

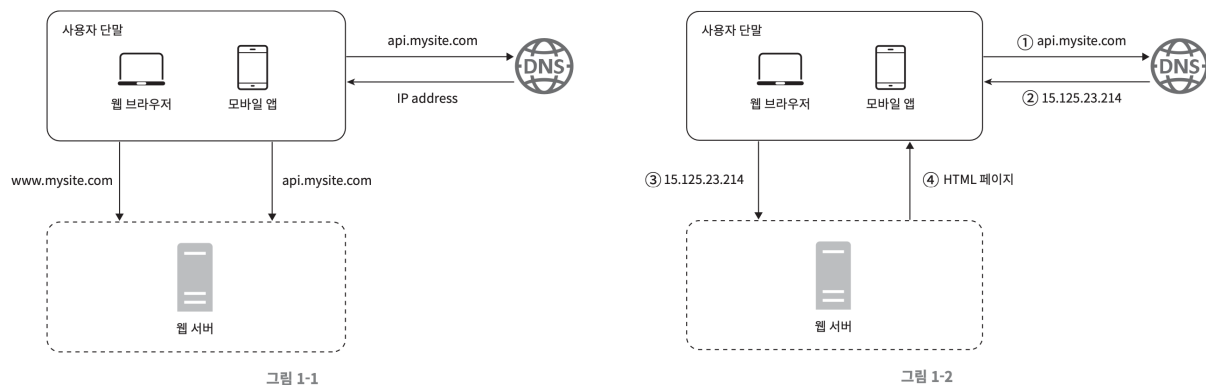
# 1장 - 사용자 수에 따른 규모 확장성

이 장에서는 **한 명의 사용자**를 지원하는 시스템에서 → **몇 백만 사용자**를 지원하는 시스템을 설계한다.

## 단일 서버

복잡한 시스템을 만들기 전 모든 컴포넌트가 단 한 대의 서버에서 실행되는 간단한 시스템부터 설계한다.

다음 그림처럼 웹, 앱, 데이터베이스, 캐시 등 전부 서버 한 대에서 실행된다.



웹, 앱, 데이터베이스, 캐시 모두 서버 한 대에서 실행

요청 흐름

오른 쪽 그림의 요청 흐름을 확인해보자.

1. 사용자는 도메인 이름(api.mysite.com)을 이용하여 웹 사이트에 접속
  - 도메인 이름을 도메인 이름 서비스(Domain Name Service, DNS)에 질의하여 IP 주소로 변환 필요
  - DNS는 보통 제 3 사업자(third party)가 제공하는 유료 서비스를 이용하게 된다. (내 시스템이 아님)
2. DNS 조회 결과로 IP 주소가 반환된다. (책 예제에서 15.125.23.214, 웹 서버의 주소)
3. 해당 IP 주소로 HTTP(HyperText Transfer Protocol) 요청이 전달
4. 요청을 받은 웹 서버는 HTML 페이지나 JSON 형태의 응답을 반환

- 이러한 요청들은 두 가지 종류의 단말로 온다. (사용자의 웹 앱, 모바일 앱)

- 웹 애플리케이션

- 비즈니스 로직, 데이터 저장 등을 처리하기 위해서는 서버 구현용 언어(자바, 파이썬)를 사용하고,  
프리젠테이션용으로는 클라이언트 구현용 언어(HTML, JS 등) 사용

- 모바일 앱

- 모바일 앱과 웹 서버 간 통신을 위해서는 HTTP 프로토콜을 이용한다.
  - HTTP 프로토콜을 통해서 반환될 응답 데이터의 포맷으로는 보통 JSON 이 사용된다.

## 데이터베이스

사용자가 늘면 서버 하나로는 충분하지 않아서 여러 서버를 두어야 한다.

- 웹 / 모바일 트래픽 처리 서버 1, 데이터베이스 서버 1 이런 식으로

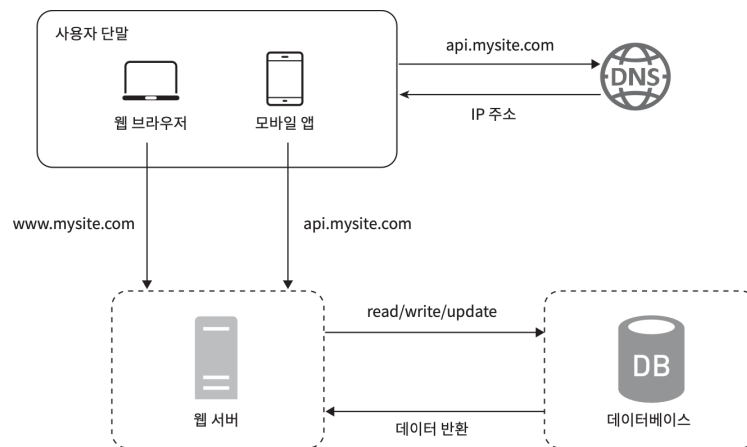


그림 1-3

## 어떤 데이터베이스를 사용할 것인가?

- 가장 기본적으로 RDB와 NoSQL에서 선택할 수 있다.
  - RDB(Relational Database Management System, RDBMS) - MySQL, 오라클, 등등
    - 테이블과 열, 컬럼으로 표현한다.

- SQL을 사용하면 여러 테이블에 있는 데이터를 관계에 따라 조인하여 합칠 수 있다.
- **NoSQL(비 관계형 데이터베이스) - CouchDB, Neo4j, Cassandra, HBase, DynamoDB 등**
  - NoSQL은 다음과 같이 네 부류로 나눌 수 있다.
    - 키-값 저장소 (key-value store)
    - 그래프 저장소 (graph store)
    - 컬럼 저장소 (column store)
    - 문서 저장소 (document store)
  - 비-관계형 데이터베이스는 일반적으로 조인 연산은 지원하지 않는다. (join이 필요하면 잘 못 설계)

▼ 일반적으로 RDB가 좋지만 **다음 상황에서는 비-관계형 데이터베이스가 좋은 선택**일 수 있다.

- 아주 낮은 응답 지연시간(latency)이 요구됨
- 다루는 데이터가 비정형(unstructured)이라 관계형 데이터가 아님
- 데이터(JSON, YAML, 등등)를 직렬화하거나(serialize) 역직렬화(deserialize) 할 수 있기만 하면 됨
- 아주 많은 양의 데이터를 저장할 필요가 있음

## 수직적 규모(scale up) 확장 vs 수평적 규모(scale out) 확장

- 수직적 규모 확장(scale up, vertical scaling)
  - 서버에 **고사양 자원**(더 좋은 CPU, 더 많은 RAM 등)을 추가하는 행위
- 수평적 규모 확장(scale out, horizontal scaling)
  - **더 많은 서버**를 추가하여 성능을 개선하는 행위

| 어떤 것이 더 유리할까?

서버로 유입되는 **트래픽 양이 적을 때는 수직적 확장**이 좋은 선택이며, 이 방식의 **장점은 단순함**이다.

#### 단, 스케일 업의 단점

- 한 대의 서버에 CPU나 메모리를 무한대로 증설할 방법은 없다.
- 장애에 대한 **자동복구(failover) 방안**이나 **다중화(redundancy) 방안**을 제시하지 않음  
즉, 서버에 장애가 발생하면 웹 사이트/ 앱은 완전히 중단한다.

이런 단점 때문에 **대규모 애플리케이션을 지원하는 데는 수평적 규모 확장법이 더 적절하다.**

[그림 1-2] 설계에서 사용자는 웹 서버로 바로 연결된다. 웹 서버가 다운되면 웹 사이트에 접속할 수 없고,  
많은 사용자가 접속하는 경우 웹 서버의 부하가 심해지면 응답 속도가 느려지거나 서버 접속이 불가능해 수 있다.

이런 문제를 해결하는 데는 **부하 분산기** 또는 **로드 밸런서(load balancer)**를 도입하는 것이 최선이다.

## 로드 밸런서 (부하 분산기)

부하 분산 집합(load balancing set)에 속한 웹 서버들에게 트래픽 부하를 고르게 분산하는 역할을 한다.

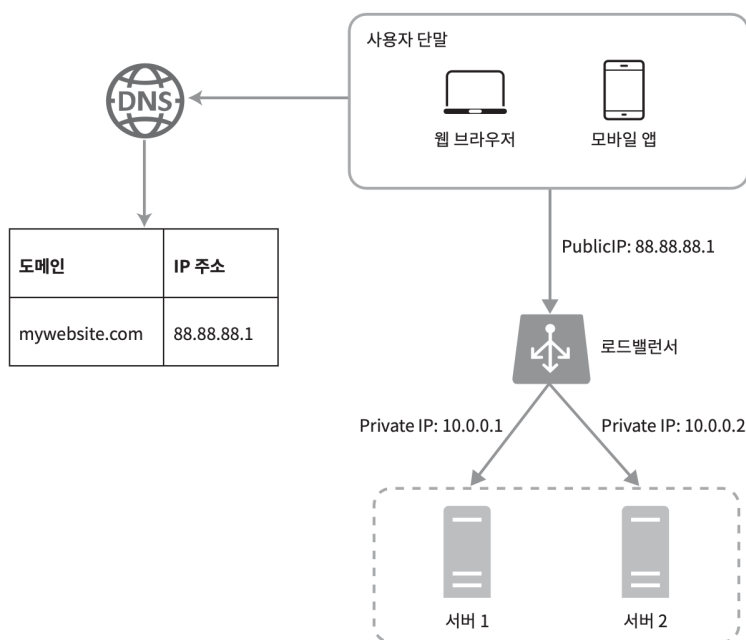


그림 1-4

사용자는 로드밸런서의 공개 IP 주소로 접속한다. 이때 웹 서버는 클라이언트의 접속을 직접 처리하지 않는다.

더 나은 보안을 위해서 서버 간 통신에는 사설 IP 주소가 이용된다.

- **사설 IP 주소 - 같은 네트워크에 속한 서버 사이의 통신에만 쓰일 수 있는 IP 주소**
  - 웹과 같은 인터넷을 통해서 접속할 수 없다.

이렇게 된 상태에서 부하 분산 집합에 다른 웹 서버를 추가하면 장애를 자동 복구하지 못하는 문제(no failover)는 해소된다. **웹 계층의 가용성(availability)은 향상된다.**

## 웹 계층의 가용성이 향상되는 구체적인 상황

- 서버 1이 다운되면(offline) 모든 트래픽은 서버 2로 전송된다. 즉, 웹 사이트 전체가 다운되는 일이 방지  
부하를 나누기 위해 새로운 서버를 추가할 수도 있다.
- 로드 밸런서가 있다면 웹 서버 계층에 더 많은 서버를 추가하는 형태로 트래픽이 많아질 때 부하 분산 가능

## 그렇다면 데이터베이스 계층은?

- 위 처럼 로드 밸런서와 앱 서버를 추가하는 형태로 부하 분산을 하면 웹 계층은 괜찮다. 이때 데이터 계층은 어떻게 할 수 있을까?
- 현재 설계에서 하나의 데이터베이스 서버 뿐이고, failover나 다중화를 지원하지 않는다.
- **데이터베이스 다중화**를 통해 이런 문제를 해결할 수 있다.

## 데이터베이스 다중화

데이터베이스 다중화 (위키피디아) 정의

- 많은 데이터베이스 관리 시스템이 다중화를 지원한다.
- 보통 서버 사이의 주(master)-부(slave) 관계를 설정하고,

**데이터 원본은 주 서버에, 사본은 부 서버에 저장하는 방식이다.**

- **쓰기 연산(write operation)은 마스터에서만 지원**

- insert , delete , update
  - 부 데이터베이스는 마스터로부터 사본을 전달받으며 읽기 연산(read operation)만을 지원
- read

대부분의 애플리케이션은 읽기 연산의 비중이 높아 통상 부 데이터베이스의 수가 주 데이터베이스보다 많다.

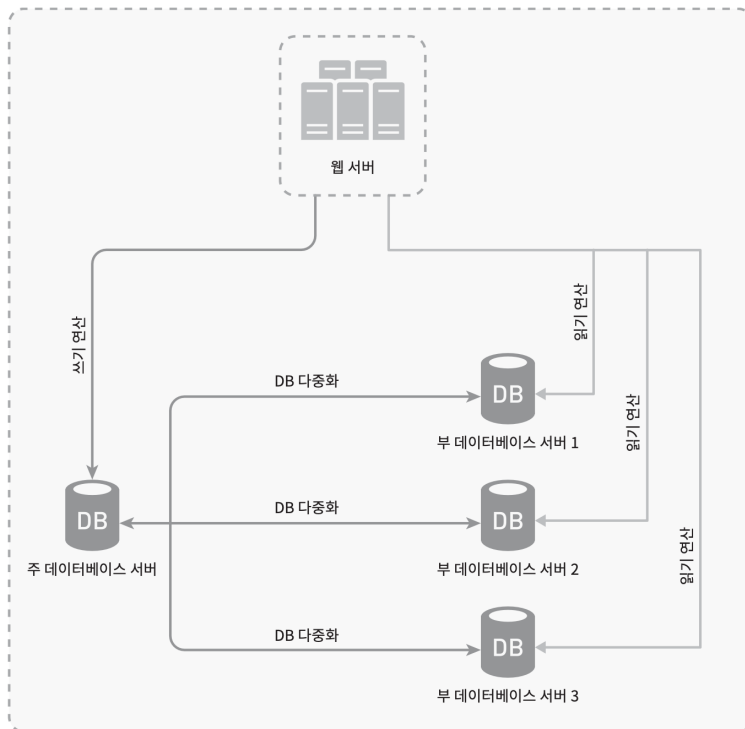


그림 1-5

## 데이터베이스 다중화의 장점

- 더 나은 성능(more performance)
  - 주-부 다중화 모델에서 모든 데이터 변경 연산은 주 데이터베이스 서버로만 전달되는 반면,  
 읽기 연산은 부 데이터베이스 서버들로 분산된다.
  - 이렇게 될 경우 병렬로 처리될 수 있는 쿼리의 수가 늘어나므로, 성능이 좋아진다.

- **안정성(reliability)**

- 데이터를 지역적으로 떨어진 여러 장소에 다중화를 시켜 놓을 수 있으므로  
자연 재해 등의 이유로

**데이터베이스 서버 가운데 일부가 파괴되어도 데이터는 보존할 수 있다.**

- **가용성(availability)**

- 데이터를 여러 지역에 복제해 둬으로써, 하나의 데이터베이스 서버에 장애가 발생  
하더라도

**다른 서버에 있는 데이터를 가져와 계속 서비스할 수 있다.**

로드 밸런서가 시스템 가용성을 높이는 상황처럼 데이터베이스에도 같은 질문을 던져보자.

**데이터베이스 서버 가운데 하나가 다운되면 무슨 일이 벌어지는가?**

- 부 서버가 한 대 뿐인데 다운된 경우라면, 읽기 연산은 한시적으로 모두 **주 데이터베이스로 전달** 된다.

또한 즉시

**새로운 부 데이터베이스 서버가 장애 서버를 대체할 것이다.**

**부 서버가 여러 대인** 경우에 읽기 연산은 나머지 부 데이터베이스 서버들로 분산될 것이며,

**새로운 부 데이터베이스 서버가 장애 서버를 대체할 것이다.**

- 하나의 주-부 서버들로 이루어 졌을 때 **주 서버가 죽으면** 부 서버가 주 서버가 되고, 모든 데이터베이스 연산은 일시적으로 새로운 주 서버상에서 수행된다. 이후 새로운 부 서버가 추가될 것이다.

**프로덕션 환경에서는 부 서버에 보관된 데이터가 최신 상태가 아닐 수 있어 조금 더 복잡**

- 데이터의 싱크가 맞지 않다면 없는 데이터는 복구 스크립트(recovery script)를 정합성을 맞춰야 됨
- 다중 마스터(multi-master)나 원형 다중화(circular replication) 방식을 도입하면 이런 상황에 대처하는 데 도움이 될 수 있지만, 해당 구성은 훨씬 복잡하며 그와 관련된

내용은 참고 문헌 [4], [5] 를 살펴보자.

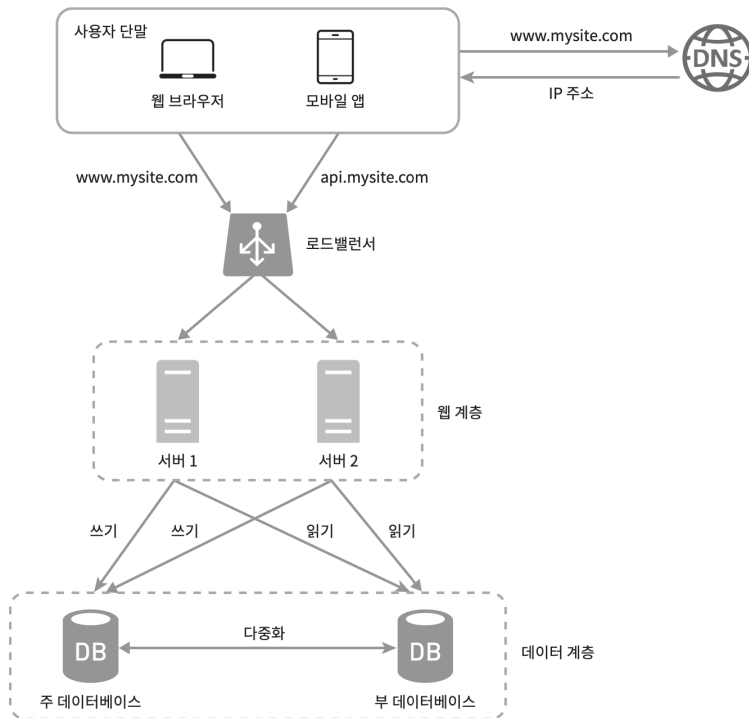


그림 1-6

위는 다음과 같이 동작

- 사용자는 DNS로부터 로드밸런서의 공개 IP 주소를 받음
- 사용자는 해당 IP 주소를 사용해 로드밸런서에 접속
- HTTP 요청은 서버 1이나 서버 2로 전달
- 웹 서버는 사용자의 데이터를 부 데이터베이스 서버에서 읽는다.
- 웹 서버는 데이터 변경 연산은 주 데이터베이스로 전달한다.

데이터

추가, 삭제, 갱신 연산 등등

웹 계층과, 데이터 계층에 대해 이해했으므로 응답 시간(latency)을 개선해보자.



응답 시간은 캐시를 붙이고, 정적 콘텐츠를 **콘텐츠 전용 네트워크(Content Delivery Network, CDN)**로 옮기자

## 캐시 (Cache) (중요\*)

- 값비싼 연산 결과 또는 자주 참조되는 데이터를 인메모리에 두고, 뒤이은 요청보다 빨리 처리할 수 있도록 하는 저장소.
- 애플리케이션 성능은 데이터베이스를 얼마나 자주 호출하느냐에 크게 좌우된다.
  - 캐시는 이 문제를 효과적으로 해결

## 캐시 계층(cache tier)

데이터가 잠시 보관되는 곳으로 데이터베이스보다 훨씬 빠르다.

또한, 별도의 캐시 계층을 두면 성능이 개선될 뿐 아니라

**데이터베이스의 부하를 줄일 수 있고,**

**캐시 계층의 규모를 독립적으로 확장**시키는 것도 가능해진다.

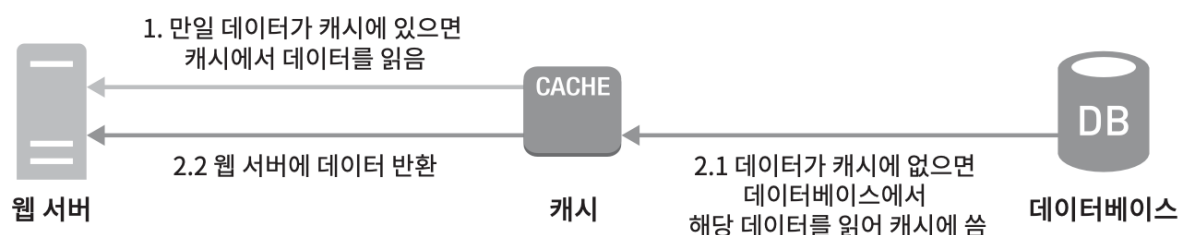


그림 1-7

## 읽기 주도형 캐시 전략(read-through caching strategy)

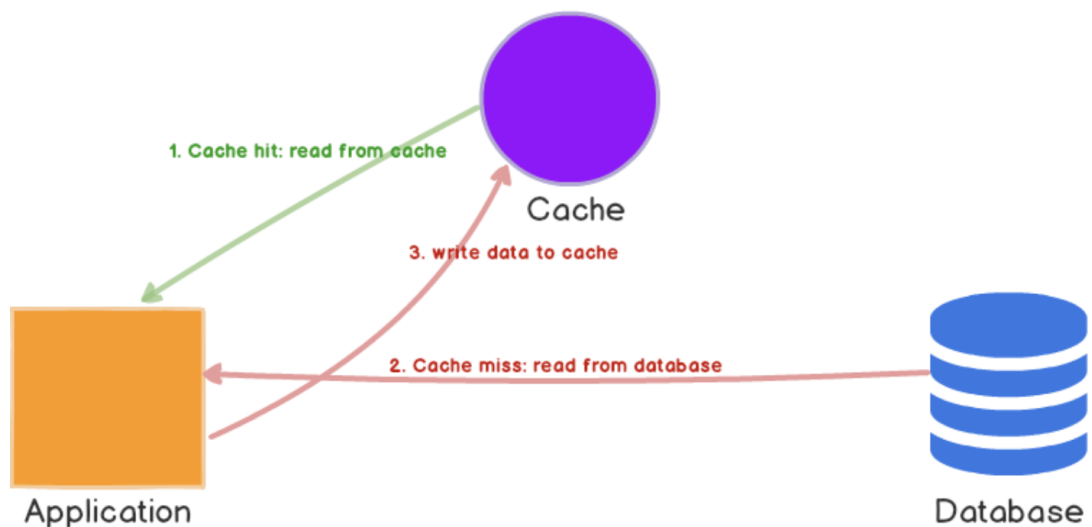
- 요청을 받은 웹 서버는 캐시에 응답이 저장되어 있는지 확인한다.
  - 저장되어 있다면 : 해당 데이터를 클라이언트 반환
  - 저장되지 않았다면 : 데이터베이스 조회 후 데이터를 찾아 캐시에 저장한 뒤 클라이언트 반환

읽기 주도형 캐시 전략 외에도 다양한 캐시 전략이 존재한다.  
캐시할 데이터의 종류, 크기, 액세스 패턴에 맞는 캐시 전략을 선택하면 된다.  
이와 관련된 비교 연구 결과는 다음을 참고하자. - [  
참고 레퍼런스 6번 확인]

▼ Caching Strategies and How to Choose the Right One - 내용 좋음

▼ **Cache-Aside**

Cache-Aside

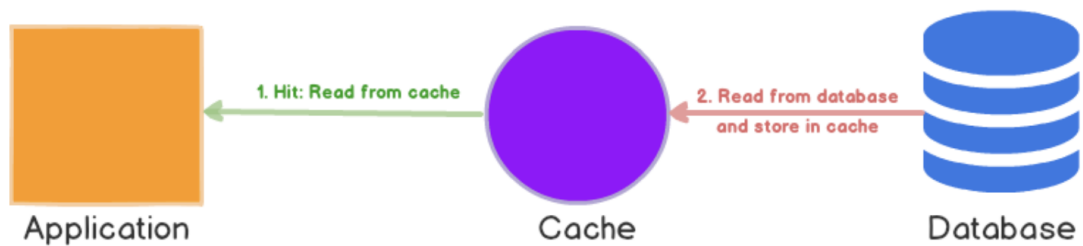


가장 일반적인 사용 방식

- 읽기 중심 워크로드에 가장 적합
- Memcached와 Redis가 널리 사용
- 캐시 오류에 대한 회복성이 뛰어남
- 캐시 클러스터가 다운되면 시스템은 데이터베이스를 조회하여 장애가 날 확률이 적음  
(부하가 피크면 엄청 느린 응답시간 + 장애 발생 가능)

▼ **Read-Through**

## Read-Through

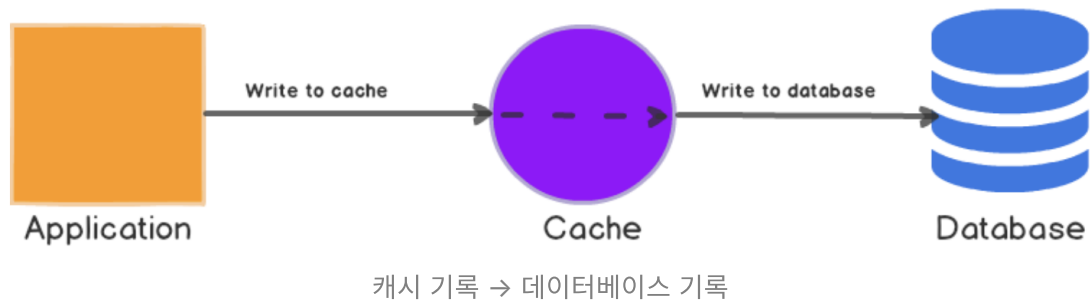


데이터베이스와 일렬로 배치

- 캐시 미스가 발생하면 데이터베이스에서 누락된 데이터를 로드하고 캐시를 채우고, 애플리케이션 반환
- Cache Aside와 Read Through 두 전략 모두 데이터를 느리게 로드한다. 즉, 먼저 읽을 때만 데이터를 로드한다.
- cache aside와 두 가지 차이점
  1. cache aside에서는 애플리케이션이 데이터베이스에서 데이터를 가져와 캐시를 채우는 역할을 담당  
read through에서 로직은 일반적으로 라이브러리 또는 독립형 캐시 공급자가 지원한다.
  2. cache aside와 달리 read through 캐시의 데이터 모델은 데이터베이스의 데이터 모델과 다를 수 없다.
- Read-Through는 **동일한 데이터를 여러 번 요청하는 read-heavy 워크로드에 가장 적합하다.**
  - ex) 뉴스 기사와 같은 것들
- 단점은 데이터를 처음 요청할 때 항상 캐시 미스가 발생하고, 캐시에 데이터를 로드하는 데 추가 비용이 발생한다.
  - 개발자가 수동으로 쿼리를 실행하는 warm up 할 수 있음.
- cache aside와 같이 캐시와 데이터베이스 간 데이터의 불일치가 발생할 수 있음
  - 불일치 문제는 쓰기 전략으로 해결 가능

## ▼ Write-Through

## Write-Through



- 데이터가 **먼저 캐시에 쓰인 다음 데이터베이스 기록**한다.
  1. 애플리케이션이 데이터를 캐시에 직접 쓴다.
  2. 캐시는 기본 데이터베이스의 데이터를 업데이트 한다.  
쓰기가 완료되면 캐시와 데이터베이스 모두 동일한 값을 가지며  
**캐시는 항상 일관성을 유지**한다.
- Write-Through는 데이터를 **캐시에 쓰고, 데이터베이스에 쓰기** 때문에 이 두 번의 쓰기 작업 때문에 추가적인 **쓰기 지연 시간을 유발**한다.  
그러나 Read-Through 캐시와 함께 사용하면 Read-Through의 모든 이점을 누릴 수 있고, 데이터 정합성에 대한 보장도 얻을 수 있어 캐시 무효화를 사용하지 않아도 된다.  
(데이터베이스에 대한 모든 쓰기가 캐시를 통과한다고 가정할 때)
- AynamoDB Accelerator(DAX)는 읽기/쓰기 캐시의 좋은 예이다.

### ▼ Write-Around

- 데이터가 데이터베이스에 **직접** 쓰여지고 읽은 데이터만 캐시로 이동한다.
- Write-Around는 Read-Through 와 함께 사용될 수 있으며 한 번 쓰고 읽은 빈도가 낮거나 아예 읽지 않는 상황일 때 좋은 성능을 제공한다.
  - ex) 실시간 로그, 채팅방 메시지 같은 것들
- **Write-Around + Cache-Aside** 형태로 결합 가능

### ▼ Write-Back or Write-Behind

애플리케이션은 데이터를 저장하는 **캐시에 데이터를 쓰고, 애플리케이션에 즉시 승인**한다.

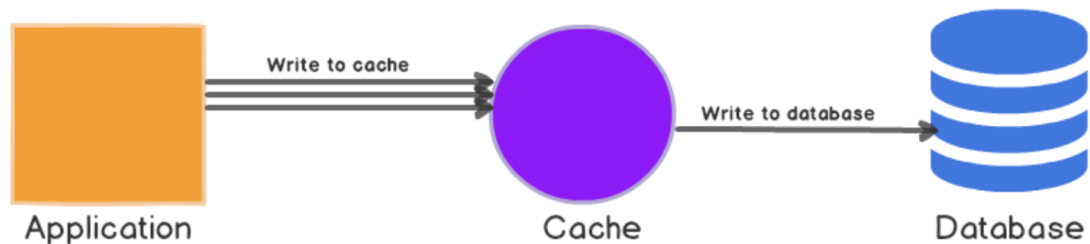
그 후 캐시에 데이터를 데이터베이스에 쓴다.

Write-Through와 비슷하지만 차이점은 Write-Through에서는 캐시에 쓰여진 데이터가 메인 데이터베이스에 동시적으로 업데이트 된다.

하지만, Write-Back은 캐시에 쓰여진 데이터가 주 데이터베이스에서 비동기적으로 업데이트 된다.

애플리케이션 관점에서 보면 응답을 반환하기 전 캐시만 업데이트하면 되므로 Write-Back 캐시에 대한 쓰기가 더 빠르다.

## Write-Back



write-behind 라고도 함

- 쓰기가 많은 워크로드에 적합하다.
- Read-Through 와 함께 사용하면 가장 최근에 업데이트되고, 액세스된 데이터를 항상 캐시에서 사용할 수 있는 혼합 워크로드에 적합하다.
- 데이터베이스 장애에 대한 복원력이 뛰어나며 약간의 데이터베이스 다운 타임을 견딜 수 있다.
- 일괄 처리 또는 병합이 지원되는 경우 데이터베이스에 대한 전체 쓰기 횟수를 줄여 부하를 줄이고, 데이터베이스 공급자가 요청 수에 따라 요금을 부과하는(ex, DynamoDB) 경우 비용을 아낄 수 있다.
- DAX는 쓰기 방식이므로 애플리케이션이 쓰기량이 많은 경우 비용 절감 효과를 볼 수 없다.

일부 개발자는 최대 부하 시 스파이크를 더 잘 흡수하기 위해 Cache-Aside + Write-Back 모두 Redis를 사용한다. 여기서 가장 큰 단점은 캐시 장애가 발생하면 데이터가 영구적으로 손실될 수 있다.

대부분의 관계형 데이터베이스 스토리지 엔진(ex, InnoDB)은 내부에 기본적으로 Write-Back Cache가 활성화 되어 있다. 쿼리는 먼저 메모리에 기록된 후 최종적으로 디스크에 플러시 된다.

## 캐시 사용 시 유의할 점 (중요\*)

다음의 사항들을 고려해야 한다.

- ▼ 캐시는 어떤 상황에 바람직한가?

- 데이터 갱신은 자주 일어나지 않지만 참조는 빈번하게 일어난다면 고려해볼 만하다.

▼ 어떤 데이터를 캐시에 두어야 하는가?

- 캐시는 데이터를 휘발성 메모리에 두므로, 영속적으로 보관할 데이터를 캐시에 두는 것은 좋지 않다.
- ex) 캐시 서버가 재시작되면 캐시 내의 모든 데이터는 사라진다. (redis 는?)

▼ 캐시에 보관된 데이터는 어떻게 만료(expire)되는가?

- 만료 정책이 없으면 데이터는 캐시에 계속 남게 된다.
- 그렇다고 만료 기한이 짧으면 데이터베이스를 자주 읽게 된다.
- 그렇다고 또 너무 길면 데이터베이스 원본과 차이가 날 가능성이 높다.
- 따라서 만료 정책을 어떻게 잡을지 고민이다. (이에 대해 고민하는 것은 좋은 습관)

▼ 일관성(consistency)은 어떻게 유지되는가?

일관성 : 데이터 저장소의 원본과 캐시 내의 사본이 같은지 여부

- 저장소의 원본을 갱신하는 연산과 캐시를 갱신하는 연산이 단일 트랜잭션으로 처리되지 않는 경우  
일관성이 깨질 수 있다.
- 여러 지역에 걸쳐 시스템을 확장해 나가는 경우 캐시와 저장소 사이의 일관성을 유지하는 것은 어렵다.  
이에 관해서 페이스북에서 제시한 논문을 참고하자. ([참고 레퍼런스 7번](#))

▼ 장애에는 어떻게 대처할 것인가?

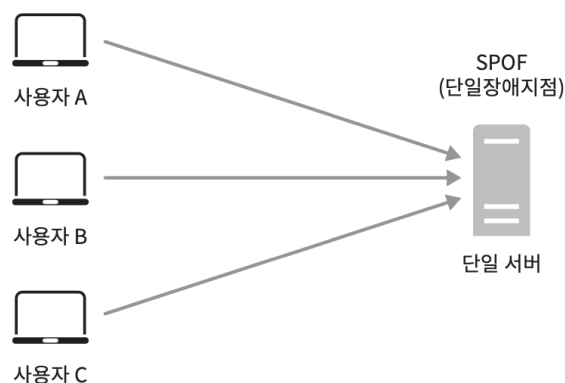


그림 1-8

- 캐시 서버를 한 대만 두는 경우 해당 서버가 단일 장애 지점(Single Point of Failure, SPOF)이 될 가능성이 있다. (단일 장애 지점 : 특정 지점에서의 장애가 전체 시스템의 동작을 중단시킬 수 있는 경우)
- 결과적으로 SPOF를 피하려면 여러 지역에 걸쳐 캐시 서버를 분산시켜야 한다.

#### ▼ 캐시 메모리는 얼마나 크게 잡을 것인가?

- **캐시 메모리가 너무 작으면** 캐시에 데이터를 저장하는 액세스 패턴에 따라 (ex, Write-Through?)

데이터가 너무 자주 캐시에서 밀려나버려(eviction) **캐시의 성능이 떨어진다.**  
이를 막을 한 가지 방법으로  
**캐시 메모리를 과할당(overprovision)**하는 것이다.

- 캐시 메모리를 과할당하면 캐시에 보관할 데이터가 갑자기 늘어났을 때 생기는 문제도 방지할 수 있다.

#### ▼ 데이터 방출(eviction) 정책은 무엇인가?

- 캐시가 다 차게되면 추가로 캐시에 데이터를 넣어야 할 경우 기존 데이터를 내보내야 된다.  
이를 캐시 데이터 방출 정책이라 하는데, 그 가운데 가장 널리 쓰이는 것은 LRU이다.
  - 다음과 같이 다른 정책들도 존재 (운영체제의 메모리 관리와 같음)
  - **LRU(Least Recently Used)** - 마지막으로 사용된 시점이 가장 오래된 데이터를 내보내는 정책
  - **LFU(Least Frequently Used)** - 사용된 빈도가 가장 낮은 데이터를 내보내는 정책
  - **FIFO(First In First Out)** - 가장 먼저 캐시에 들어온 데이터를 가장 먼저 내보내는 정책

## 콘텐츠 전송 네트워크(CDN)

**정적 콘텐츠**를 전송하는 데 쓰이는, **지리적으로 분산된 서버의 네트워크**이다.

- 이미지, 비디오, CSS, JavaScript 파일 등을 캐시할 수 있다.
- 이 책에서는 동적 콘텐츠에 대해서는 다음과 같은 간단한 설명만 있음

- 요청 경로(request path), 질의 문자열(query string), 쿠키(cookie), 요청 헤더(request header) 등의 정보를 기반으로 **HTML 페이지를 캐시**하는 것이다.
- 이에 대해서 자세 알고 싶다면 레퍼런스 9번을 참고하자.

어떤 사용자가 웹 사이트를 방문하면, 해당 사용자에게 가장 가까운 CDN 서버가 정적 콘텐츠 전달

CDN 서버로부터 멀면 멀수록 웹사이트는 천천히 로드된다.

- ex) CDN 서버가 샌프란시스코에 있다면 LA에 있는 사용자는 유럽 사용자보다 빠른 웹 사이트 경험

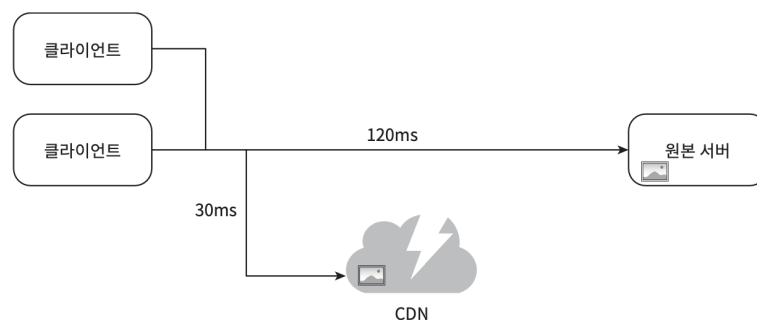


그림 1-9

물리적인 거리가 가까울 수록 더 빠름

## CDN이 어떻게 동작하는지 간단하게 살펴보기

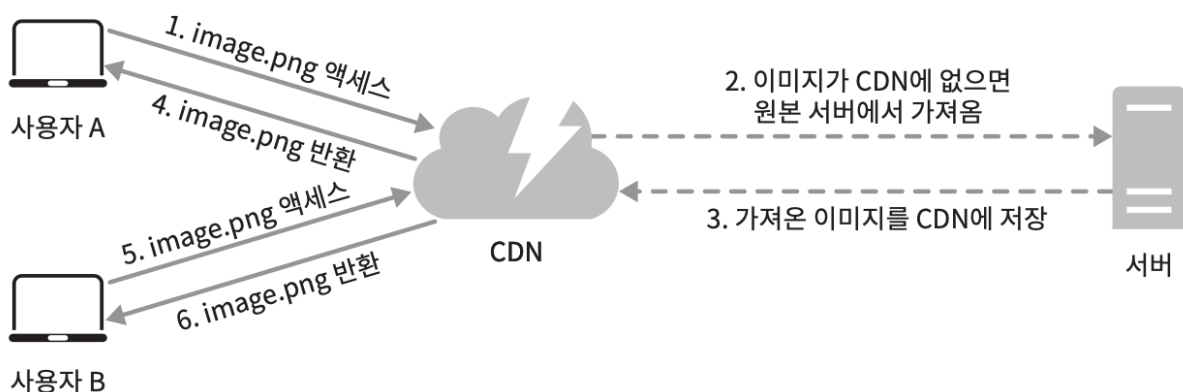


그림 1-10

CDN이 어떻게 동작하는지 설명하는 그림



## 1. 사용자 A가 이미지 URL을 이용해 image.png에 접근

- URL의 도메인은 CDN 서비스 사업자가 제공한 것
  - 다음 두 개는 클라우드 프론트와 아카마이 CDN이 제공하는 URL 예시
  - `https://mysite.cloudfront.hello.jpg`
  - `https://mysite.akamai.com/image-manager/img/logo.jpg`

## 2. CDN 서버의 캐시가 해당 이미지가 없는 경우, 서버는 원본(origin) 서버에 요청하여 파일을 가져옴

- 원본 서버는 웹 서버일 수도 있고, S3 같은 온라인 저장소 일 수 있음

## 3. 원본 서버가 파일을 CDN 서버에 반환

- 응답의 HTTP 헤더에는 해당 파일이 얼마나 오래 캐시될 수 있는지를 설명하는 TTL 값이 들어 있음
  - TTL : Time To Live의 약자로 파일이 얼마나 살아있을 수 있는지에 대한 정보

## 4. CDN 서버는 파일을 캐시하고 사용자 A에게 반환한다. 이미지는 TTL에 명시된 시간이 끝날 때까지 캐시

## 5. 사용자 B가 같은 이미지에 대한 요청을 CDN 서버에 전송

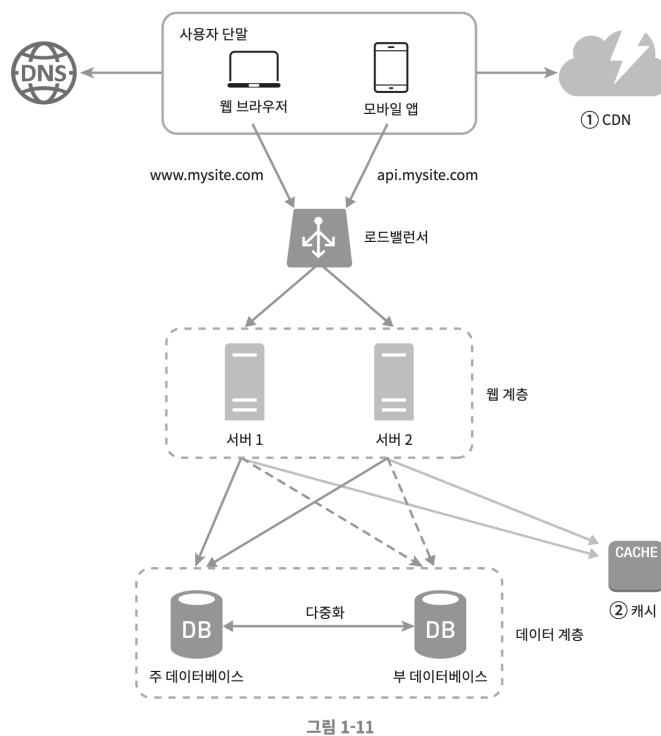
## 6. 만료되지 않은 이미지에 대한 요청은 캐시를 통해 처리

# CDN 사용 시 고려해야 할 사항

- **비용**
  - 보통 제 3 사업자에 의해 운영되며, 들어오고 나가는 데이터 전송 양에 따라 요금을 낸다.
  - 자주 사용되지 않는 콘텐츠는 캐싱의 의미가 없으므로 빠자.
- **적절한 만료 시간 설정**
  - 시간에 민감한(time-sensitive) 콘텐츠의 경우 만료 시점을 잘 정해야 된다. 너무 길지도, 짧지도 않아야 된다.
  - 너무 길면 콘텐츠의 신선도는 떨어지고, 너무 짧으면 원본 서버에 빈번히 접속하게 된다.
- **CDN 장애에 대한 대처 방안**
  - CDN 자체가 죽었을 때 웹/앱이 어떻게 동작해야 할까?

- 일시적으로 CDN이 응답하지 않는 경우, 해당 문제를 감지해 원본 서버로부터 직접 콘텐츠를 가져오도록 클라이언트를 구성하는 것이 필요할 수도 있다.
- 콘텐츠 무효화(invalidation) 방법
  - 아직 만료되지 않는 콘텐츠라 하더라도 아래 방법 가운데 한 가지를 쓰면 CDN에서 제거할 수 있다.
    1. CDN 서비스 사업자가 제공하는 API를 이용하여 콘텐츠 무효화
    2. 콘텐츠의 다른 버전을 서비스하도록 오브젝트 버저닝(object versionning)이  
용  
콘텐츠의 새로운 버전을 지정하기 위해서는 URL 마지막에 버전 번호를 인자로  
주면 된다.  
ex)

`image.png?v=2` 이런 형태



## 달라진 부분

1. 정적 콘텐츠(JS, CSS, 이미지 등)는 더 이상 웹 서버를 통해 서비스하지 않으며, CDN을 통해 제공하여 더 나은 성능을 보장
2. 캐시가 데이터베이스 부하를 줄여준다.

## 무상태(stateless) 웹 계층

웹 계층을 수평적으로 확장하는 방법을 고민한다.

이를 위해

**상태 정보(사용자 세션 데이터와 같은)를 웹 계층에서 제거한다.**

좋은 전략은 상태 정보를 관계형 데이터베이스나 NoSQL 같은 지속성 저장소에 보관하고, 필요할 때 가져온다.

이렇게 구성된

**웹 계층을 무상태 웹 개층**이라 부른다.

## 상태 정보 의존적인 아키텍처

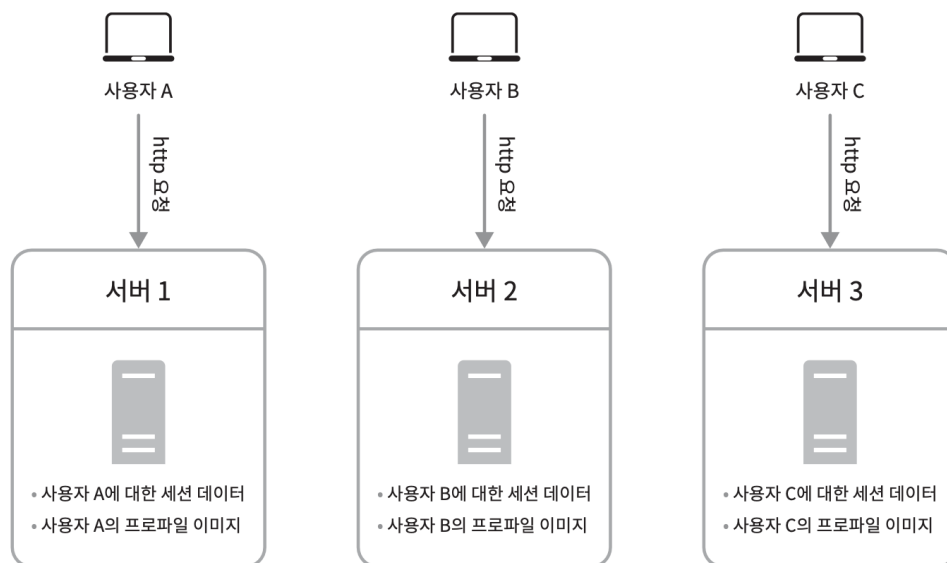


그림 1-12

상태 정보 의존적인 아키텍처

- 사용자 A가 서버 2로 요청을 보내면 **인증은 실패**한다. 다른 경우도 마찬가지로, **같은 클라이언트로부터의 요청은 항상 같은 서버로 전송**되어야 함
  - 이런 경우를 대비해 대부분의 로드밸런서가 **고정 세션(sticky session)**이라는 기능을 제공한다. (참고)
  - 하지만 고정 세션은 로드 밸런서에 부담을 준다.

- 또한 로드 밸런서 뒷단에 서버를 추가하거나 제거하기도 까다롭고, 서버의 장애를 처리하기도 복잡해짐

## 무상태 아키텍처

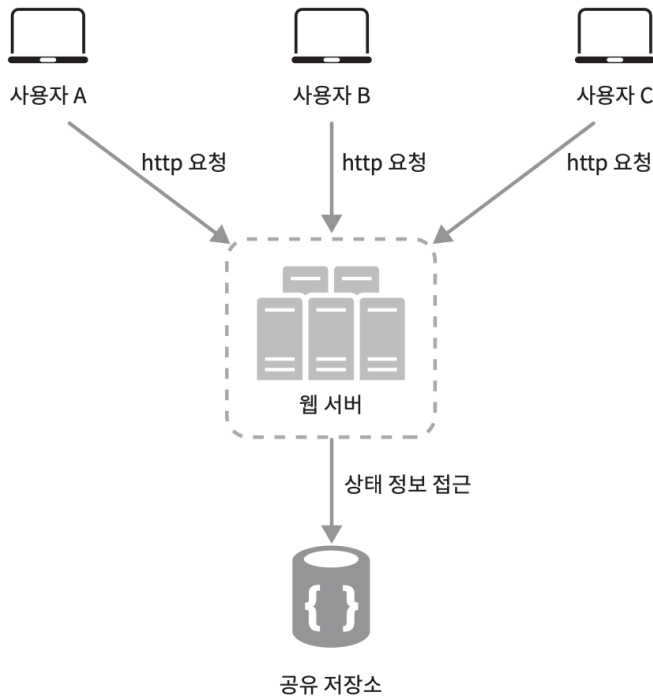


그림 1-13

무상태 아키텍처

이 구조라면 사용자의 HTTP 요청은 어느 서버로든 보낼 수 있다.

**웹 서버는 상태 정보가 필요할 경우 공유 저장소(shared storage)로부터 데이터를 가져온다.**

즉,

상태정보는 웹 서버로부터 물리적으로 분리되어 있다.

이런 구조는 단순하고, 안정적이며, 규모 확장이 쉽다.

여기까지 정보를 반영한 설계 사진

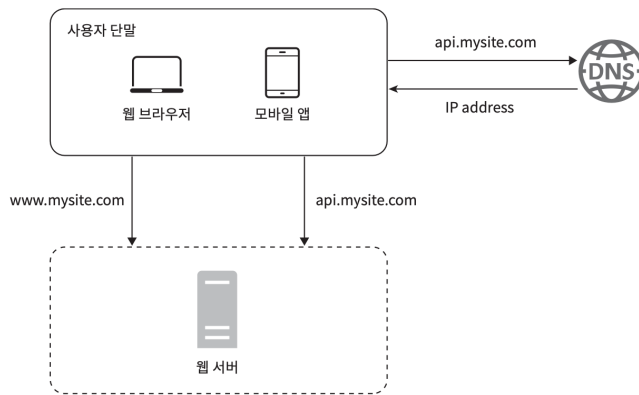


그림 1-1

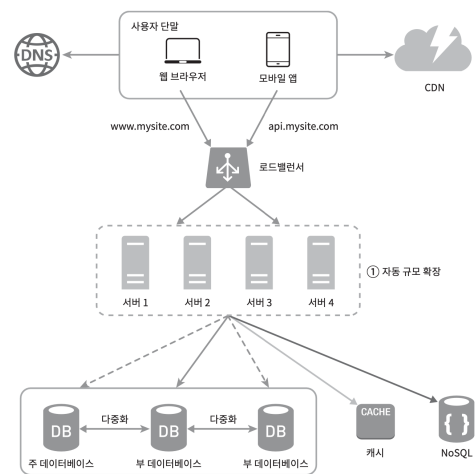


그림 1-14

오른쪽 그림에서 세션 데이터를 웹 계층에서 분리하고 지속성 데이터 보관소에 저장하도록 만들었다.

이

공유 저장소는 관계형 데이터베이스일 수도 있고, Memcached/Redis 같은 캐시 시스템일 수 있고,

NoSQL일 수도 있다. → 어떤 것을 사용해도 괜찮다. 상태 정보를 글로벌로 관리할 수 만 있으면 알맞게 사용 가능

오른쪽 그림을 보면 NoSQL로 관리했는데, 이유는 규모 확장이 편해서라고 한다.

(1)은 웹 서버들을 트래픽의 규모에 따라 자동 규모 확장(autoscaling)을 할 수 있다. → autoscaling vs cache

이제 전 세계 사용자의 이목을 받는 서비스라고 생각해보자.

가용성을 높이고, 전 세계 어디서도 쾌적하게 사용하게 하려면 여러 데이터 센터(data center)를 지원해야 된다.

## 데이터 센터(Data Center)

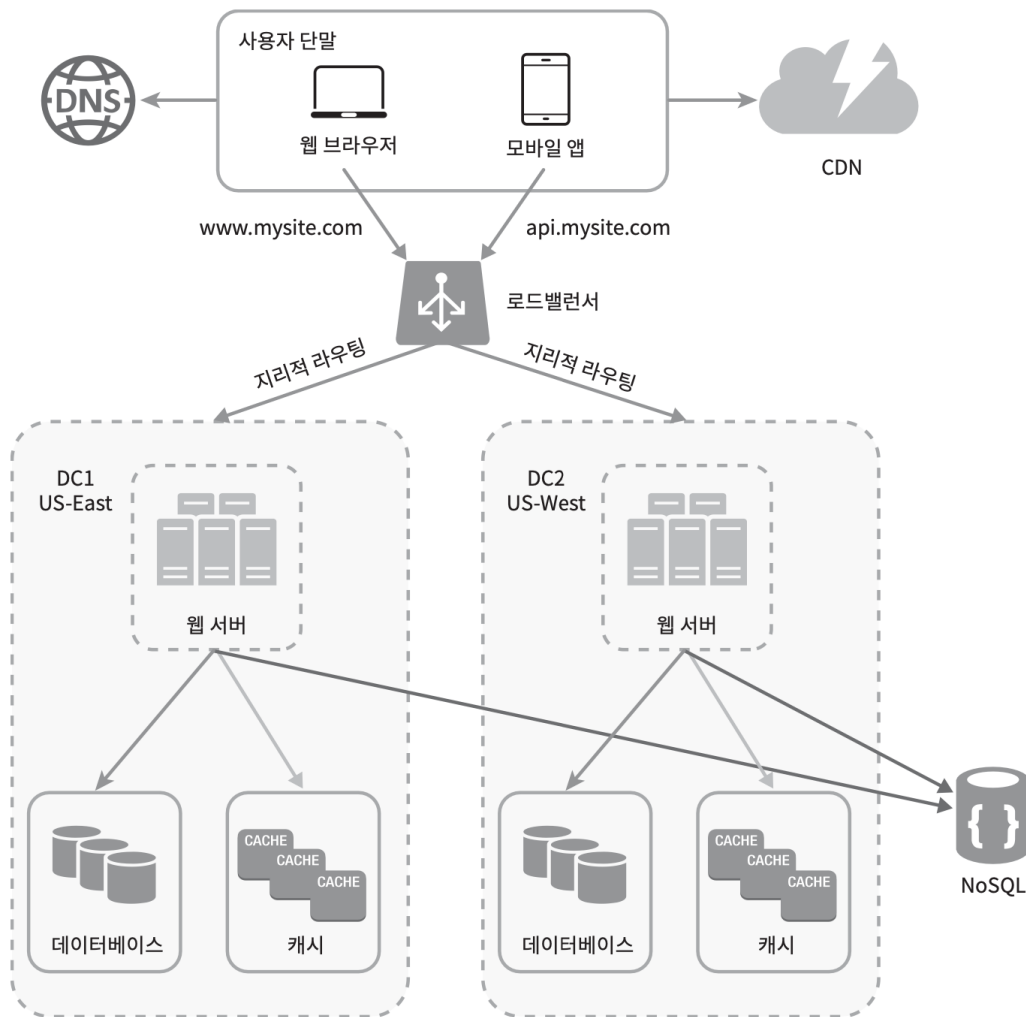


그림 1-15

두 개의 데이터 센터를 이용하는 사례

**장애가 없는 상황에서** 사용자는 **가장 가까운 데이터 센터로 안내된다.**

이 절차를

**지리적 라우팅(geoDNS-routing 또는 geo-routing)**이라고 부른다.

지리적 라우팅에서의 **geoDNS**는 **사용자의 위치에 따라**

**도메인 이름을 어떤 IP 주소로 변환할지 결정**

할 수 있도록 해주는 DNS 서비스이다.

**x%** 사용자는 US-East 센터로, **(100-x)%**의 사용자는 US-West 센터로 간다고 가정해보자.

이들 데이터 센터 중 하나에 심각한 장애가 발생하면 모든 트래픽은 장애가 없는 데이터 센터로 전송된다.

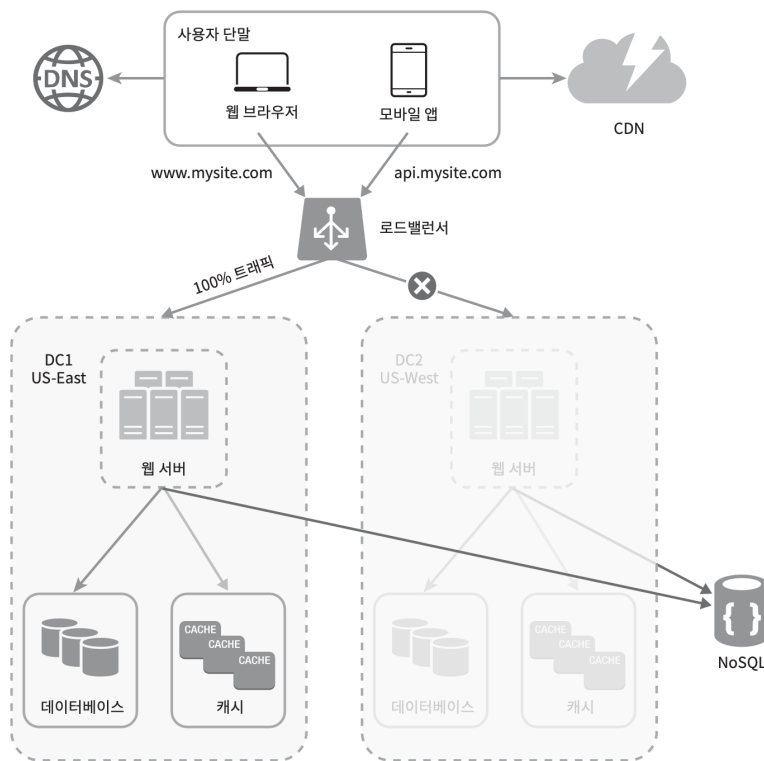


그림 1-16

West 데이터 센터가 장애가 발생하면 East 데이터 센터가 다 책임

이러한 상황 같이 **다중 데이터센터 아키텍처를 만들려면 몇 가지 기술적 난제를 해결해야 한다.**

### • 트래픽 우회

- 올바른 데이터 센터로 트래픽을 보내는 효과적인 방법을 찾아야 함
- GeoDNS는 사용자에게서 가장 가까운 데이터센터로 트래픽을 보낼 수 있도록 해 준다.

### • 데이터 동기화(synchronization)

- 데이터 센터마다 별도의 데이터베이스를 사용하고 있다면, 장애가 자동으로 복구되어(failover) 트래픽이 다른 데이터베이스로 우회된다고 해도, **해당 데이터센터에는 찾는 데이터가 없을 수 있다.**

**이런 상황을 막는 보편적인 전략은 데이터를 여러 데이터 센터에 다중화하는 것이다.**

- 넷플릭스가 여러 데이터센터에 걸쳐 데이터를 어떻게 다중화하는지를 보려면 11번을 읽어보자.

- **테스트와 배포(deployment)**

- 여러 데이터 센터를 사용하도록 시스템이 구성된 상황이라면 웹 사이트 또는 애플리케이션을 여러 위치에서 테스트해보는 것도 중요하다.
- **자동화된 배포 도구는 모든 데이터 센터에 동일한 서비스가 설치되도록 하는 데 중요한 역할을 한다. (11)**

시스템을 큰 규모로 확장하기 위해서는 시스템의 **컴포넌트를 분리하여, 각기 독립적으로 확장**할 수 있게 해야 된다.

**메시지 큐**는 많은 실제 분산 시스템이 이 문제를 풀기 위해 채용하고 있는 핵심적 전략 가운데 하나다.

---

## 메시지 큐(Message Queue)

메시지 큐는 **메시지의 무손실(durability, 메시지 큐에 일단 보관된 데이터는 소비자가 꺼낼 때까지 완전히 보관된다는 특성)을 보장하는 비동기 통신(asynchronous communication)을 지원하는 컴포넌트**이다.

메시지의 버퍼 역할을 하며, 비동기적으로 전송한다.

메시지 큐의 기본 아키텍처는 간단하다.

**생산자 또는 발행자(producer/publisher)**라고 불리는 입력 서비스가

**메시지를 만들어 메시지 큐에 발행(publish)**한다.

큐에는 보통

**소비자 혹은 구독자(consumer/subscriber)**라 불리는 서비스 혹은 서버가 연결 되는데,

**메시지를 받아 그에 맞는 동작을 수행하는 역할을 한다. (다음 그림을 참고)**



그림 1-17



메시지 큐를 이용하면 서비스 또는 서버 간 결합이 느슨해져서,  
규모 확장성이 보장되어야 하는  
안정적 애플리케이션을 구성하는데 좋다.

생산자는 소비자 프로세스가 다운되어 있어도 메시지를 발행할 수 있고,

소비자는 생산자 서비스가 가용한 상태가 아니더라도 메시지를 수신할 수 있다.

**사용 예 - 이미지의 크로핑, 샤프닝, 블러핑 등을 지원하는 사진 보정 애플리케이션을 만드는 상황**



그림 1-18

이러한 보정은 시간이 오래 걸릴 수 있는 프로세스이므로 비동기로 개발했을 때 편하다.

다음 그림처럼 웹 서버는 사진 보정 작업(job)을 메시지 큐에 넣고,

사진 보정 작업(worker) 프로세스들은 이 작업을 메시지 큐에서 꺼내 비동기적으로 완료한다.

이렇게 하면 생산자 소비자 서비스의 규모는 각기 독립적으로 확장할 수 있다.

큐의 크기가 커지면 더 많은 작업 프로세스를 추가해 처리 시간을 주리고, 큐가 거의 항상 비어 있는 상태라면,  
작업 프로세스의 수는 줄일 수 있다.

## 로그, 메트릭 그리고 자동화

몇 개 서버에서 실행되는 소규모 웹 사이트를 만들 때는 로그나 메트릭(metric), 자동화(automation) 같은 것은 하면 좋지만, 필수는 아니다. 하지만 웹 사이트와 사업 규모가 커지면 해당 도구에 필수적으로 투자해야 한다.

- **로그 (시스템의 오류와 문제들을 보다 쉽게 찾아낼 수 있도록 도와줌)**

- 에러 로그는 서버 단위로 모니터링 할 수도 있지만, 로그를 단일 서비스로 모아주는 도구를 활용하면 더 편리하게 검색하고 조회할 수 있다.

- **메트릭**

- 메트릭을 잘 수집하면 사업 현황에 관한 정보를 얻을 수 있고, 시스템의 현재 상태를 쉽게 파악할 수 있다.
- 다음은 메트릭 중에서 특히 유용한 것들이다.
  - **호스트 단위 메트릭** : CPU, 메모리, 디스크 I/O에 관한 메트릭이 여기 해당
  - **종합(aggregated) 메트릭** : 데이터베이스 계층의 성능, 캐시 계층의 성능 같은 것들이 해당
  - **핵심 비즈니스 메트릭** : 일별 능동 사용자(DAU, daily active user), 수익(revenue), 재방문(retention) 같은 것들이 여기 해당한다.

- **자동화**

- 시스템이 크고 복잡해지면 생산성을 높이기 위해 자동화 도구를 활용해야 한다.
  - ex) **지속적 통합(CI, continuous integration)**을 도와주는 도구 같은 것들

## **메시지 큐, 로그, 메트릭, 자동화 등을 반영하여 수정한 설계안**

지면 관계상 하나의 데이터 센터만 그림에 포함했다고 함 (데이터 센터를 다중화할 수 있다고도 염두)

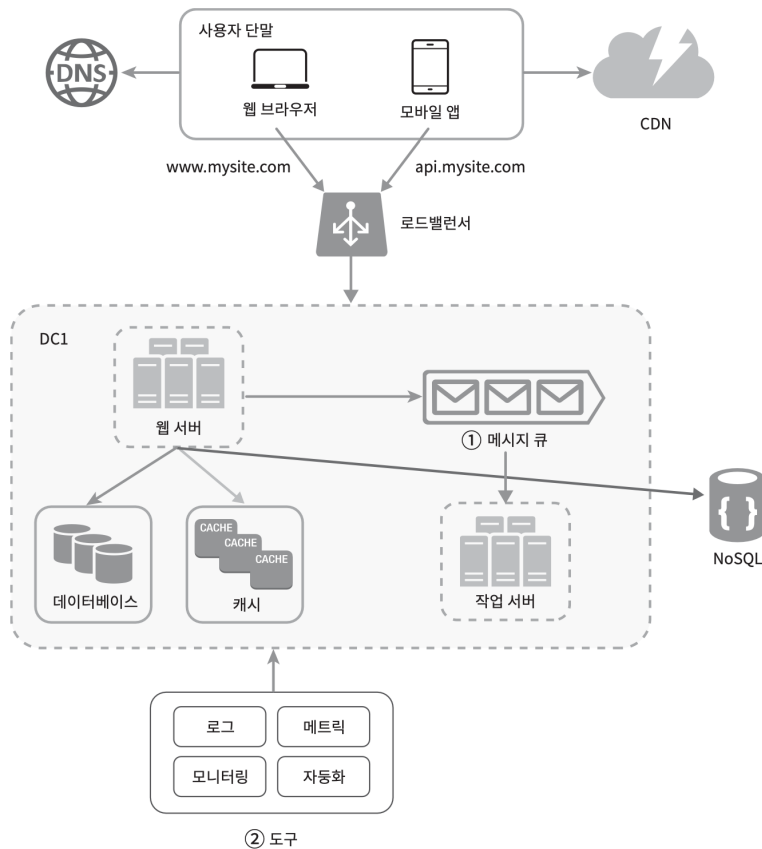


그림 1-19

1. 메시지 큐는 각 컴포넌트가 보다 느슨한 결합(loosely coupled)될 수도 있도록 하고, 결합에 대한 내성을 높인다.
2. 로그, 모니터링, 메트릭, 자동화 등을 지원하기 위한 장치를 추가하였다.

## 데이터베이스의 규모 확장

데이터베이스의 규모를 확장하는 두 가지 접근법

1. 수직적 확장, 스케일 업
2. 수평적 확장, 스케일 아웃

### 수직적 확장, 스케일 업

## 수직적 규모 확장 (CPU, RAM, 디스크 등 자원 증설)



기존 서버에 더 많은 또는 고성능의 자원(CPU, RAM, 디스크, 등)을 증설하는 방법  
ex) 아마존 AWS의 RDS(Relational Database Service)는  
**24TB RAM**을 갖춘 서버도 상품으로 제공

- 24TB의 메모리를 가진 EC2는 있는데, RDS는 안 보이는데.. 흠..

위와 같은 수준의 고성능 데이터베이스 서버를 사용하면 많은 양의 데이터를 보관하고 처리할 수 있다.

ex) 스택오버플로(stackoverflow)는 2013년 한 해 동안 방문한  
천만 명의 사용자를 전부 단 한대의 마스터 데이터베이스로 처리했음

### 수직적 향상에 치명적인 약점 (26p)

- 데이터베이스 서버 하드웨어에는 한계가 있으므로 CPU, RAM 등을 무한 증설할 수는 없다.
  - 사용자가 계속 늘어나면 한 대 서버로는 감당하기 어렵게 될 것이다.
- SPOF(Single Point of Failure)로 인한 위험성이 크다.
- 비용이 많이 든다. (고성능 서버로 갈수록 가격이 비싸짐)

- 만약 하드웨어 기술이 좋아져서(양자 컴퓨팅 같은) 하드웨어 자체 스펙이 엄청 좋아진다면?

## 수평적 확장, 스케일 아웃



데이터베이스의 수평적 확장은 **샤딩(sharding)**이라고도 부르는데, 더 많은 서버를 추가함으로써 성능을 향상시킬 수 있다.

**샤딩**은 대규모 데이터베이스를 **샤드(shard)**라고 부르는 작은 단위에 분할하는 기술을 일컫는다.

모든 샤드는 같은 스키마를 쓰지만 샤드에 보관되는 데이터 사이에는 중복이 없다.

- 샤딩과 파티셔닝은 무엇이 다르지? 스키마를 나눠서 사용하는 건가? 흠..

## 샤드로 분할된 데이터베이스의 예

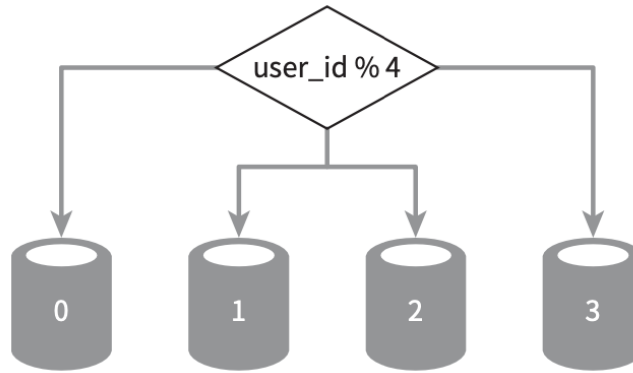


그림 1-21

사용자 데이터를 어느 샤드에 넣을지는 사용자 ID에 따라 정한다.

위 그림의 예는

`user_id % 4` 를 해시 함수로 사용하여 데이터가 보관되는 샤드를 정한다.

결과가 0이면 0번 샤드, 1이면 1번 샤드

- 굳이 이렇게 해야 되나? 라운드 로빈이랑 뭐가 다르지? 모듈러 연산의 비용이 비싸지 않나..? 단순히 % 4 로만 연산해서 괜찮나?

#### ▼ Claude 한테 물어본 결과

어느 샤드에 데이터를 저장할 지 결정하는 방식이 여러 가지가 존재하는 것으로 알고 있습니다.

이때 대부분 모듈러 연산을 해서 샤드를 정하는 것으로 알고 있습니다. 맞나요?

맞다면 왜 이 방식이 널리 사용되는지

다른 방식들과 비교해서 어떤 장단점이 있는지와 다른 방식들도 설명해주세요.

데이터 샤딩에서 모듈러 연산은 실제로 매우 일반적인 방식입니다. 다양한 샤딩 전략들을 비교하며 설명해드리겠습니다.

모듈러 해싱 방식의 주요 장점은 다음과 같습니다:

첫째, 결정적(deterministic) 특성을 가집니다. 동일한 키에 대해 항상 같은 샤드를 반환하므로 데이터 위치를 예측할 수 있고, 읽기 작업 시 어느 샤드를 조회해야 할지 즉시 알 수 있습니다.

둘째, 계산 비용이 매우 낮습니다. 단순한 산술 연산으로 구현되어 복잡한 로직이나 외부 상태 관리가 필요하지 않습니다.

셋째, 구현과 유지보수가 간단합니다. 복잡한 매핑 테이블이나 메타데이터 관리가 필요하지 않아 운영 복잡도가 낮습니다.

다른 주요 샤딩 전략들을 살펴보면:

범위 기반 샤딩(Range-based Sharding):

키의 범위를 기준으로 데이터를 분할합니다. 예를 들어, `user_id` 1-1000은 샤드1, 1001-2000은 샤드2와 같이 나눕니다. 연속된 데이터 조회에 유리하지만, 특정 범위에 데이터가 몰릴 수 있고 재샤딩이 복잡한 단점이 있습니다.

디렉토리 기반 샤딩(Directory-based Sharding):

룩업 테이블을 사용하여 키와 샤드를 매핑합니다. 유연한 데이터 분배가 가능하지만, 룩업 테이블 관리에 따른 오버헤드와 추가적인 조회 비용이 발생합니다.

일관된 해싱(Consistent Hashing):

해시 링을 사용하여 데이터를 분산합니다. 샤드 추가/제거 시 데이터 재배포를 최소화할 수 있는 장점이 있으나, 구현이 복잡하고 데이터 분포가 완전히 균등하지 않을 수 있습니다.

지리적 샤딩(Geographic Sharding):

사용자의 지리적 위치를 기준으로 데이터를 분산합니다. 지역별 접근 지연시간을 최적화할 수 있지만, 사용자 이동성 처리가 복잡하고 지역별 불균형이 발생할 수 있습니다.

이러한 맥락에서 모듈러 해싱이 널리 사용되는 이유는 구현 단순성, 예측 가능성, 낮은 연산 비용이라는 실용적인 이점들 때문입니다. 다만 완벽한 해결책은 아니며, 데이터 분포가 고르지 않을 수 있다는 한계가 있습니다.

실제 시스템 설계 시에는 워크로드 특성, 확장성 요구사항, 운영 복잡도 등을 종합적으로 고려하여 적절한 전략을 선택해야 합니다. 때로는 여러 전략을 조합하여 사용하기도 합니다. 예를 들어, 모듈러 해싱과 일관된 해싱을 함께 사용하여 균등한 분포와 효율적인 재샤딩을 모두 달성할 수 있습니다.

샤딩 전략을 구현할 때 가장 고려해야 할 중요한 것은 **샤딩 키(sharding key)**를 어떻게 정하느냐 하는 것이다.

**파티션 키(partition key)**라고도 부르며, 데이터가 어떻게 분산될 지 정하는 하나 이상의 컬럼으로 구성된다.

위에서 본 그림의 샤딩 키는 `user_id` 이다.

샤딩 키를 통해 올바른 데이터베이스에 질의를 보내, 데이터 조회나 변경을 처리하므로 효율을 높일 수 있다.

샤딩 키를 정할 때는 **데이터를 고르게 분할 할 수 있도록 하는 것이 가장 중요하다.**

샤딩은 데이터베이스 규모 확장을 실현하는 좋은 기술이지만, 완벽하진 않다.

샤딩을 도입하면 시스템이 복잡해지고  
**풀어야 할 새로운 문제**도 생긴다는 사실을 인지하고 도입하자.

## 샤딩을 사용할 때 고려할 사항

- 데이터 재 샤딩(resharding) 이 필요한 경우

1. 데이터가 너무 많아져서 하나의 샤드로는 더 이상 감당하기 어려울 때
2. 어떤 샤드에만 할당된 공간 소모가 다른 샤드에 비해 빨리 진행될 때
  - 이를 **샤드 소진(shard exhaustion)**이라고도 부르며  
이런 현상이 발생하면 샤드 키를 계산하는 함수를 변경하고 데이터를 재배포 해야 됨  
5장에서 다루는  
**안정 해시(consistent hashing) 기법을 활용하면 해결 가능**

- 유명인사(celebrity), 핫스팟 키(hotspot key) 문제

- 특정 샤드에 질의가 집중되어 서버에 과부하가 걸리는 문제이다.

저스틴 비버, 레이디 가가 등 유명 인사가 전부 같은 샤드에 저장되는 데이터베이스가 있다고 가정

이때 SNS 애플리케이션을 구축하게 되면 결국 해당 샤드는

**read** 연산 때문에 과부하가 오게 된다.

이 문제를 해결하려면 유명인사 각각에 샤드 하나씩을 할당해야 할 수도 있고, 심지어 더 잘게 쪼개야 할 수도..?

- 조인과 비정규화(join and de-normalization) 문제

- 일단 하나의 데이터베이스를 여러 샤드 서버로 쪼개고 나면, 여러 샤드에 걸친 데이터를 조인하기가 힘들어진다. 이를 해결하는 방법은 데이터베이스를 비정규화하여 하나의 테이블에서 질의가 수행될 수 있도록 만드는 것이다.

다음은 데이터베이스 샤딩을 적용한 아키텍처이다.

또한 데이터베이스 부하를 줄이기 위해 굳이 RDB가 요구되지 않는 기능은 NoSQL로 이전했다.



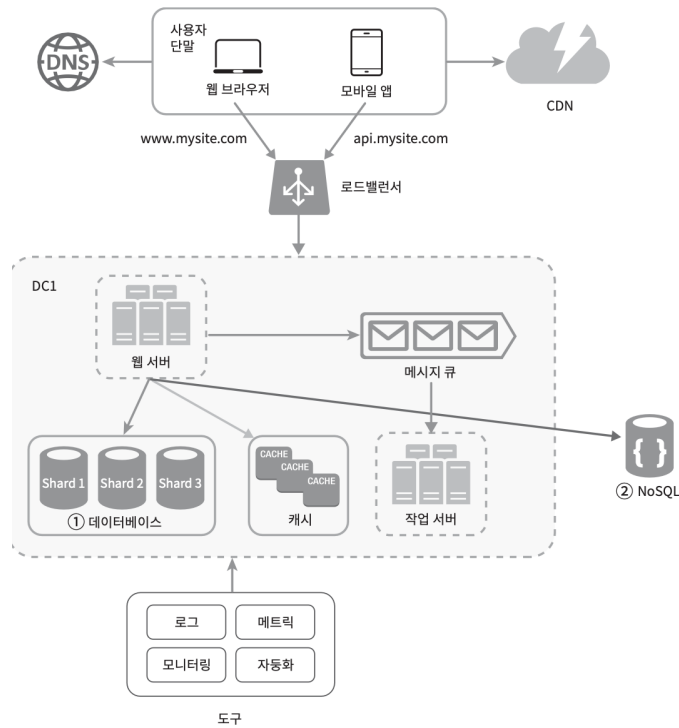


그림 1-23

## 백만 사용자, 그리고 그 이상

이번 장(1장)에서 다룬 내용을 반복하다면 원하는 규모의 시스템을 달성할 수 있게 된다. 그러나 수백만 사용자 이상을 지원하려면 새로운 전략을 도입해야 한다.

ex) 시스템을 최적화하고

더 작은 단위의 서비스로 분할해야 할 수도 있다. (MSA?)

## 1장 간단 정리

- 웹 계층은 상태 계층으로
- 모든 계층에 다중화 도입
- 가능한 한 많은 데이터를 캐시할 것
- 여러 데이터 센터를 지원할 것
- 정적 콘텐츠는 CDN을 통해 서비스 할 것
- 데이터 계층은 샤딩을 통해 그 규모를 확장할 것
- 각 계층은 독립적 서비스로 분할할 것

- 시스템을 지속적으로 모니터링하고, 자동화 도구들을 활용할 것

---

## 참고 문헌 - 책에서 나오는 레퍼런스 링크 모음

### Chapter 1: SCALE FROM ZERO TO MILLIONS OF USERS

---

1. [Hypertext Transfer Protocol](#)
2. [Should you go Beyond Relational Databases?](#)
3. [Replication](#)
4. [Multi-master replication](#)
5. [NDB Cluster Replication: Bidirectional and Circular Replication](#)
6. [Caching Strategies and How to Choose the Right One](#)
7. [Scaling Memcache at Facebook](#)
8. [Single point of failure](#)
9. [Amazon CloudFront Dynamic Content Delivery](#)
10. [Configure sticky sessions for your Classic Load Balancer](#)
11. [Active-Active for Multi-Regional Resiliency](#)
12. [Amazon EC2 High Memory Instances](#)
13. [What it takes to run Stack Overflow](#)
14. [What the heck are you actually using NoSQL for?](#)