S3와 유사한 객체 저장소

S3란?

AWS가 제공하는 RESTful API 기반 인터페이스로 이용 가능한
객체 저장소

저장소 시스템 101

블록 저장소 vs 파일 저장소 vs 객체 저장소

블록 저장소

HDD나 SSD처럼 서버에 물리적으로 연결되는 형태의 드라이브

원시 블록을 서버에 볼륨 형태로 제공한다.

가장 유연하고 융통성이 높은 저장소

파일 저장소

블록 저장소 위에 구현되는 더 높은 수준의 추상화를 제공하는 저장소

데이터는 계층적으로 구성되는 디렉터리 안에 보관된다.

가장 널리 사용되는 범용 저장소 솔루션

객체 저장소

데이터 영속성을 높이고 대규모 애플리케이션을 지원

비용을 낮추기 위해 의도적으로 성능을 희생한다. (다른 유형의 저장소에 비해 상대적으로 느리다.)

데이터 아카이브나 백업에 주로 사용

데이터를 수평적 구조 내에 객체로 보관

용어 정리

- **버킷**: 객체를 보관하는 논리적 컨테이너
    - 데이터를 S3에 업로드하기 위해 우선 버킷을 만들어야 함

    - 버킷 이름은 전역적으로 유일해야 함
- **객체**: 버킷에 저장하는 개별 데이터
    - 메타데이터를 가짐 (객체를 기술하는 key-value 쌍의 집합)
    - 객체 데이터는 어떤 것도 가능
- **버전**: 한 객체의 여러 버전을 같은 버킷 안에 둘 수 있도록 하는 기능
    - 버킷 별 별도 설정 가능
- **URI**: 객체 저장소는 버킷, 객체에 접근할 수 있는 RESTful API 제공
    - 각 객체는 해당 API URI를 통해 고유 식별 가능
- **SLA**: 서비스 제공자 - 클라이언트 간 맺어지는 계약

  아마존 S3 Standard-IA가 만족하는 SLA
    - 여러 AZ에 걸쳐 99.999999999%의 객체 내구성 제공
    - 하나의 AZ가 전체 소실되어도 데이터 복원 가능
    - 연간 99.9% 가용성 제공

▼ 1️⃣ 설계 범위 확정

요구사항

- 지원해야 할 기능
    - 버킷 생성
    - 객체 업로드 / 다운로드
    - 객체 버전
    - 버킷 내 객체 목록 출력 기능
- 데이터 크기
    - 아주 큰 객체(수 GB) ~ 다량의 소형 객체(수 KB)
- 데이터 양
    - 매년 100PB
- SLA
    - 식스 나인(99.9999%)의 데이터 내구성, 포 나인(99.99%)의 서비스 가용성

규모 추정

디스크 용량

객체 크기가 다음 분포를 따른다고 가정

- 객체 중 20%는 1MB 미만의 작은 객체이다.
- 객체 중 60%는 1MB ~ 64MB의 중간 크기 객체이다.
- 나머지는 64MB 이상의 대형 객체이다.

IOPS

SATA 인터페이스를 탑재하고 7200rpm을 지원하는 하드 디스크 하나가 초당 100~150회의 임의 데이터 탐색을 지원할 수 있다고 가정한다. **(= 100~150 IOPS)**

저장 가능한 객체 수 추정

객체 유형별 중앙값을 사용한다(소형: 0.5MB, 중형: 32MB, 대형: 200MB)

- 40%의 저장 공간 사용률을 유지하는 경우 저장소에 수용 가능한 객체 수
  - 100PB = 100 * 1000 * 1000 * 1000 MB = 10^11MB
  - 10^11 * 0.4 / (0.2 * 0.5MB + 0.6 * 32MB + 0.2 * 200MB) = 6억 8천만 개
  - 모든 객체의 메타데이터 크기가 대략 1KB라고 가정 시 메타데이터 저장을 위한 공간 = 0.68TB

▼ 2️⃣ 개략적 설계

🤪 객체 저장소에 관한 4가지 사실

1. 객체 불변성

객체 저장소에 보관되는 객체는 변경이 불가능하다.

삭제한 다음 새 버전 객체로 완전히 대체만 가능하다.

2. 키-값 저장소

객체 저장소에서 객체를 가져오기 위해 해당 객체의 URI를 사용한다.

이 때 URI는 키고, 데이터는 값에 해당하기 때문에 키-값 저장소라고 볼 수 있다.

3. 저장은 1회, 읽기는 여러 번

객체 저장소에 대한 요청 중 95% 가량은 읽기 요청이다. (링크드인 조사)

4. 소형 및 대형 객체 동시 지원

다양한 크기의 객체를 문제 없이 저장할 수 있다.

🗺️ 설계안

- **로드밸런서**: RESTful API에 대한 요청을 API 서버에 분산하는 역할
- **API 서비스**: IAM 서비스, 메타데이터 서비스, 저장소 서비스에 대한 호출을 조율하는 역할
- **IAM 서비스**: 인증, 권한 부여, 접근 제어를 중앙에서 맡아 처리함
- **데이터 저장소**: 실제 데이터를 보관하고 필요할 때마다 읽어가는 장소
  - 모든 데이터 관련 연산은 객체 ID(UUID)를 통함
- **메타데이터 저장소**: 객체의 메타데이터를 보관하는 장소
  - 메타데이터와 데이터 저장소는 논리적인 구분일뿐 구현 방법은 여러가지일 수 있다.

🌊 작업 흐름

객체 업로드

1. 클라이언트는 `bucket-to-share` **버킷을 생성**하기 위한 HTTP PUT 요청을 보낸다.
2. 보낸 요청은 API 서비스로 전달된다.
3. API 서비스는 IAM을 호출하여 해당 사용자가 WRITE 권한을 가졌는지 확인한다.
4. API 서비스는 메타데이터 데이터베이스에 버킷 정보를 등록하기 위해 메타데이터 저장소를 호출한다.
5. 버킷 정보가 만들어지면 그 사실을 알리는 메시지가 클라이언트에 전송된다.
6. 버킷이 만들어지고 나면 클라이언트는 `script.txt` **객체를 생성**하기 위한 HTTP PUT 요청을 보낸다.
7. API 서비스는 해당 사용자의 신원 및 WRITE 권한 소유 여부를 확인한다.
8. 문제가 없으면 API 서비스는 HTTP PUT request body에 실린 객체 데이터를 데이터 저장소로 보낸다.
9. 데이터 저장소는 해당 데이터를 객체로 저장하고 해당 객체의 UUID를 반환한다.
10. API 서비스는 메타데이터 저장소를 호출하여 새로운 항목을 등록한다.

- object_id(UUID), bucket_id, object_name 등의 정보가 포함된다.

객체 다운로드

1. 클라이언트는 `GET /bucket-to-share/script.txt` 요청을 로드밸런서로 보낸다.
2. 로드밸런서는 이 요청을 API 서버로 보낸다.
3. API 서비스는 IAM을 호출하여 사용자가 해당 버킷에 READ 권한을 가졌는지 확인한다.
4. 권한이 있으면 API 서비스는 해당 객체의 UUID를 메타데이터 저장소에서 가져온다.
5. API 서비스는 해당 UUID를 사용해 데이터 저장소에서 객체 데이터를 가져온다.
6. API 서비스는 HTTP GET 요청에 대한 응답으로 해당 객체 데이터를 반환한다.

▼ 3️⃣ 상세 설계

데이터 저장소

주요 컴포넌트

1. **데이터 라우팅 서비스**

   데이터 노드 클러스터에 접근하기 위한 RESTful 또는 gRPC 서비스를 제공하는 무상태 서비스

   - 배치 서비스를 호출하여 데이터를 저장할 최적의 데이터 노드를 판단
   - 데이터 노드에서 데이터를 읽어 API 서비스에 반환
   - 데이터 노드에 데이터 기록

2. **배치 서비스**

   어느 데이터 노드에 데이터를 저장할지 결정하는 역할

   - 내부적으로 가상 클러스터 지도를 유지하여 클러스터의 물리적 형상 정보 보관
   - 지도에 보관되는 데이터 노드의 위치 정보를 이용하여 데이터 사본이 물리적으로 다른 위치에 놓이도록 함
     - **데이터의 물리적인 분리는 높은 데이터 내구성을 달성하는 핵심 !**
   - 모든 데이터 노드와 지속적으로 박동 메시지를 주고받으며 상태를 모니터링함

    - 15초 동안 응답하지 않는 데이터 노드는 지도에 죽은 노드로 표시
- 배치 서비스 클러스터를 *합의 프로토콜*을 사용하여 구축할 것을 권장
    - 일부 노드에 장애가 생겨도 건강한 노드 수가 50% 이상이면 서비스 지속을 보장

3. **데이터 노드**

   실제 객체 데이터가 보관되는 곳

   - 다중화 그룹: 여러 노드에 데이터를 복제함으로써 데이터의 안정성과 내구성을 보증
   - 서비스 데몬: 각 데이터 노드에서 배치 서비스에 주기적으로 박동 메시지를 보내는 역할
   - **박동 메시지**의 내용
     - 해당 데이터 노드에 부착된 디스크 드라이브의 수
     - 각 드라이브에 저장된 데이터의 양
   - 배치 서비스가 처음 박동 메시지를 받은 데이터 노드에게 ID를 부여하고 가상 클러스터 지도에 추가한 후 아래 정보를 반환함
     - 해당 데이터 노드에 부여한 고유 식별자
     - 가상 클러스터 지도
     - 데이터 사본을 보관할 위치

데이터 저장 흐름

1. API 서비스는 객체 데이터를 데이터 저장소로 포워딩한다.
2. 데이터 라우팅 서비스는 해당 객체에 UUID를 할당하고 배치 서비스에 해당 객체를 보관할 데이터 노드를 질의한다.
3. 배치 서비스는 가상 클러스터 지도를 확인하여 데이터를 보관할 주 데이터 노드를 반환한다.
   - 안정 해시를 사용해서 결정적인 계산 결과를 반환하고 다중화 그룹의 추가/삭제에 대비한다.
4. 데이터 라우팅 서비스는 저장할 데이터를 UUID와 함께 주 데이터 노드에 직접 전송한다.
5. 주 데이터 노드는 데이터를 자기 노드에 지역적으로 저장하고 두 개의 부 데이터 노드에 다중화한다.

- 모든 부 데이터 노드에 다중화 성공 시 데이터 라우팅 서비스에 응답을 보낸다. **(강력한 데이터 일관성 보장)**

6. 객체의 UUID(ID)를 API 서비스에 반환한다.

🎁 데이터 일관성 - 지연 시간 trade-off

주 데이터 노드 1개와 부 데이터 노드 2개가 있다고 가정

1. 데이터를 3개 노드에 모두 보관하면 성공적으로 보관했다고 간주하는 경우
   - 데이터 일관성 측면 👍🏻 응답 지연 😡
2. 데이터를 주 데이터 노드 1개, 부 데이터 노드 1개에 보관하면 성공적으로 보관했다고 간주하는 경우
   - 중간 정도의 데이터 일관성과 응답 지연 제공
3. 데이터를 주 데이터 노드에 보관하면 성공적으로 보관했다고 간주하는 경우
   - 데이터 일관성 측면 😡 응답 지연 👍🏻

데이터는 어떻게 저장되는가

작은 파일이 많아지면 성능이 떨어진다 ... 왜?

1. 낭비되는 블록 수가 늘어난다.
2. 시스템의 아이노드 용량 한계를 초과한다.

→ 작은 객체를 큰 파일 하나로 모아서 해결하자 !

참고

- 용량 임계치에 도달한 파일은 읽기 전용으로 변경하고 새로운 파일을 만든다.
- 쓰기 연산은 순차적으로 이루어져야 한다. (내용이 뒤섞이면 안됨)

  ⚠ 쓰기 대역폭이 심각하게 줄어든다?

  - 서버에서 오는 요청을 처리하는 코어별로 전담 읽기-쓰기 파일을 두어야 한다.

객체 소재 확인

어떻게 UUID로 객체 위치를 찾을까?

object_mapping
object_id file_name

- **object_id**: 객체의 UUID
- **file_name**: 객체를 보관하는 파일 이름
- **start_offset**: 파일 내 객체 시작 주소

object_mapping
start_offset object_size

- **object_size**: 객체의 바이트 단위 크기

🤔 어떤 저장소를 쓸까?

1. 파일 기반 키-값 저장소(ex: RocksDB)
    - 쓰기 연산 성능은 좋지만 읽기 성능이 느리다.
2. 관계형 데이터베이스
    - 읽기 연산 성능은 좋지만 쓰기 성능이 느리다.

→ 객체 저장소는 불변성의 특징을 가졌기 대문에 읽기 연산 성능이 좋은 **관계형 데이터베이스**를 선택한다.

- 위치 데이터를 다른 데이터 노드와 공유하지 않아도 되므로 데이터 노드마다 관계형 데이터베이스를 설치할 수 있다.
- 파일 기반 관계형 데이터베이스인 `SQLite`를 추천한다.

개선된 데이터 저장 흐름

1. API 서비스는 새로운 객체를 저장하는 요청을 데이터 노드 서비스에 전송한다.
2. 데이터 노드 서비스는 객체를 읽기-쓰기 파일 `/data/c`의 마지막 부분에 추가한다.
3. 해당 객체에 대한 새로운 레코드를 `object-mapping` 테이블에 추가한다.
4. 데이터 노드 서비스는 API 서비스에 해당 객체의 UUID를 반환한다.

데이터 내구성

하드웨어 장애

- 데이터를 여러 대 하드 드라이브에 복제하여 한 드라이브에서 발생한 장애가 전체 데이터 가용성에 영향을 주지 않도록 한다.

장애 도메인

- 중요한 서비스에 문제가 발생했을 때 부정적인 영향을 받는 물리적 또는 논리적 구획(ex: AZ)

소거 코드

- 데이터를 작은 단위로 분할하여 다른 서버에 배치하고 일부가 소실되었을 때 복구하기 위한 정보(패리티)를 만들어 중복성을 확보한다.
- 장애가 생기면 남은 데이터와 패리티를 조합하여 부분을 복구한다.

- 다중화와의 차이?
  - 내구성과 저장소 효율성이 우월하지만 계산 자원 소모가 많고 성능이 좋지 않음
  - 응답 지연이 중요하다면 **다중화**, 저장소 비용이 중요하다면 소거 코드를 사용하자

정확성 검증

- 데이터 검증을 위한 체크섬을 두어 데이터를 전송받을 때 새로 계산한 체크섬과 원본 체크섬을 비교한다.
- 체크섬 알고리즘: **MD5**, SHA1, HMAC, ...

메타데이터 데이터 모델

스키마

bucket
bucket_name bucket_id owner_id enabler_versioning

object
bucket_name object_name object_version object_id

규모 확장 (1) - `bucket` 테이블

한 사용자가 만들 수 있는 버킷 수에 제한이 있음 → 테이블 크기 작음

CPU 용량이나 네트워크 대역폭의 부족을 대비해 데이터베이스 사본을 만들어 부하 분산

규모 확장 (2) - `object` 테이블

객체 메타데이터 보관 → 샤딩을 통한 규모 확장 필요

1. bucket_id를 기준으로 같은 버킷 내 객체를 같은 샤드에 배치하기
   - 핫스팟 문제 발생 가능
2. object_id를 기준으로 샤딩
   - 부하 균등 분산 가능
   - URI를 기준으로 하는 질의에 효율적 지원 불가
3. bucket_name, object_name을 결합하여 샤딩

- 대부분의 메타데이터 관련 연산은 객체 URI를 기준으로 함을 이용
- 같은 접두어를 갖는 버킷 내 객체 목록 확인 ← 애매...

버킷 내 객체 목록 확인

S3는 버킷 내 객체를 잘 정리할 수 있도록 하기 위해 `접두어` 개념을 지원한다.

접두어를 잘 사용하면 디렉터리와 비슷하게 데이터를 정리할 수 있다.

질의할 때 하위 객체를 조회할 수 없으므로 주어진 접두어를 가진 모든 객체를 재귀적으로 출력해야 한다.

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

객체 버전

실수로 지우거나 덮어 쓴 객체를 쉽게 복구할 수 있는 기능

문서의 모든 이전 버전을 메타데이터 저장소에 유지하고 삭제 플래그를 건다.

큰 파일의 업로드 성능 최적화

멀티파트 업로드를 통해 큰 파일을 잘게 쪼개 독립적으로 업로드하자 !

동작 방식

1. 클라이언트가 멀티파트 업로드를 시작하기 위해 객체 저장소 호출
2. 데이터 저장소가 uploadId(업로드 식별자) 반환
3. 클라이언트는 파일을 작은 객체로 분할한 뒤 업로드 시작
4. 클라이언트는 각 파트를 ETag와 함께 데이터 저장소에 올림
5. 조각 하나가 업로드 될 때마다 데이터 저장소는 ETag를 반환
    - ETag는 기본적으로 해당 조각에 대한 MD5 해시 체크섬(정상 업로드 검사에 이용)
6. 모든 조각 업로드 완료 후 클라이언트는 멀티파트 업로드 종료 요청을 보냄
    - uploadId, 조각 번호 목록, ETag 목록

7. 데이터 저장소는 받은 조각 번호 목록을 사용해 원본 객체 복원
8. 복원이 완료되면 클라이언트에게 성공 메시지 반환

garbage collection

멀티파트 업로드 후에 조각들은 쓸모없고 저장 용량만 확보한다... 그 외 쓰레기들을 치우자

- **객체의 지연된 삭제**: 삭제 표시만 하고 실제로 지우지 않은 것들
- **갈 곳 없는 데이터**: 반쯤 업로드됐거나 취소된 멀티파트 데이터
- **훼손된 데이터**: 체크섬 검사에 실패한 데이터

동작 방식

1. 쓰레기 수집기는 /data/b의 객체를 /data/d로 복사한다. 이 때 이미 삭제된 객체들은 건너뛴다.
2. 모든 객체 복사 후 `object_mapping` 테이블을 갱신한다.
    - file_name, start_offset의 값이 새 위치를 가리키도록 수정된다.
    - 일관성을 보장하기 위해 갱신 연산은 같은 트랜잭션에서 수행하는 것을 권장한다.

▼ 4️⃣ 마무리

- 블록 저장소, 파일 저장소, 객체 저장소의 차이
- 객체 업로드/다운로드, 버킷 내 객체 목록, 객체 버전 기능 구현 방법
- 데이터 저장소, 메타데이터 저장소 구현 방법
- 어떻게 데이터가 영속적으로 저장되는가?
- 데이터의 안정성과 내구성을 높일 방안
- 멀티파트 업로드
- garbage collection