15장 - 구글 드라이브 설계

- **구글 드라이브**는 `파일 저장` 및 `동기화` 서비스로 `문서`, `사진`, 등 `파일`을 **클라우드에 보관**할 수 있도록 함
- 해당 `파일`은 컴퓨터, 스마트폰 등 **어떤 단말에서도 이용 가능**해야 함
- 보관된 `파일`은 치구, 가족, 동료 등 **손쉽게 공유**할 수 있어야 함

1단계 : 문제 이해 및 설계 범위 확정

지원자: 가장 중요하게 지원해야 할 기능들은 무엇인가요?

면접관: 파일 업로드/다운로드, 파일 동기화, 그리고 알림입니다.

지원자: 모바일 앱이나 웹 앱 간운데 하나만 지원하면 되나요, 아니면 둘 다 지원해야 하나

면접관: 둘 다 지원해야 합니다.

지원자: 파일을 암호화해야 할까요?

면접관: 네.

지원자: 파일 크기에 제한이 있습니까?

면접관: 10GB 제한이 있습니다.

지원자: 사용자는 얼마나 됩니까?

면접관: DAU 기준으로 천만(10Million) 명입니다.

이번 장에서 설계에 집중하는 기능 목록

- **파일 추가. (구글 드라이브 안으로 떨구는 drag-and-drop 기능)**
- **파일 다운로드.**

- **여러 단말에 파일 동기화. (한 단말에서 파일을 추가하면 다른 단말에 자동 동기화)**
- **파일 갱신 이력 조회(revision history)**
- **파일 공유.**
- **파일이 편집되거나 삭제되거나 새롭게 공유되었을 때 알림 표시.**

이번 장에서 고려하지 않을 부분

- 구글 문서 편집 및 협업(collaboration) 기능. (여러 사용자의 동시 편집 기능을 고려하지 않음)

비-기능적 요구사항

- **안정성** : 데이터 손실은 발생하면 안 됨
- **빠른 동기화 속도**
- **네트워크 대역폭** : 네트워크 대역폭을 불필요하게 많이 소모하면 안 됨
- **규모 확작성** : 많은 양의 트래픽도 처리할 수 있게
- **높은 가용성** : 일부 서버가 장애가 발생하거나, 느려지거나, 네트워크 일부가 끊겨도 사용할 수 있게

개략적 추정치

- 가입 사용자는 오천 만명, 천만 명의 DAU 사용자가 있음
- 모든 사용자에게 10GB의 무료 저장공간 할당
- 매일 각 사용자가 평균 2개의 파일을 업로드한다고 가정. (각 파일 평균 크기 : `500KB`)
- 읽기:쓰기 비율 - `1:1`
- 필요한 저장공간 총량 = 5천만 사용자 * 10GB = `500 Petabyte(페타바이트)`
- 업로드 API QPS = 1천만 사용자 * 2회 업로드 / 24시간 / 3600초 = 약 240
- 최대 QPS = QPS * 2 = 480

2단계 : 개략적 설계안 제시 및 동의 구하기

- 모든 것을 담은 한 대서버에서 시작해 점진적으로 천만 사용자 지원이 가능한 시스템을 발전한다.

`서버 한 대`로 시작하는 상황

- 파일을 올리고 다운로드 하는 과정을 처리할 웹 서버
- 사용자 데이터, 로그인 정보, 파일 정보 등의 메타데이터를 보관할 데이터베이스
- 파일을 저장할 저장소 시스템. 파일 저장을 위해 1TB의 저장 공간을 사용
- 아파치 웹 서버 설치, MySQL 데이터베이스 설치, 업로드 되는 파일을 저장할 `drive/`라는 디렉터리 준비
  - 디렉터리 내부에는 네임스페이스라 불리는 하위 디렉터리 준비.
  - 각 네임스페이스 안에는 `특정 사용자`가 **올린 파일이 보관**
    - 해당 파일들은 원래 파일과 같은 이름을 가짐
    - 각 파일과 폴더는 그 상대 경로를 네임스페이스 이름과 결합하면 유일하게 식별할 수 있음

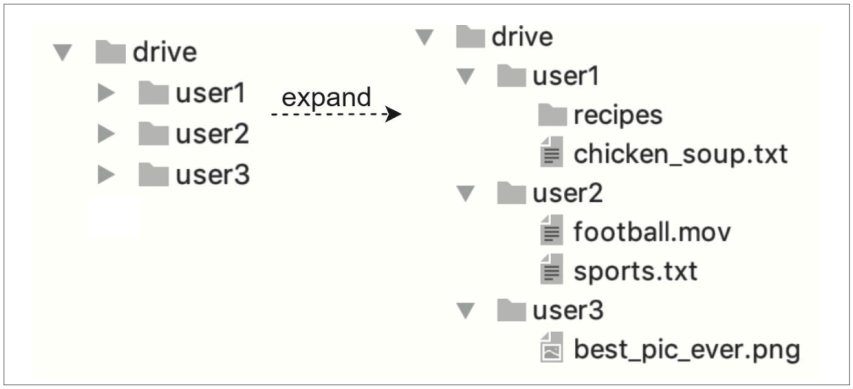

그림 15-3

`drive/` 디렉터리에 실제 파일이 보관된 사례

기본적으로 **세 가지 API**가 필요 (`파일 업로드`, `다운로드`, `파일 갱신 히스토리`)

1. 파일 업로드 API

- 이 시스템은 두 가지 종류의 업로드 지원
  1. **단순 업로드**
     - 파일 크기가 작을 때 사용
  2. **이어 올리기(resumable upload)**
     - 파일 사이즈가 크고 네트워크 문제로 업로드가 중단될 가능성이 높을 때 사용
     - ex) `/upload?uploadType=resumable`
       - 인자 : `uploadType=resumable` , `data: 업로드할 로컬 파일`
     - 이어 올리기의 세 단계의 절차 [2]
       1. 이어 올리기 URL을 받기 위한 최초 요청 전송
       2. 데이터를 업로드하고 업로드 상태 모니터링
       3. 업로드에 장애가 발생하면 장애 발생시점부터 업로드를 재시작

2. 파일 다운로드 API

- ex) `/download`
  - 인자 : `path: 다운로드할 파일의 경로`

```
예)
{
   "path": "/recipes/soup/best_soup.txt"
}
```

3. 파일 갱신 히스토리 API

- ex) `/list_revisions`
  - 인자 : `path: 갱신 히스토리를 가져올 파일의 경로` , `limit: 히스토리 길이의 최대치`

```
예)
{
   "path": "/recipes/soup/best_soup.txt",
```

```
    "limit": 20
}
```

한 대 서버의 제약 극복

- 파일 시스템의 여유공간이 없을 때 방법으로 샤딩하여 여러 서버에 나누어 저장하는 방법이 있다.

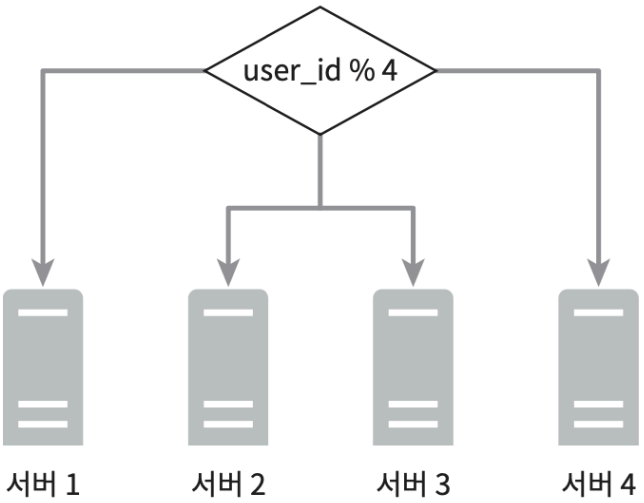

그림 15-5

user_id 기준 샤딩 예제

- 다른 **대형 플랫폼**(넷플릭스 , 에어비엔비)은 저장소로 **아마존 S3를 사용**한다. - **이 서비스도 S3 사용**
  - S3(Simple Storage Service)는 규모 확장성, 가용성, 보안 성능을 제공하는 객체 저장소 서비스이다.
- S3는 다중화도 지원한다. (같은 지역, 여러 지역에 걸쳐 다중화 할 수 있음)

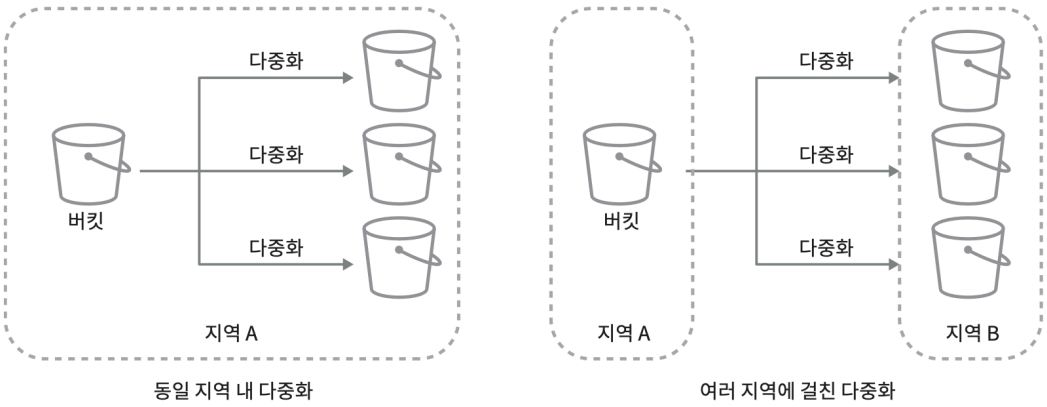

그림 15-6

- 여러 지역에 걸쳐 다중화하면 데이터 손실을 막고 가용성을 최대한 보장할 수 있으므로 `오른쪽`을 선택
- **S3 버킷(bucket)**은 마치 파일 시스템의 폴더와 같다.

- 미래에 비슷한 문제가 벌어지는 것을 막기 위해 미리 개선할 부분은 다음과 같다.
  - `로드밸런서` : 네트워크 트래픽을 분산하기 위해 로드밸런서 사용
    - 로드밸런서를 통해 트래픽을 고르게 분산할 수 있음
    - 특정 웹 서버에 장애가 발생 시 자동으로 해당 서버 우회
  - `웹 서버` : 로드밸런서를 추가하면 웹 서버를 손쉽게 추가할 수 있음 (트래픽 폭증 시 쉽게 대응 가능)
  - `메타데이터 데이터베이스`
    - 데이터베이스를 파일 저장 서버에서 분리하여 SPOF 회피
    - 다중화 및 샤딩 정책을 적용해 규모 확장성 요구사항에 대응
  - `파일 저장소`
    - S3를 파일 저장소로 사용
    - 가용성과 데이터 무손실을 보장하기 위해 두 개 이상의 지역에 데이터 다중화

이렇게 모든 부분을 개선하고 나면 다음 그림처럼

`웹 서버`, `메타데이터 데이터베이스`, `파일 저장소`가 한 대 서버에서 **여러 서버로 잘 분리**된 상황이다.

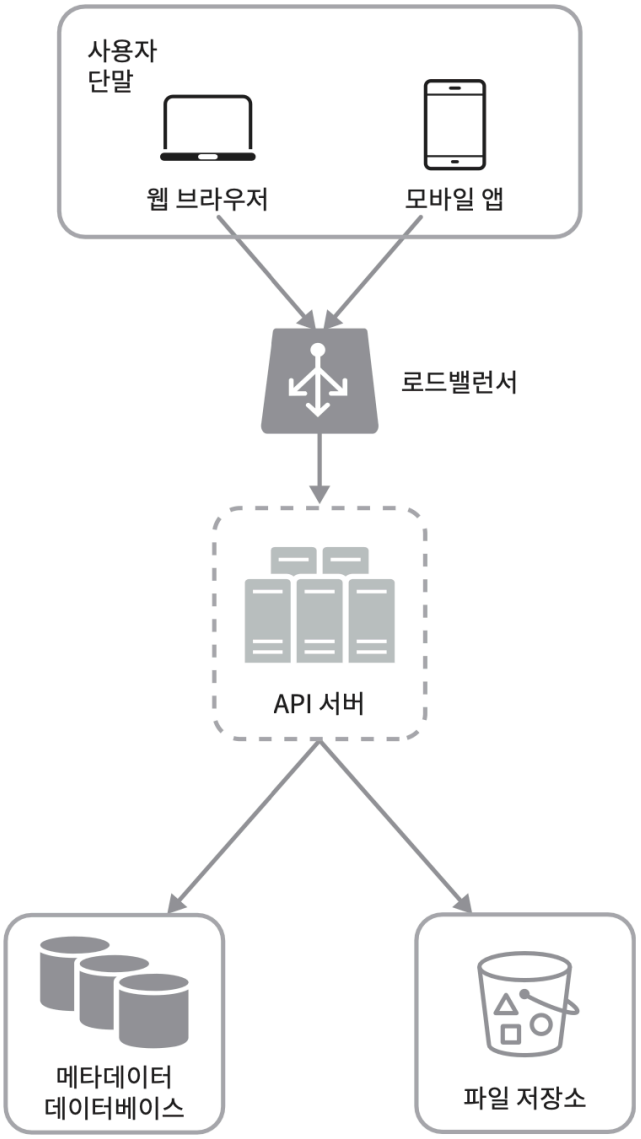

그림 15-7

동기화 충돌

- 두 명 이상의 사용자가 같은 파일이나 폴더를 동시에 업데이트하려고 하는 경우
- 다음 전략을 사용하면 위와 같은 상황을 해소할 수 있다.
  - 먼저 처리 되는 변경은 **성공**, 나중에 처리 되는 변경은 **충돌**한 것으로 표시

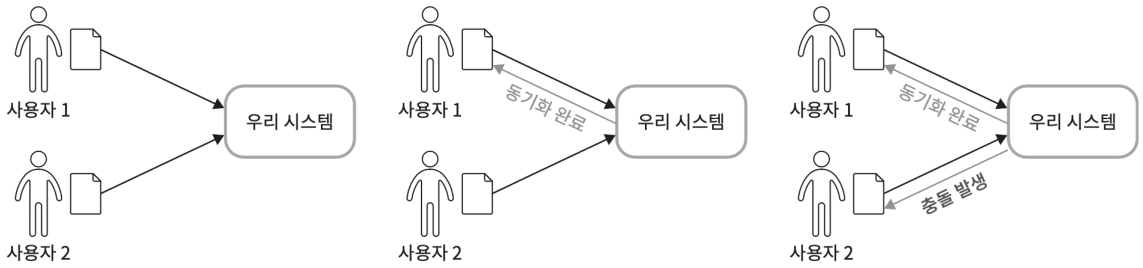

그림 15-8

- 동기화 문제를 해결하는 과정은 [4], [5]를 참고하자.

개략적 설계안

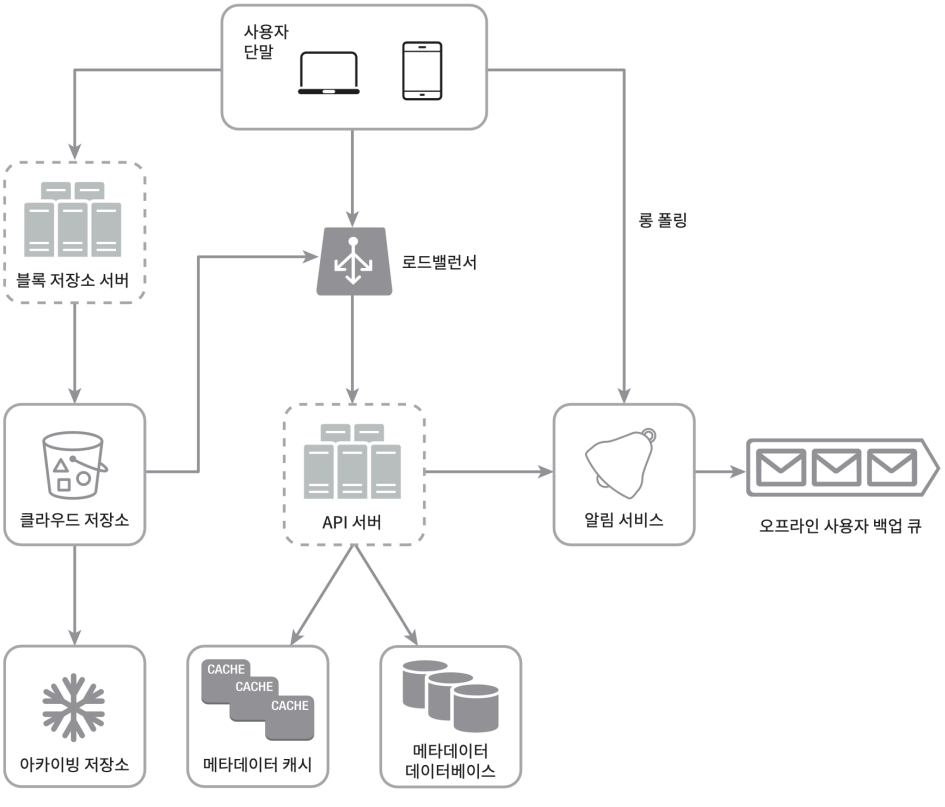

그림 15-10

각 컴포넌트를 살펴보면 다음과 같다.

사용자 단말

- 사용자가 이용하는 웹 브라우저나 모바일 앱 등의 클라이언트

블록 저장소 서버(block server)

- 파일 블록을 클라우드 저장소에 업로드하는 서버
- **블록 수준 저장소(block-lebel storage)**라고도 하며, **클라우드 환경에서 데이터 파일을 저장하는 기술**
- 파일을 여러 개의 블록으로 나눠 젖아하며, 각 블록에는 고유한 해시값이 할당
  - 이 해시값은 메타데이터 데이터베이스에 저장
- 각 블록은 독립적인 객체로 취급, 클라우드 저장소 시스템(ex. S3)에 보관
- 파일을 재구성하려면 블록들을 원래 순서대로 합쳐야 함
  - 이 책 예제에선 한 블록은 드롭박스의 사례[4]를 참고하여 최대 4MB로 정함
    - 4번은 동기화에 대한 부분 아닌가? 6번 youtube를 보면 되나?

클라우드 저장소

- 파일은 블록 단위로 나눠져 클라우드 저장소에 보관

아카이빙 저장소(cold storage)

- 오랫동안 사용되지 않은 비활성(inactive) 데이터를 저장하기 위한 컴퓨터 시스템

로드밸런서

- 요청을 모든 API 서버에 고르게 분산하는 구실

API 서버

- 파일 업로드 외 거의 모든 것을 담당하는 서버
- 사용자 인증, 사용자 프로파일 관리, 파일 메타데이터 갱신 등

메타데이터 데이터베이스

- `사용자`, `파일`, `블록`, `버전` 등의 메타데이터 정보를 관리
- 실제 파일은 클라우드에 보관, 이 데이터베이스에는 오직 메타데이터만 둠

알림 서비스

- 발생/구독 프로토콜 기반 시스템
- 예시 설계안의 경우 클라이언트에게 파일이 추가, 편집, 삭제 되었음을 알려 파일의 최신 상태 확인하는데 사용

오프라인 사용자 백업 큐(offline backup queue)

- 클라이언트가 접속 중이 아니라서 파일의 최신 상태를 확인할 수 없을 때 해당 정보를 이 큐에 두고
  추후 클라이언트가 접속했을 때 동기화될 수 있도록

3단계 : 상세 설계

다루는 내용

- 블록 저장소 서버
- 메타데이터 데이터베이스
- 업로드 절차
- 다운로드 절차
- 알림 서비스
- 파일 저장소 공간 및 장애 처리 흐름

블록 저장소 서버

- 큰 파일들은 업데이트가 일어날 때마다 전체 파일을 서버로 보내면 네트워크 대역폭을 많이 잡아먹게 된다.
  따라서 최적화 방법으로 다음 두 가지가 있다.
  1. **델타 동기화(delta sync)** : 파일이 수정되면 전체 파일 대신 수정이 일어난 블록만 동기화 [7], [8]
  2. **압축(compression)**
     - 블록 단위로 압축해 두면 데이터 크기를 많이 줄일 수 있음
     - 압축 알고리즘은 `파일 유형`에 **따라** 정함 (ex. **텍스트 파일** : `gzip` **이미지** : `다른 알고리즘`)

> 블록 저장소 서버는 클라이언트가 보낸 파일을 **블록 단위로 나누고**, 각 블록에
> **압축 알고리즘을 적용**하고, 필요하다면 **암호화**까지 진행한다.
> 아울러 전체 파일을
> `저장소 시스템`으로 보내는 대신 **수정된 블록만 전송**해야 된다.

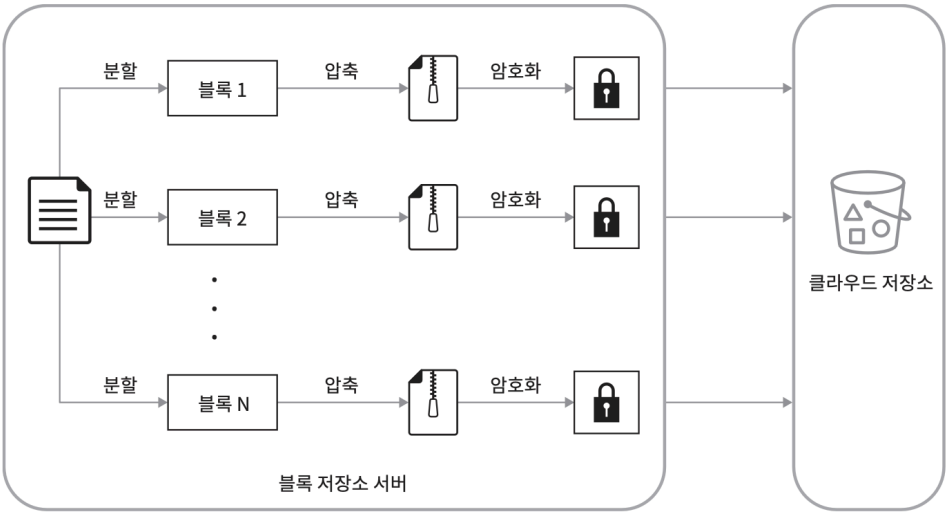

그림 15-11

새 파일이 추가되었을 때 블록 저장소 서버가 어떻게 동작하는지 나타내는 그림

- 주어진 파일을 작은 블록으로 분할
- 각 블록 압축
- 클라우드 저장소로 보내기 전 암호화

- 클라우드 저장소로 전송

델타 동기화 전략의 동작

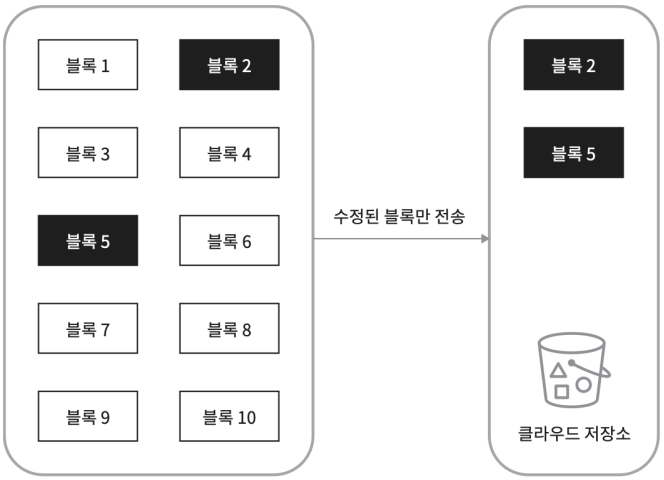

그림 15-12

`검정색 블록`이 `수정된 블록`

- 이렇게 `델타 동기화 전략`과 `압축 알고리즘`으로 네트워크 대역폭 사용량을 줄일 수 있음

높은 일관성 요구사항

- 이 시스템은 **강한 일관성(strong consistency) 모델**을 기본으로 지원해야 한다.
  - `강한 일관성` : 같은 파일이 단말이나 **사용자에 따라 다르게 보이는 것은 허용할 수 없음**
- 메타데이터 캐시와 데이터베이스 계층에도 강한 일관성이 적용되어야 함
  - `메모리 캐시`는 **보통 최종 일관성(eventual consistency) 모델**을 지원
- 따라서 강한 일관성을 달성하려면 다음 사항을 보장해야 함
  - 캐시에 보관된 사본과 데이터베이스에 있는 원본(master)이 일치
  - 데이터베이스에 보관된 원본에 변경이 발생하면 캐시에 있는 사본을 무효화
  - `RDB`는 **ACID(Atomicity, Consistency, Isolation, Durability)를 통해** 강한 일관성 보장 [9]
    하지만
    `NoSQL`은 이를 **기본으로 지원하지 않음** (동기화 로직 안에 프로그램을 넣어야 함)

따라서
해당 설계에서는 ACID를 기본 지원하는 `RDB`를 **채택**

메타데이터 데이터베이스

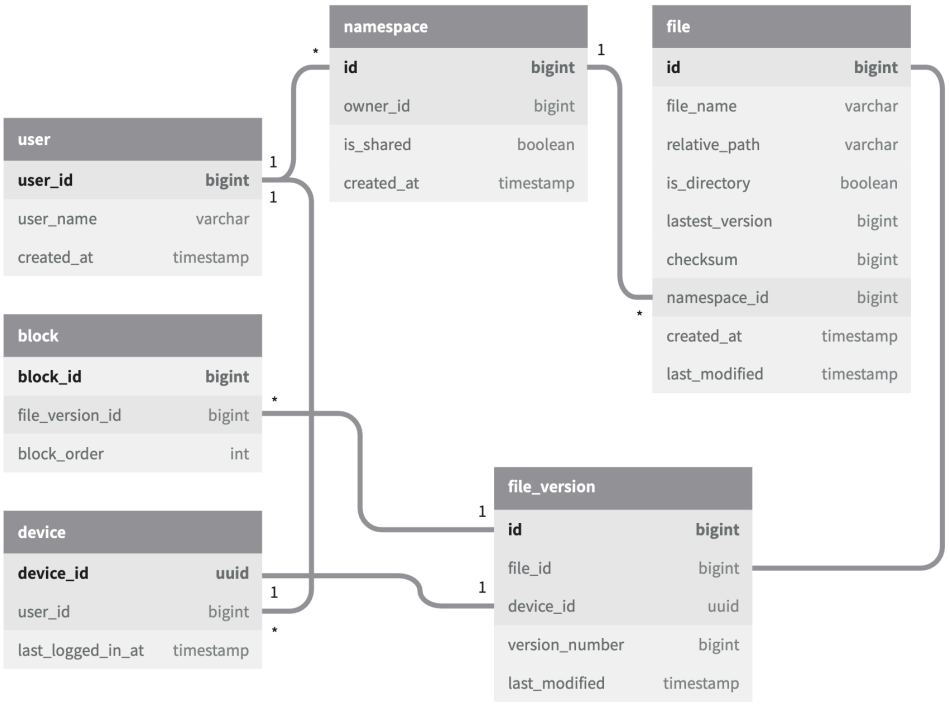

그림 15-13

구글 드라이브 설계를 위한 데이터베이스 스키마 (간추린 형태)

- `user` : 이름, 이메일, 프로파일 사진 등 사용자에 관계된 기본 정보
- `device` : 단말 정보. push_id는 모바일 푸시 알림을 보내고 받기 위한 것, 한 사용자가 여러 단말 가능
- `namespace` : 사용자의 루트 디렉터리 정보 보관
- `file` : 파일의 최신 정보 보관
- `file_version` : 파일의 갱신 이력 보관. (갱신 이력 훼손 방지를 위해 읽기 전용 테이블)
- `block` : 파일 블록에 대한 정보 보관. 특정 버전 파일은 파일 블록을 올바른 순서로 조합하면 복원 가능

업로드 절차

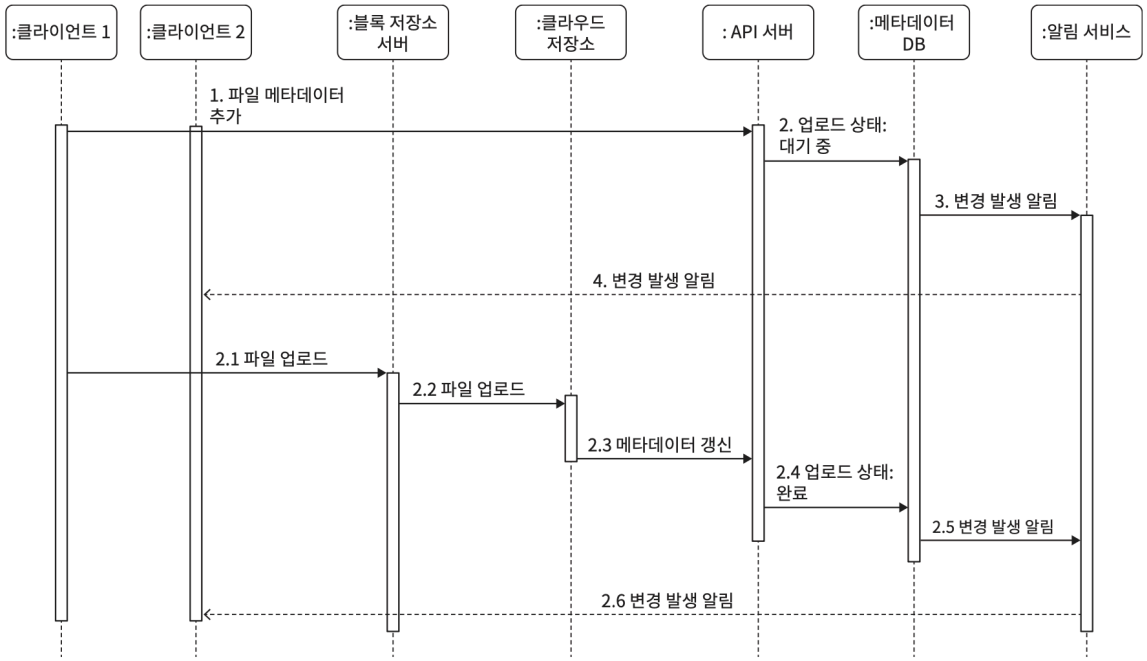

그림 15-14

- 이 그림은 **두 개 요청이 병렬적으로 전송**된 상황이다.

  첫 번째 요청 : 파일 **메타데이터를 추가**하기 위함

  두 번째 요청 : 파일을 **클라우드 저장소로 업로드** 하기 위함

파일 메타데이터 추가

1. 클라이언트 1이 새 파일의 메타데이터를 추가하기 위한 요청 전송
2. 새 파일의 메타데이터를 데이터베이스에 저장하고 업로드 상태를 대기중(pending)으로 변경
3. 새 파일이 추가되었음을 알림 서비스에 통지

파일을 클라우드 저장소에 업로드

2.1. 클라이언트 1이 파일을 블록 저장소 서버에 업로드

2.2. 블록 저장소 서버는 파일을 블록 단위로 쪼갠 다음 압축 후 암호화 한 뒤 클라우드 저장소에 전송

2.3. 업로드가 끝나면 클라우드 스토리지는 완료 콜백 호출. 이 콜백 호출은 API 서버로 전송

2.4. 메타 데이터 DB에 기록된 해당 파일의 상태를 완료(uploaded)로 변경

2.5. 알림 서비스에 업로드가 끝났음을 통지

2.6. 알림 서비스는 관련된 클라이언트(클라이언트 2)에게 파일 업로드가 끝났음을 알림

- `파일 수정`도 위와 비슷

다운로드 절차

- 파일 다운로드는 파일이 새로 추가되거나 편집되면 자동으로 시작
- 클라이언트는 다른 클라이언트가 파일을 편집하거나 추가했다는 사실을 **어떻게 감지할 수 있지?**
    - `클라이언트 A`가 **접속 중**이고 `다른 클라이언트`가 **파일을 변경**하면

      알림 서비스가 `클라이언트 A`에게 **변경이 발생했단 알림 발행** (새 버전을 끌어가야 한다고)
    - `클라이언트 A`가 **네트워크에 연결된 상태가 아닐** 경우 데이터는 **캐시에 보관**될 것
      해당 클라이언트의 상태가
      접속 중으로 바뀌면 그때 해당 클라이언트는 **새 버전을 가져갈** 것 임
- 파일 변경을 감지한 클라이언트는 API 서버를 통해 메타데이터를 새로 가져가야 함.
  이후 블록들을 다운받아 파일을 재구성할 수 있음

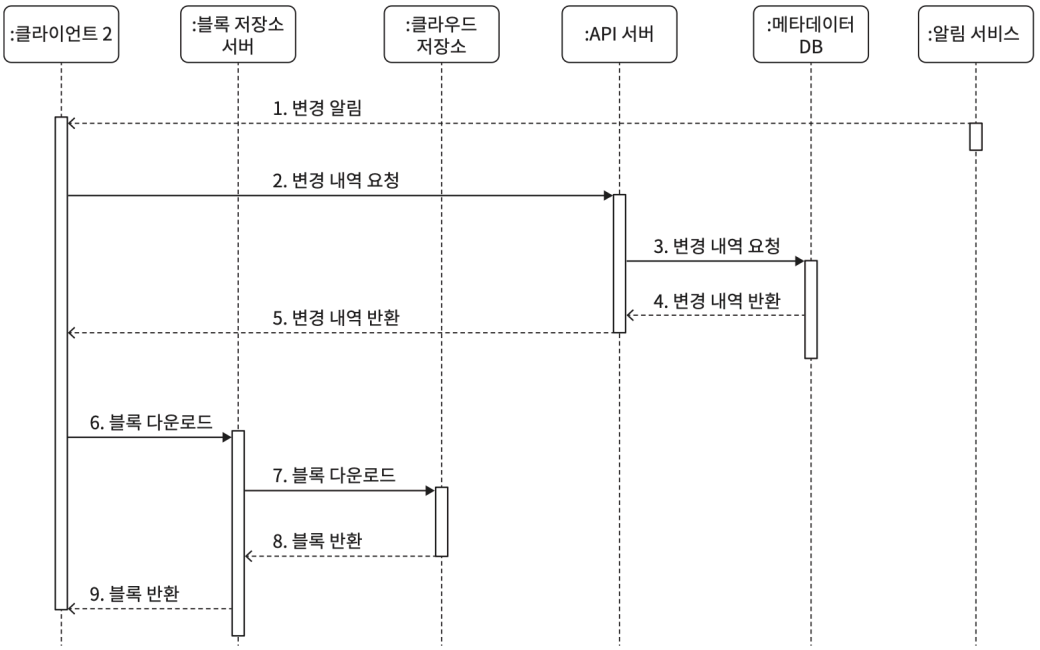

그림 15-15

▼ 위 그림의 자세한 흐름

1. 알림 서비스가 클라이언트 2에게 누군가 파일을 변경했음을 알림
2. 알림을 확인한 클라이언트 2는 새로운 메타데이터 요청
3. API 서버는 메타데이터 데이터베이스에게 새 메타데이터 요청
4. API 서버에게 새 메타데이터가 반환
5. 클라이언트 2에게 새 메타데이터가 반환됨
6. 클라이언트 2는 새 메타데이터를 받는 즉시 블록 다운로드 요청 전송
7. 블록 저장소 서버는 클라우드 저장소에서 블록 다운로드
8. 클라우드 저장소는 블록 서버에 요청된 블록 반환
9. 블록 저장소 서버는 클라이언트에게 요청된 블록 반환.
   클라이언트 2는 전송된 블록을 사용해 파일 재구성

알림 서비스

- **파일의 일관성을 위해** 로컬에서 파일 수정을 감지하는 순간 다른 클라이언트에게 알려야 함. 이때 알림을 사용
  - 단순하게 알림 서비스는 이벤트 데이터를 클라이언트들로 보내는 서비스
- 다음 두 가지 정도의 선택지 존재

1. **롱 폴링(long polling)**. 드롭박스가 이 방식을 채택 [10]
2. **웹소켓(WebSocket)**. 클라이언트와 서버 사이에 지속적인 통신 채널 제공. 양방향 통신 가능

- 이 책에서는 `롱 폴링` 사용 이유는 다음과 같음
  - 채팅 서비스와 달리 알림 서비스와 **양방향 통신이 필요하지 않음**
    - 서버는 파일이 변경된 사시을 클라이언트에게 알려주어야 하지만 반대 바향의 통신을 요구되지 않음
  - 웹소켓은 **실시간 양방향 통신이 요구되는 채팅 같은 서비스에 적합**
    - 구글 드라이브의 경우 알림을 보내는 일이 자주 발생하지 않음
    - 알림을 보낼 때에도 단시간에 많은 양의 데이터를 보낼 필요가 없음
- 각 클라이언트는 알림 서버와 롱 폴링용 연결을 유지하다 특정 파일에 대한 변경을 감지하면 해당 연결을 끊음
  - 이때 클라이언트는 반드시 메타데이터 서버와 연결해 파일의 최신 내역을 다운로드 해야 함
  - 해당 다운로드 작업이 끝났거나 연결 타임아웃 시간에 도달한 경우 즉시 새 요청을 보내 롱 폴링 연결 복원

(파일) 저장소 공간 절약

- 파일 갱신 이력의 보존과 안전성을 위해 파일의 여러 버전을 여러 데이터센터에 보관
  - 그렇다고 모든 버전을 자주 백업하면 저장용량 부족
  - 이런 문제 해결을 위해 다음의 `세 가지 방법` 사용

중복 제거(de-depe)

- 중복된 파일 블록을 계정 차원에서 제거하는 방법
- 두 블록이 같은 블록인지는 해시 값을 비교

지능적 백업 전략 도입

- **한도 설정** : 보관해야 하는 파일 버전 개수에 상한을 두는 방식 (상한에 도달하면 제일 오래된 버전은 버림)
- **중요한 버전만 보관** : 불필요한 버전과 사본이 만들어지는 것을 피하려면 그 중 중요한 것만 골라야 됨

> ### 자주 쓰이지 않는 데이터는 아카이빙 저장소(cold storage)로 이동

- 몇달 혹은 수년간 이용되지 않은 데이터가 이에 해당
- 아마존 S3 글래시어(glacier) 같은 아카이빙 저장소 이용료는 S3보다 훨씬 저렴 [11]

장애 처리 흐름

- **로드밸런서 장애**
  - 부(secondary) 로드밸런서가 활성화되어 트래픽을 이어 받아야 함
  - 로드 밸런서끼리는 보통 heartbeat 신호를 주기적으로 보내 상태를 모니터링 함
  - 일정 시간 동안 박동 신호에 응답하지 않은 로드밸런서는 장애가 발생한 것으로 간주
- **블록 저장소 서버 장애**
  - 블록 저장소 서버에 장애가 발생했다면 다른 서버가 미완료 상태 또는 대기 상태인 작업을 이어 받아야 함
- **클라우드 저장소 장애**
  - S3 버킷은 여러 지역에 다중화할 수 있으므로,
    특정 지역에서 장애가 발생했다면 다른 지역에서 파일을 가져오자.
- **API 서버 장애**
  - API 서버들은 무상태 서버
  - 즉, 로드밸런서는 API 서버에 장애가 발생하면
    트래픽을 해당 서버로 보내지 않음으로써 장애 서버를 격리할 수 있음
- **메타데이터 캐시 장애**
  - 메터데이터 캐시 서버도 다중화
  - 한 노드에 장애가 생겨도 다른 노드에서 데이터를 가져올 수 있음

    - 장애가 바생한 서버는 새 서버로 교체
- **메타데이터 데이터베이스 장애**
    - `주` 데이터베이스 서버 장애
        - `부 데이터베이스` 서버 가운데 하나를 주 데이터베이스로 `승격`
        - 부 데이터베이스 서버 새로 증설
    - `부` 데이터베이스 서버 장애
        - 다른 부 데이터베이스 서버가 읽기 연산을 처리하도록 하고 장애 서버는 새 것으로 교체
- **알림 서비스 장애**
    - 접속 중인 모든 사용자는 알림 서버와 롱 폴링 연결을 하나씩 유지
        - 즉, 알림 서비스는 많은 사용자와의 연결을 유지하고 관리해야 함
        - 2012 드롭박스 발표자료[6]에 따르면 한 대의 드롭박스 알림 서버가 관리하는 연결의 수는
          1백만 개가 넘음. 즉, 한 대 서버가 발생하면 백만 명 이상의 사용자가 롱 폴링 연결을 다시 해야 함
          주의할 점으로
          `한 대 서버로 백만 개 이상의 접속을 유지하는 것은 가능` 하지만

          `동시` 에 백만 개 접속을 `시작하는 것` 은 불가능
        - 따라서 롱 폴링 연결을 복구하는 것은 상대적으로 느릴 수 있음
- **오프라인 사용자 백업 큐 장애**
    - 이 큐 또한 다중화해야 함
    - 큐에 장애가 발생하면 구독 중인 클라이언트들은 백업 큐로 구독 관계 재설정

4단계 : 마무리

- 이번 장에서 만든 설계안을 크게 두 가지 부분으로 나누면 다음과 같다.
    1. 파일의 메타데이터를 관리하는 부분
    2. 파일 동기화를 처리한느 부분
- 알림 서비스는 위 두 부분과 병존하는 또 하나의 중요 컴포넌트

    - 롱 폴링을 사용하여 클라이언트로 하여금 파일의 상태를 최신으로 유지할 수 있음
- 정답은 없음. 회사마다 요구하는 제약 조건이 다름

추가로 논의

- 블록 저장소 서버를 거치지 않고 파일을 클라우드 저장소에 직접 업로드 한다면?
    - 이 방식의 장점은 파일 전송을 클라우드 저장소로 직접 하게 되니까 업로드 시간이 빨라짐
    - 단점은 다음과 같음
        - `분할`, `압축`, `암호화 로직`을 **클라이언트에** 두어야 함
            - 플랫폼별로 따로 구현해야 함 - (iOS, 안드로이드, 웹 등) 따로 구현
            - 위에서 살펴본 예제는 이 모두를 `블록 저장소 서버`라는 곳에 모아 뒀으므로 이럴 필요가 없음
        - 클라이언트가 해킹 당할 가능성이 있으므로 암호화 로직을 클라이언트에 두는 것은 좋지 않음
- 접속 상태를 관리하는 로직을 별도 서비스로 옮기는 것
    - 관련 로직을 알림 서비스에서 분리한다면 다른 서비스에서도 쉽게 활용할 수 있게 됨

Chapter 15: DESIGN GOOGLE DRIVE

1. Google Drive
2. Upload file data
3. Amazon S3
4. Differential Synchronization
5. Differential Synchronization Youtube Talk
6. How We've Scaled Dropbox
7. The rsync algorithm - Andrew Tridgell and Paul Mackerras (1996)
8. Librsync

9. ACID
10. Dropbox Security Whitepaper
11. Amazon S3 Glacier