하기 때문에 지연 시간 증가

정확성 검증

- 대규모 시스템의 경우에는 데이터 훼손 문제는 디스크에 국한되지 않고, 메모리의 데이터가 망가지는 일도 자주 일어남
- 메모리의 데이터가 훼손되는 문제는 프로세스 경계에 데이터 검증을 위한 checksum을 두어 해결 가능
    - 데이터 에러를 발견하는 데 사용되는 작은 크기의 데이터 블록

- 원본 데이터의 checksum을 알면 전송 받은 데이터의 정확성은 해당 데이터의 checksum을 다시 계산한 후 해당 절차로 확인 가능
  - 새로 계산한 checksum이 원본 checksum과 다르면 데이터가 망가진 것
  - 같은 경우에는 아주 높은 확률로 데이터는 온전하다고 볼 수 있음
    - 확률이 100%는 아니지만, 아닐 확률이 아주 낮기 때문에 현실적으로는 같다고 할 수 있음
- checksum 알고리즘
  - MD5, SHA1, HMAC, ..
  - 좋은 checksum 알고리즘은 입력이 조금이라도 달라지면 크게 달라진 checksum을 내놓음
  - 본 설계안에서는 MD5 같은 간단한 알고리즘을 사용
- 본 설계안의 경우 checksum은 객체 데이터 끝에 두고 있음
  - 파일을 읽기 전용으로 전환하기 직전에 전체 파일의 checksum을 계싼한 다음에 파일 말미에 추가됨
- ex) (8+4) 소거 코드와 checksum 확인 메커니즘을 동시에 활용하는 경우, 객체 데이터를 읽는 절차
  1. 객체 데이터와 checksum을 가져옴
  2. 수신된 데이터의 checksum을 계산함
     a. 두 checksum이 일치하면 데이터에는 에러가 없다고 간주
     b. checksum이 다르면 데이터는 망가진 것이기 때문에 다른 장애 도메인에서 데이터를 가져와 복구 시도
  3. 데이터 8조각을 전부 수신할 때까지 1과 2를 반복
  4. 원래 객체를 복원한 다음 클라이언트에게 전송

메타데이터 데이터 모델

데이터베이스 스키마는 3가지 질의를 지원할 수 있어야 함

- 객체 이름으로 객체 ID 찾기

- 객체 이름에 기반하여 객체 삽입 또는 삭제
- 같은 접두어를 갖는 버킷 내의 모든 객체 목록 확인

메타데이터 데이터베이스 스키마

bucket		**object**
bucket_name		bucket_name
bucket_id		object_name
owner_id		object_version
enable_versioning		object_id

bucket 테이블의 규모 확장

- 보통 한 사용자가 만들 수 있는 버킷의 수에는 제한이 있고, 이 테이블의 크기는 작음
- ex) 백 만명의 고객이 각각 10개의 버킷을 가지고 있고, 한 레코드의 크기는 10KB

  = 100만 * 10 * 1KB = 10GB = 데이터베이스 서버 한 대에 충분히 저장 가능
- CPU 용량이나 네트워크 대역폭이 부족한 경우에는 데이터베이스 사본을 만들어 읽기 부하를 부산하면 됨

object 테이블의 규모 확장

- 객체 메타데이터를 보관함
- 본 설계안이 다루는 규모의 경우 객체 메타데이터를 데이터베이스 서버 한 대에 보관하기는 불가능
- 샤딩을 통해 객체 메타데이터 테이블의 규모를 확장함
  - bucket_id를 기준으로 같은 버킷 내 객체는 같은 샤드에 배치
    - 버킷 안에 수십억 개의 객체가 있는 핫스팟 샤드를 지원하지 못함
  - object_id를 기준으로 샤딩
    - 부하를 균등하게 분산한다는 측면에서 긍정적임
    - 질의 1과 2를 효과적으로 지원하지 못함

- bucket_name과 object_name을 결합하여 샤딩
  - 대부분의 메타데이터 관련 연산이 객체 URI를 기준으로 하고 있음
  - bucket_name과 데이터를 균등하게 분산하려면 object_name의 순서쌍을 해싱한 값을 샤딩 키로 사용하면 됨
  - 질의 1과 2는 간단하게 지원할 수 있지만 질의 3은 애매함

버킷 내 객체 목록 확인

- 객체 저장소는 객체를 파일 시스템처럼 계층적 구조로 보관하지 않음
- 수평적 경로로 접근함 s3://<버킷 이름>/<객체 이름>
- 사용자가 버킷 내 객체들을 잘 정리할 수 있도록 하기 위해 '접두어'라는 개념을 지원함
  - 객체 이름의 시작 부분 문자열
  - 접두어를 잘 사용하면 디렉터리와 비슷하게 데이터를 정리할 수 있음

AWS S3가 제공하는 목록 출력 명령어가 쓰이는 방법

1. 어떤 사용자가 가진 모든 버킷 목록 출력

```
aws s3 list-buckets
```

2. 주어진 접두어를 가진, 같은 버킷 내 모든 객체 목록 출력

```
aws s3 ls s3://mybucket/abc/
```

```
주어진 접두어 다음에 오는 슬래시 나머지 부분은 잘림

CA/cities/losangeles.txt
CA/cities/sanfrancisco.txt
NY/cities/ny.txt
federal.txt
```

3. 주어진 접두어를 가진, 같은 버킷 내 모든 객체를 재귀적으로 출력

```
aws s3 ls s3://mybucket/abc/ --recursive
```

```
CA/ 접두어를 갖는 모든 항목이 출력됨

CA/cities/losangeles.txt
CA/cities/sanfrancisco.txt
```

단일 데이터베이스 서버

특정 사용자가 가진 모든 버킷으로 출력하는 질의

```
SELECT * FROM bucket WHERE owner_id = {id}
```

같은 접두어를 갖는, 버킷 내 모든 객체를 출력하는 질의

```
SELECT * FROM object
WHERE bucket_id = "123" AND object_name LiKE 'abc/%'
```

분산 데이터베이스 서버

특정 샤드에 있는 데이터 질의(검색 질의를 모든 샤드에 돌린 다음 결과를 취합하는 방법)

1. 메타데이터 서비스는 모든 샤드에 다음 질의를 돌림

```
SELECT * FROM object
WHERE bucket_id = "123" AND object_name LiKE 'a/b/%'
```

2. 메타데이터 서비스는 각 샤드가 반환한 객체들을 취합하여 그 결과를 호출 클라이언트에 반환

⇒ 동작하기는 하지만 페이징 기능은 구현하기 복잡함

일반적으로 페이징하는 방법 = 오프셋 사용

```
SELECT * FROM object
WHERE bucket_id = "123" AND object_name LiKE 'a/b/%'
ORDER BY onject_name OFFSET 0 LIMIT 10
```

객체가 여러 샤드에 나눠져 있기 때문에 샤드마다 반환하는 객체 수가 제각각임

애플리케이션 코드는 모든 샤드의 질의 결과를 받아 취합한 다음 정렬하여 그중 10개만 추려야 함

이번에 반환할 페이지에 포함되지 못한 객체는 다음에 다시 고려해야 함

⇒ 샤드마다 추적해야 하는 오프셋이 달라질 수 있음

⇒ 해당 문제를 해결하는 방법이 있기는 하지만 손해 보는 부분도 있음

- 객체 저장소는 규모와 내구성 최적화에 치중하고, 객체 목록 출력 명령의 성능을 보장하는 것은 우선순위가 높지 않음
- 버킷 ID로 샤딩하는 별도 테이블에 목록 데이터를 비정규화하는 것도 한 가지 방법
    - 객체 목록을 출력할 때는 해당 테이블에 있는 데이터만 사용

객체 버전

- 버킷 안에 한 객체의 여러 버전을 둘 수 있는 기능
- 실수로 지우거나 덮어 쓴 객체를 쉽게 복구할 수 있음

버전이 유지되는 객체의 업로드가 이루어지는 방법

1. 해당 객체가 속한 버킷에 버전 기능을 켜 두기
2. 클라이언트는 객체를 업로드하기 위한 HTTP PUT 요청
3. API 서비스는 사용자의 신원 및 권한 확인

4. 확인 후 문제 없으면 API 서비스는 데이터를 데이터 저장소에 업로드
5. 데이터 저장소는 새 객체를 만들어 데이터를 영속적으로 저장하고 API 서비스에 새로운 UUID 반환
6. API 서비스는 메타데이터 저장소를 호출하여 새 객체의 메타데이터 정보를 보관
7. 버전 기능을 지원하기 위해 메타데이터 저장소의 객체 테이블에는 object_version이라는 이름의 열이 있음
    - 기존 레코드를 덮어쓰는 대신 bucket_id와 object_name은 같지만 object_id, object_version은 새로운 값인 레코드를 추가
    - object_id는 5단계에서 반환한 UUID
    - object_version은 새로운 레코드가 테이블에 추가될 때 만들어지는 TIMEUUID
    - 같은 object_name을 갖는 항목 가운데 object_version에 기록된 TIMEUUID 값이 가장 큰 것이 최신 버전

버전이 다른 객체를 업로드 하는 것 뿐만 아니라 삭제도 가능해야 함

- 객체를 삭제할 때는 해당 객체의 모든 버전을 버킷 안에 그대로 둔 재로 삭제 marker만 추가
- 삭제 marker는 객체의 새로운 버전
- 삽입되는 순간에 해당 객체의 새로운 '현재 버전'이 됨
- 현재 버전 객체를 가져오는 GET 요청을 하면 [404: Object Not Found] 오류가 반환됨

큰 파일의 업로드 성능 최적화

큰 객체를 버킷에 직접 업로드했으면 시간이 아주 오래 걸림

큰 객체는 작게 쪼갠 다음 독립적으로 업로드하는 것이 좋은 방법

모든 조각이 업로드되고 나면 객체 저장소는 그 조각을 모아서 원본 객체를 복원

⇒ 멀티파트 업로드

멀티파트 업로드 동작 방법

1. 클라이언트가 멀티파트 업로드를 시작하기 위해 객체 저장소 호출
2. 데이터 저장소가 uploadId 반환: 해당 업로드를 유일하게 식별한 ID
3. 클라이언트는 파일을 작은 객체로 분할한 뒤에 업로드 시작

   파일 크기는 1.6GB이고 클라이언트가 이 파일을 8조각으로 나눈다고 가정하면 조각 하나의 크기 = 200MB

   클라이언트는 각 파트를 2단계에서 받은 ETag와 함께 데이터 저장소에 올림
4. 조각 하나가 업로드 될 때마다 데이터 저장소는 ETag를 반환

   ETag = 해당 조각에 대한 MD5 해시 checksum / 멀티파트 업로드가 정상적으로 되었는지 검사할 때 사용
5. 모든 조각을 업로드하고 나면 클라이언트는 멀티파트 업로드를 종료하라는 요청을 보냄

   이 요청에는 uploadId, 조각 번호 목록, ETag 목록이 포함되어야 함
6. 데이터 저장소는 전송 받은 조각 번호 목록을 사용해 원본 객체를 복원

   복원에는 몇 분정도 소요될 수 있음
7. 복원이 끝나면 클라이언트로 성공 메시지 반환

객체 조립이 끝난 뒤에는 조각들은 더 이상 쓸모가 없다는 것

이런 조각을 삭제하여 저장 용량을 확보하는 쓰레기 수집 프로세스를 구현할 필요가 있음

쓰레기 수집

쓰레기 수집은 더 이상 사용되지 않은 데이터에 할당된 저장 공간을 자동으로 회수하는 절차

- 객체의 지연된 삭제: 삭제했다고 표시는 하지만 실제로 지우지는 않음
- 갈 곳 없는 데이터: 반쯤 업로드된 데이터 또는 취소된 멀티파트 업로드 데이터
- 훼손된 데이터: checksum 검사에 실패한 데이터

마무리

참 잘했어요~! 👍🏾