사용자 수에 따른 규모 확장성

이번 챕터에서는 작은 수의 사용자를 커버할 수 있는 사례부터 시작해서, 대규모 사용자까지 대응 할 수 있는 구조를 만들어나갈 것이다.

웹 계층

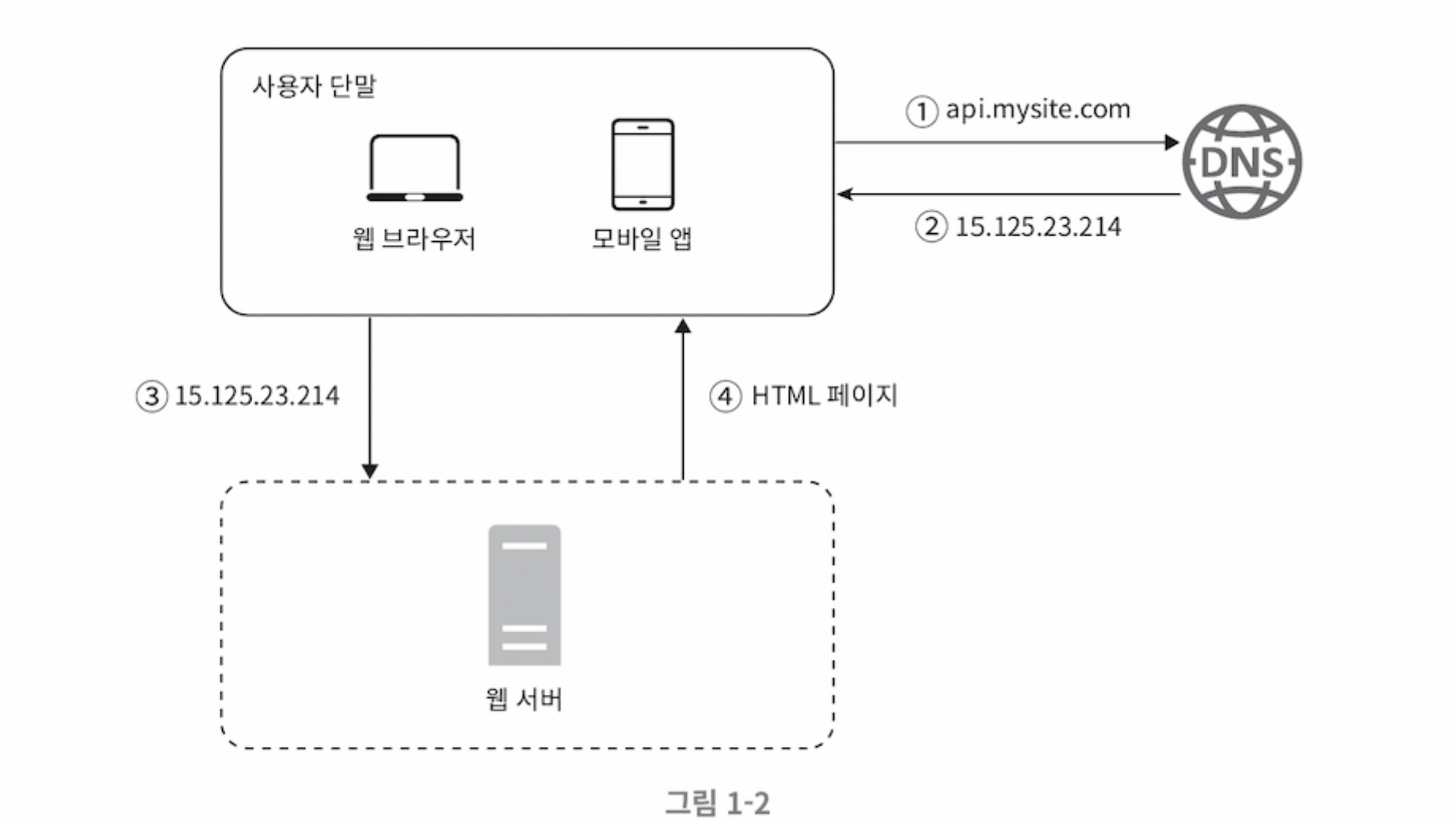

그림 1-2

1. 도메인을 통한 질의
2. DNS 결과 반환
3. 웹서버에 요청
4. 웹서버는 데이터 반환

데이터는 보통 HTTP 프로토콜을 사용하여 JSON 데이터를 반환할 것이다.

데이터 계층

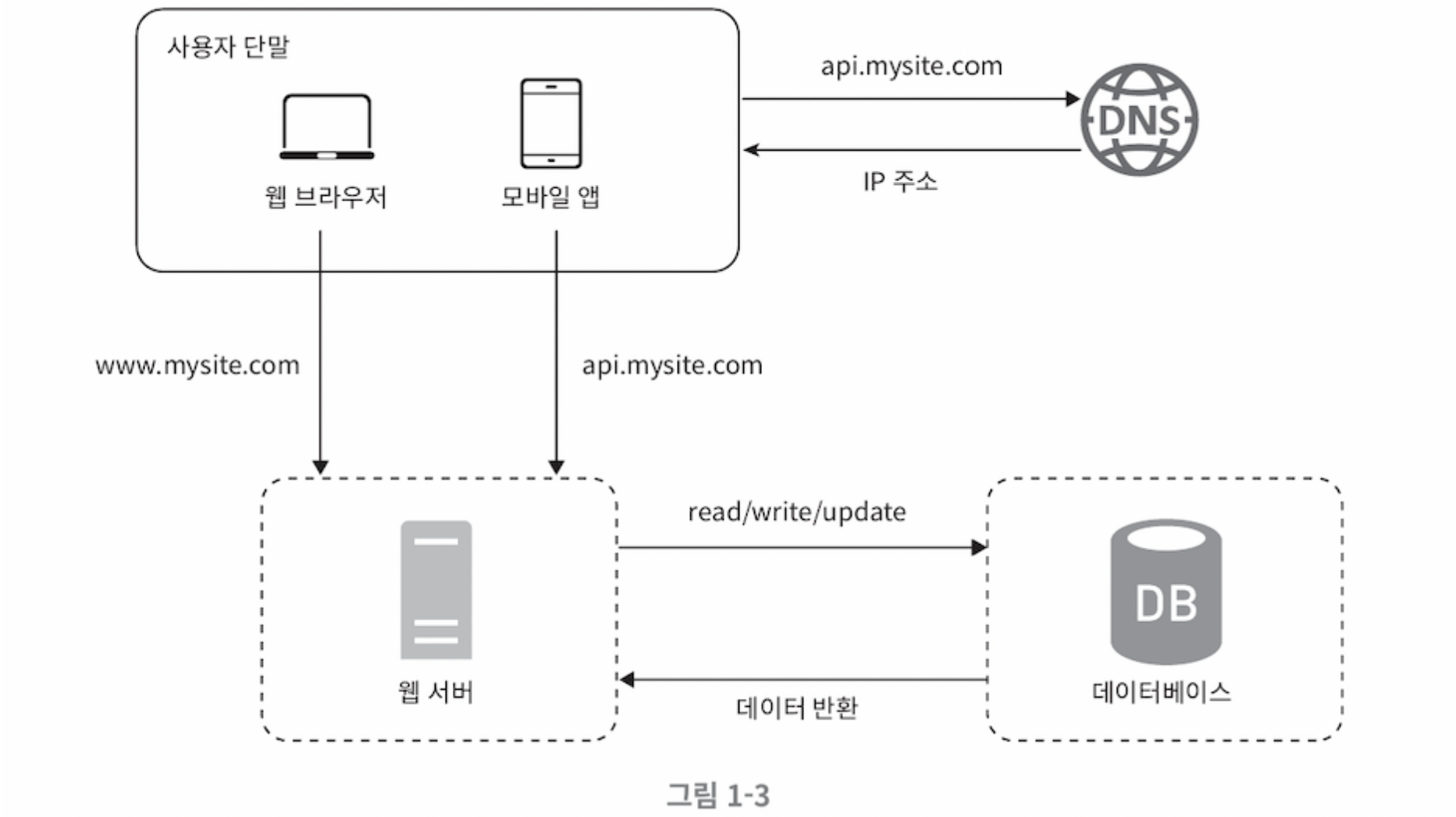

그림 1-3

- 사용자가 늘어나면, 데이터베이스용 서버를 하나 새로 두어야한다.
- 하나는 웹/모바일 트래픽 처리, 다른 하나는 데이터베이스
- 각 계층을 분리하면, 독립적으로 확장해나갈 수 있다.

어떤 데이터베이스를 사용할 것인가?

- RDBMS
  - MySQL, MariaDB, PostgreSQL
  - row, column으로 구성
  - 각 데이터들간의 관계로 정형화된 데이터를 관리
  - transaction을 통해 일관성있는 결과를 보장
- NoSQL
  - MongoDB, DynamoDB, Redis
  - key-value, graph, column, document
  - 아주 낮은 응답 지연시간 요구
  - 비정형 데이터
  - 데이터를 직력화하거나, 역직렬화 할 수 있기만 하면 됨
  - 아주 많은 데이터를 저장할 필요가 있음

Scale up vs Scale out

- Scale up
  - 한 서버의 성능을 올리는 것
  - 보통 하드웨어 스펙을 올리는 방식으로 사용

    - 하지만, 한계가 있고 자동복구 방안이나 다중화를 제시 X
- Scale out
    - 비슷한 성능 혹은 조금 더 좋은 서버들을 여러대로 증설하는 방법
    - 하나의 서버를 두는 것이 아니기 때문에, 무상태로 설계 할 필요가 있음
    - 로드밸런서를 통해, 서버간의 부하를 분산해줘야함

로드밸런서

- 로드밸런서는 보통 public ip로 접근
- private ip로 구성된 여러 서버 중 하나로 부하를 분산
- 서버1이 다운되면, 모든 트래픽이 서버2로 전송된다.
- 트래픽이 너무 몰리면, 로드밸런서를 통해 처리 가능
    - 로드밸런서가 바라보는 서버를 여러대 추가하면, 로드밸런서가 알아서 처리해 줌

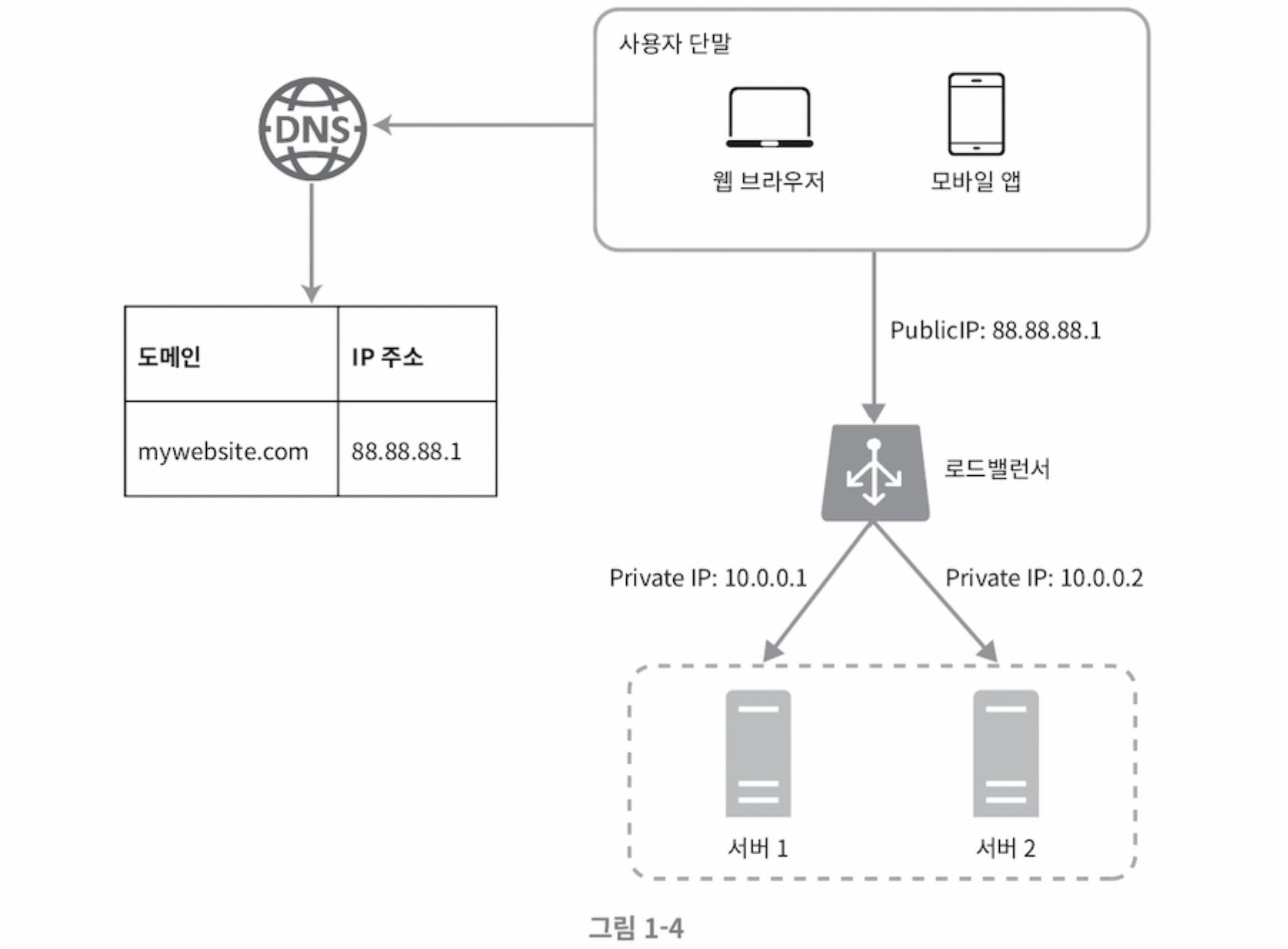

그림 1-4

데이터베이스 다중화

- master-slave 를 통해 다중화
    - write는 master에
    - read는 slave에
    - 보통 쓰기보다 읽기 작업이 더 많으므로, 읽기 작업을 하는 slave를 여러대 둔다

- 성능
  - master-slave 구조를 통해 자연스럽게 성능 향상
- 안정성
  - 하나의 데이터베이스가 파괴되어도, 데이터베이스를 다중화해놓았기 때문에 데이터 보존.
- 가용성
  - 하나의 데이터베이스가 파괴되어도, 다른 데이터베이스에서 질의 가능

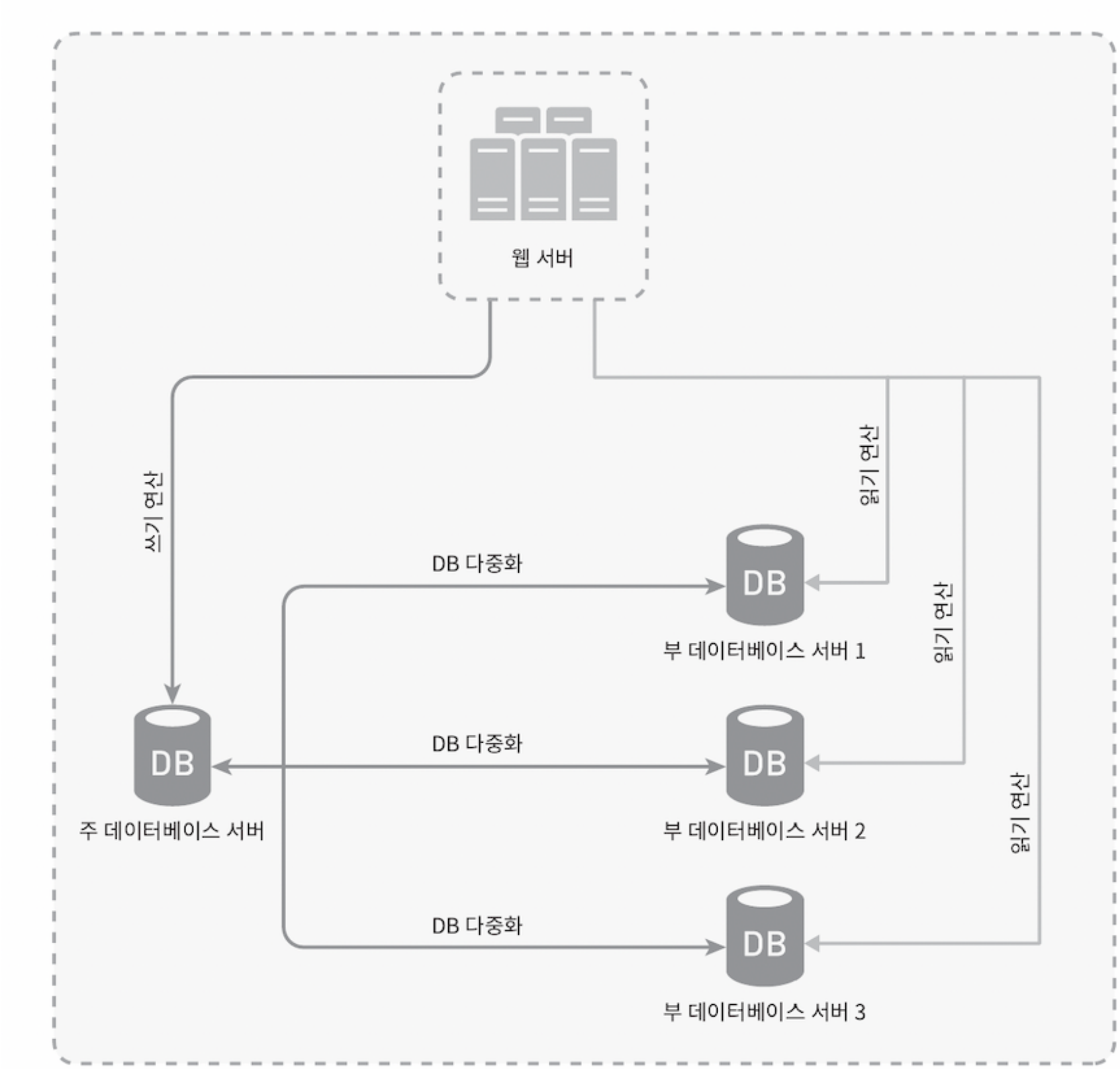

그림 1-5

> 데이터베이스 다중화는 master-slave (active-active 구조) 도 있지만, 여러 구조가 있다. Active-Standby 와 같이 다른 구조도 있으니 추후 알아보자.

현재까지의 결과

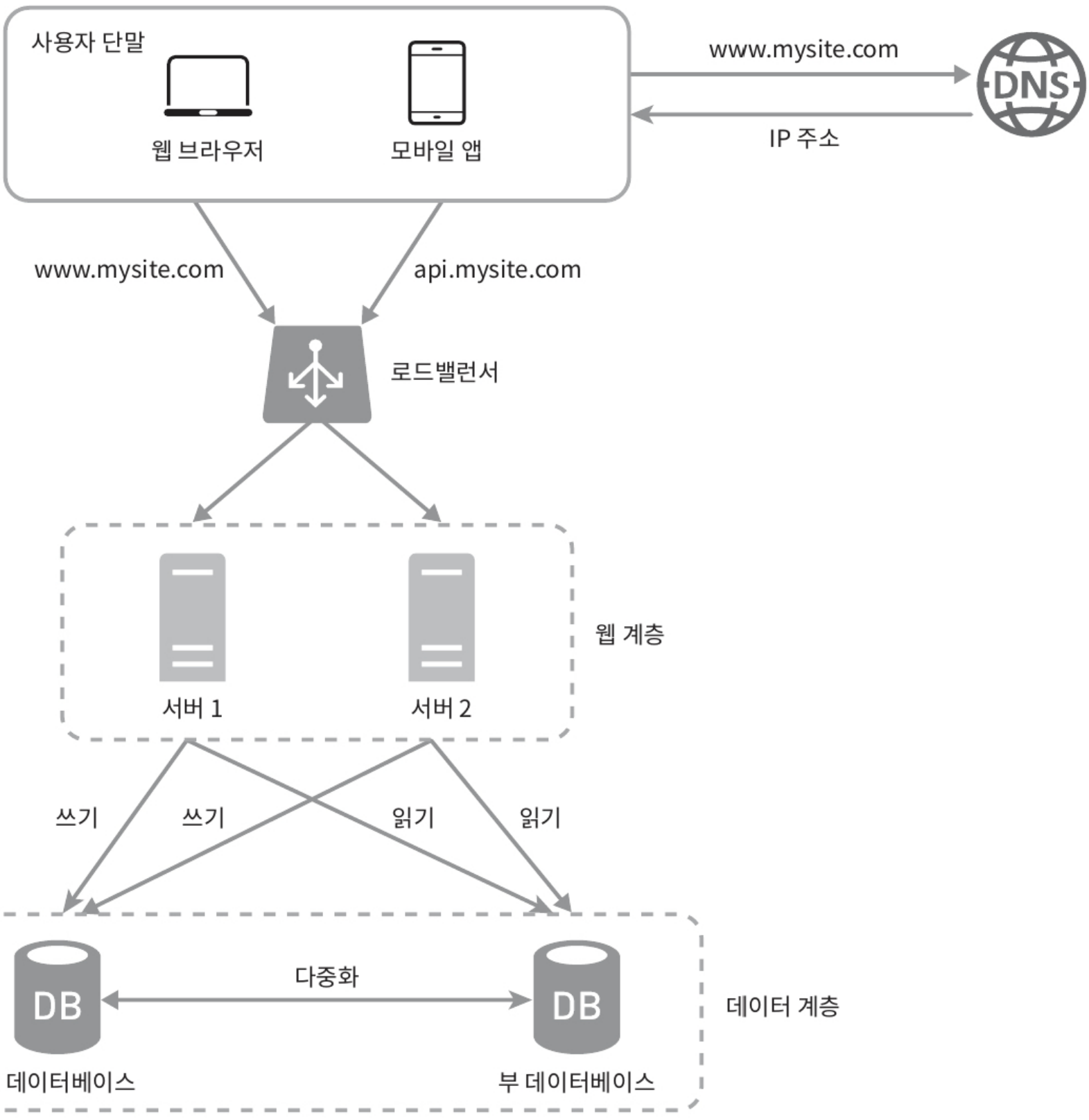

그림 1-6

1. 도메인을 통해 DNS 질의
2. DNS가 IP 주소를 반환
3. 반환된 IP를 통해 로드밸런서에 요청
4. 요청된 트래픽을 웹서버에 적절히 나눠서 요청
5. 웹서버는 쓰기 연산이라면 master에, 읽기 연산이라면 slave에 요청을한다. 이 때, master는 slave에게 데이터를 복제한다.

캐시

응답 시간을 위해, 이제는 캐시를 도입해야 할 차례이다.

보통 **값비싼 연산 또는 자주 참조되는 데이터**에 대해서 캐시를 도입한다.

> “약간의 실시간성을 배제하고, 빠른 응답을 준다.”

이러한 데이터들은 메모리에 저장되고, 빠르게 요청을 처리할 수 있게 한다.

캐시 계층

- 데이터베이스보다 훨씬 빠른 캐시는 따로 캐시 계층을 두어 성능 개선을 할 수 있다.
- 또한, 다른 계층을 둠으로써 규모를 독립적으로 확장할 수 있다.

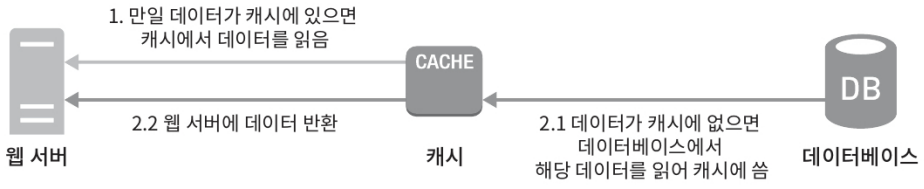

그림 1-7

위 구조는 read-through 전략으로, 캐시에 데이터가 있다면 캐시에 데이터를 두고 없다면 데이터베이스에서 읽어서 캐시에 업데이트 한 뒤에 읽는다.

이 외에로 여러가지 캐시 전략이 (Cache-Aside, Write-Through, Write-Around, Write-Back) 있는데, 심심할 때 알아보자.

또한, local cache 와 global cache와 같이 캐시를 어디에 둘 것인가에 대한 고민도 해보면 좋다.

캐시 사용 시 유의할 점

- 데이터 갱신이 자주 일어나지 않지만, 빈번하게 참조될 경우 사용을 고려
- 영속적으로 보관해야 될 데이터는 캐시에 저장하는 것이 좋지 않다. 캐시는 일반적으로 휘발성이다.
    - redis는 persistence store를 지원하기는 한다.
- 만료 정책을 잘 고려하자. 적절하게 이 데이터가 언제 만료되어야 할 지 고민을 많이해야한다. 너무 짧다면, 데이터베이스 접근이 너무 많고.. 길다면, 실시간성이 너무 떨어진다.
- 일관성을 고려하자. 원본과 캐시 데이터가 같지 않다면, 일관성이 깨질 수 있다.
- SPOF를 고려하자. 캐시 서버가 만약 한 대로 이루어져있다면, 이 캐시 서버가 고장 났을 때 결국 장애를 일으키는 원인이 되어버린다. 그렇기에 여러 서버를 두고 분산시키자.
- 캐시 메모리의 크기. 너무 작으면, eviction이 자주 일어나서 캐시의 이점을 못 얻는다.
- Eviciton 정책도 잘 정하자. LFU, LRU, FIFO 중 적절한 정책을 경우에 따라 정하자.

CDN

정적 콘텐츠를 전송하는데 쓰는, 분산 서버 네트워크.

이미지, 비디오, css, js 등을 캐시 할 수 있다.

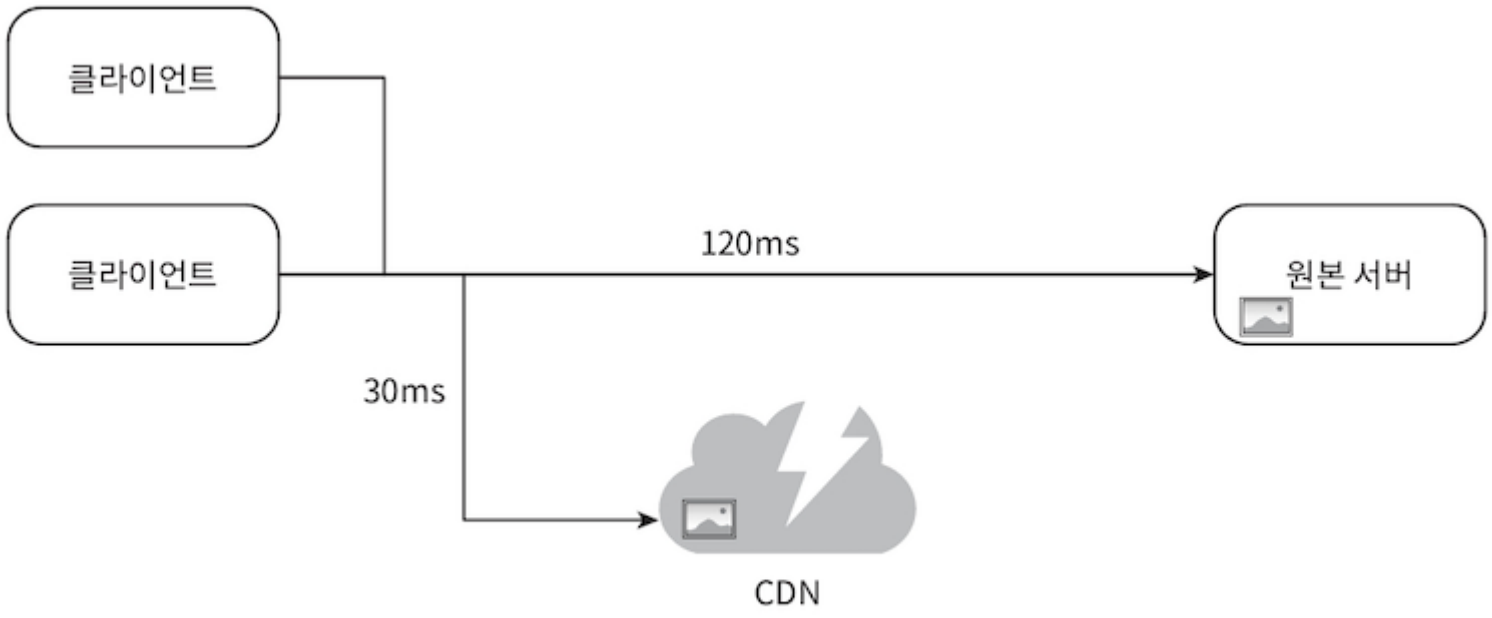

그림 1-9

CDN의 작동 방식

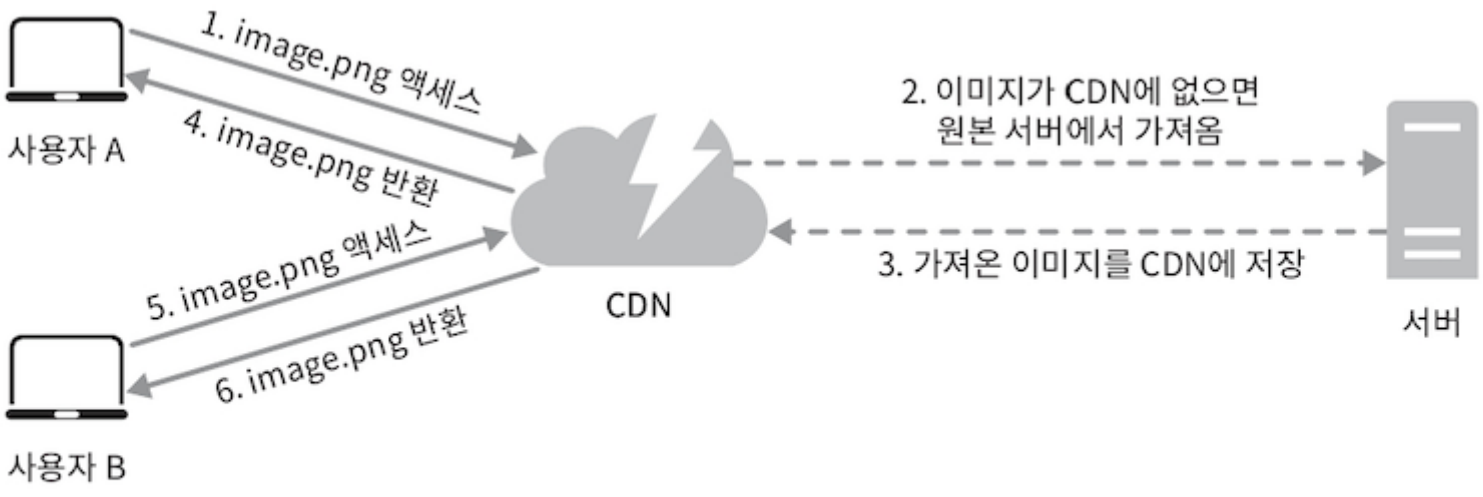

그림 1-10

- CDN을 적용하게 되면, 위와 같이 image 같은 것에 접근하게 된다면 캐시와 같이 CDN에 존재하는지를 미리 체크하게 된다.
- 만약, CDN에 존재하지 않는다면 원본 서버에 접근하여 가져와서 CDN에 저장하고 반환하게 된다.
  - 이 때, TTL을 통해 캐시를 얼마나 할 지 설정된다.
- CDN에 이미 존재한다면, CDN에서 바로 반환한다.

> 캐시의 Read-Through 전략을 생각하면 된다.

CDN 고려 사항

- 비용
  - 보통 CDN은 제 3자에 의해 운영된다. 데이터 전송양에 따라 요금을 내기 때문에, 자주 사용하지 않는 데이터라면 CDN을 적용하지 않는 것도 고려해봐도 좋다.
- TTL

    - 캐시와 마찬가지로, 적절한 만료시간을 정하자. 너무 짧으면 원본에 너무 자주 접근 할 것이고, 너무 길다면 컨텐츠의 신선함이 떨어질 것이다.
- 장애 대처 방안
    - CDN에서 문제가 발생했다면, 원본 서버에서 가져오는 전략을 적절히 클라이언트를 구성하자.
- 콘텐츠 무효화 방법
    - CDN에서 제공하는 방법
    - 다른 버전을 서비스 하도록 버저닝

현재까지의 결과

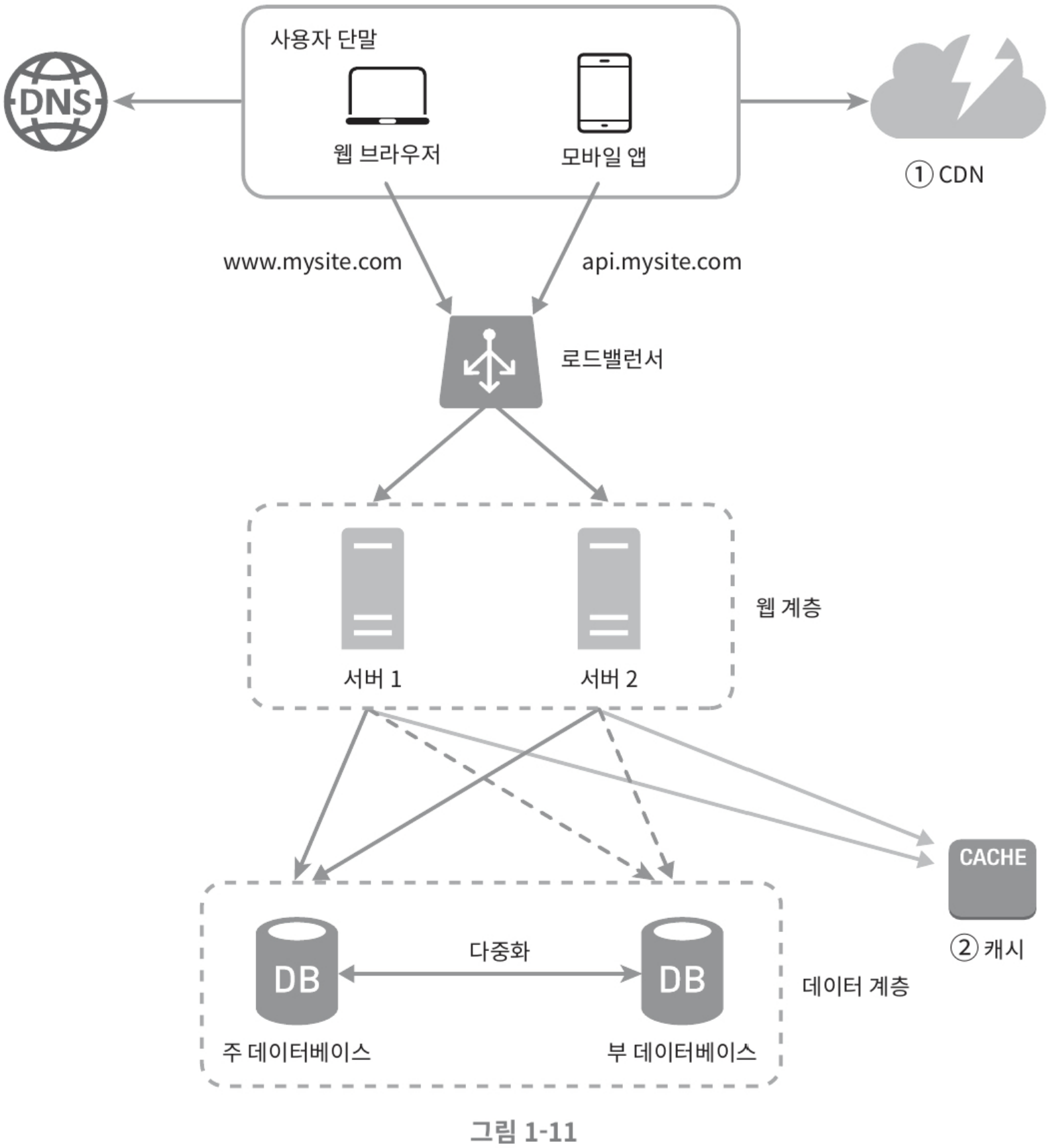

그림 1-11

이제 정적인 데이터를 웹 서버를 통해 서비스 X → 성능 향상

또한, 캐시가 데이터 베이스 부하를 줄여주고 있다.

무상태 계층

웹 계층을 수평적으로 확장하기 위해서는, 상태 정보 (세션..) 을 웹 계층에서 제거해야한다.

이러한 정보를 제거하기 위해서는, NoSQL이나 RDBMS에 상태 정보를 저장하고 필요할 때 가져오는 것.

Stateful 계층

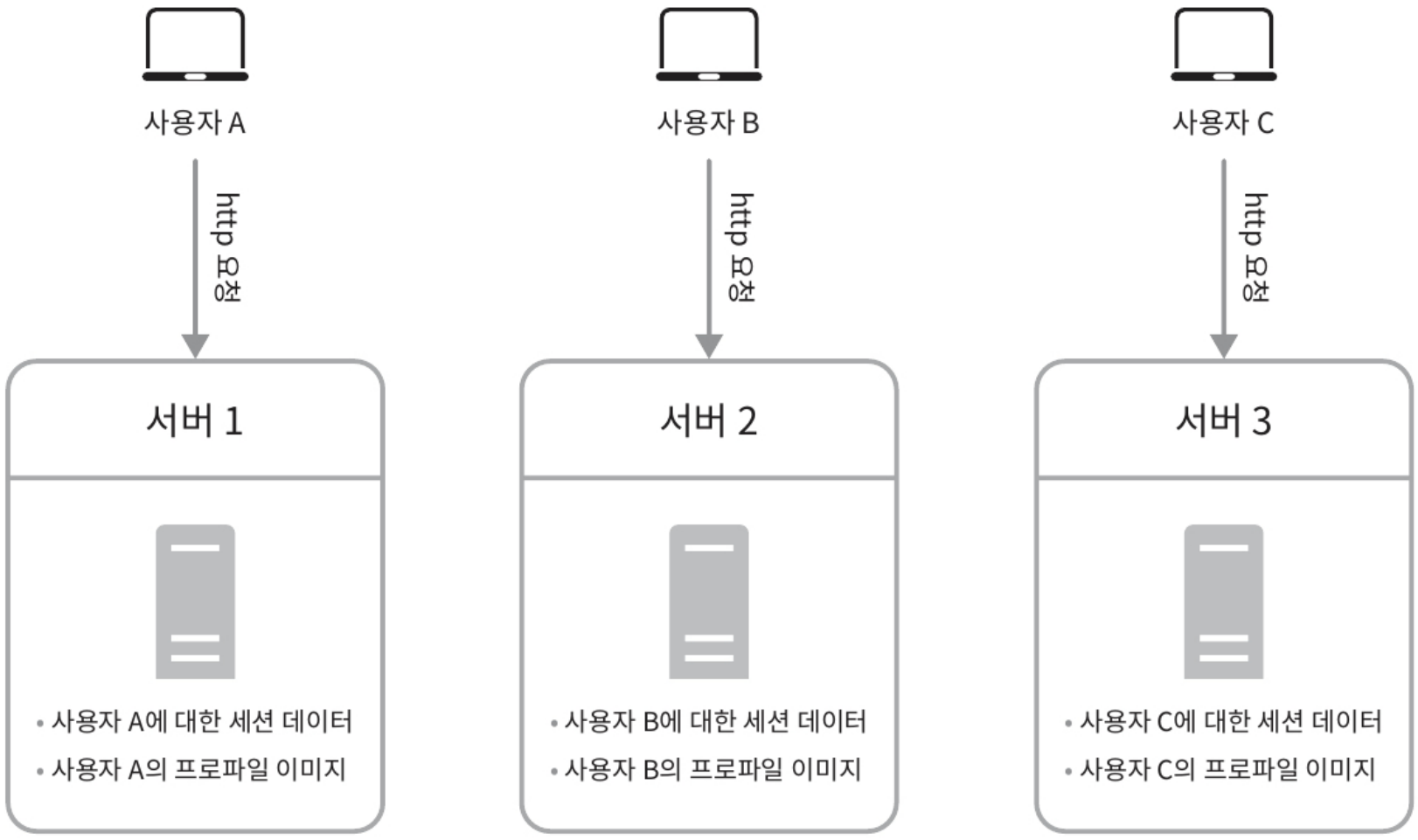

그림 1-12

- 상태 정보를 서버에 보관하기 때문에, 같은 클라이언트에 대해서 항상 같은 서버로 요청을 해야한다는 문제.
- 그렇기 위해, sticky-session이나 session-clustering을 할 수 있지만.. 이것도 LB에게 부담을 준다.

Stateless 계층

위에서 stateful의 문제점을 알아봤기 때문에, stateless한 계층을 설계해보자.

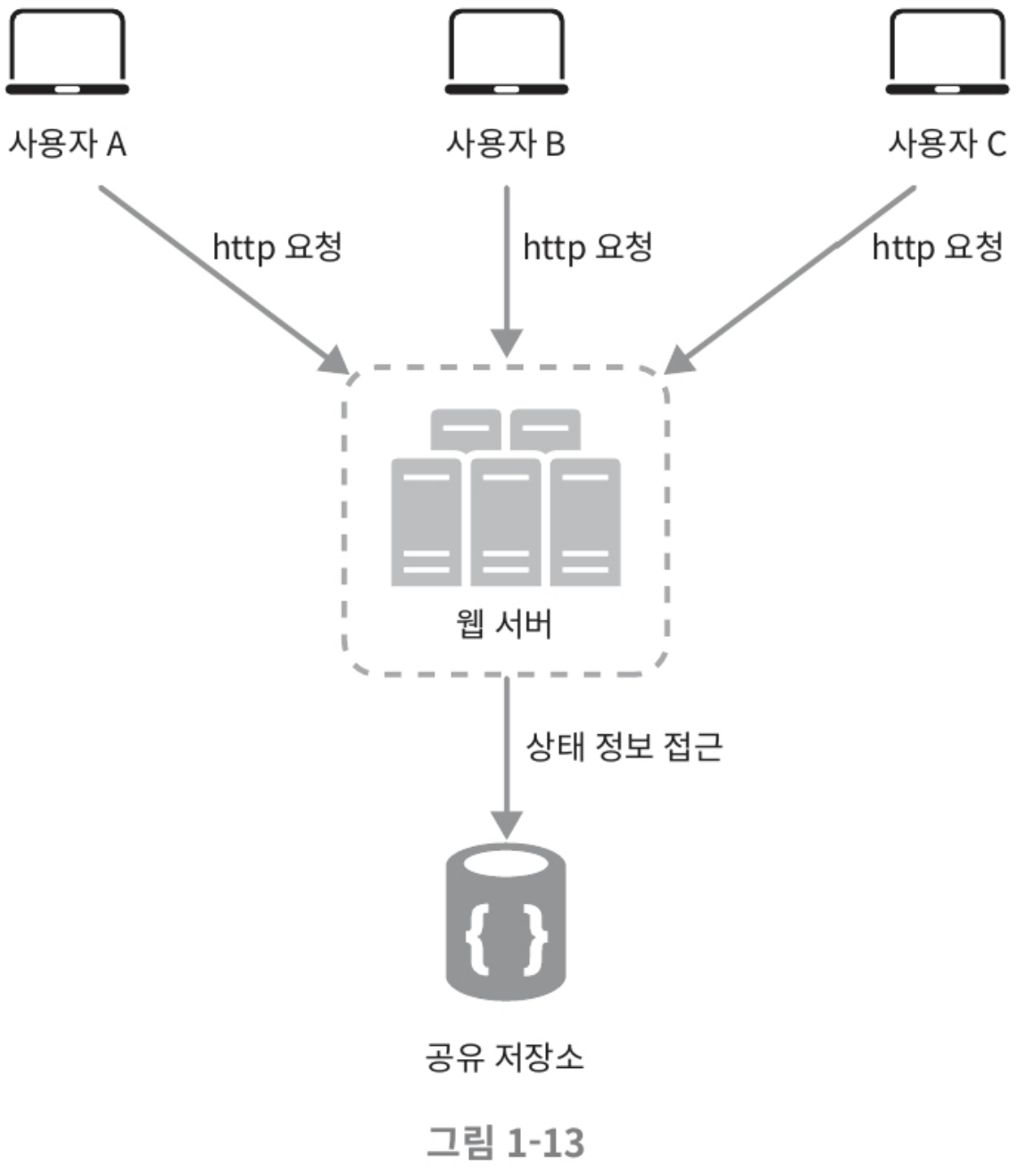

그림 1-13

- 이제 서버는 상태 정보를 서버의 메모리 (세션) 에 저장하지 않고, 공유 저장소 (보통 NoSQL) 에 저장하게 되었다.
- 그렇기 때문에, 항상 같은 서버에 접근할 필요가 없다. LB에게도 부담이 없다.

현재까지의 결과

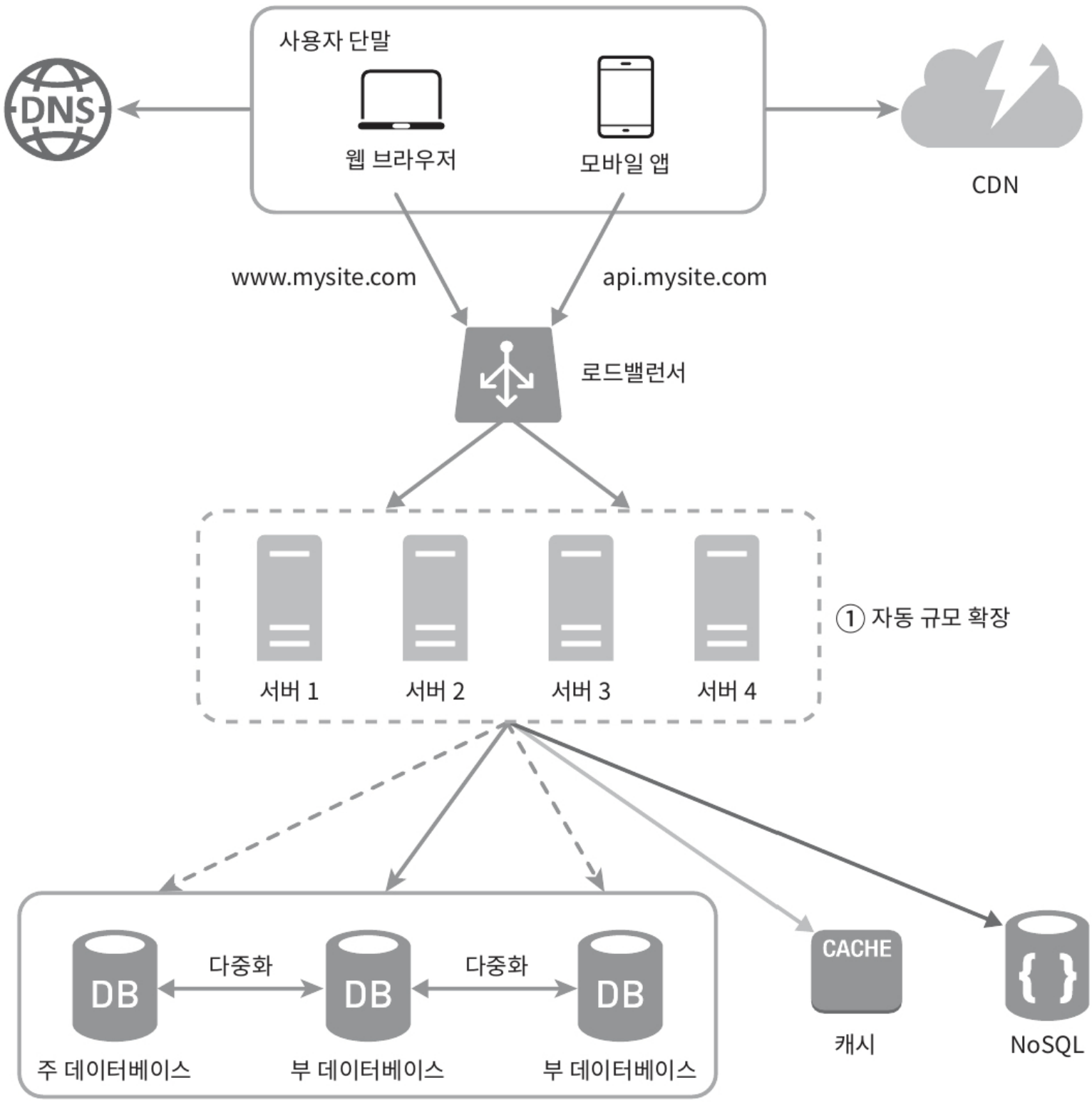

그림 1-14

- 이제 상태 정보를 관리하기 위해 NoSQL 을 도입하였다.
- NoSQL이 RDBMS보다 확장하기 편하기 때문.
- NoSQL (공유 상태 저장소) 를 도입했기 때문에, 이제는 웹서버는 무상태성을 가지게 되었고 규모 확장도 간편해졌다.

> NoSQL 도 당연하게, SPOF가 되지 않도록 failover를 고려하자.

데이터 센터

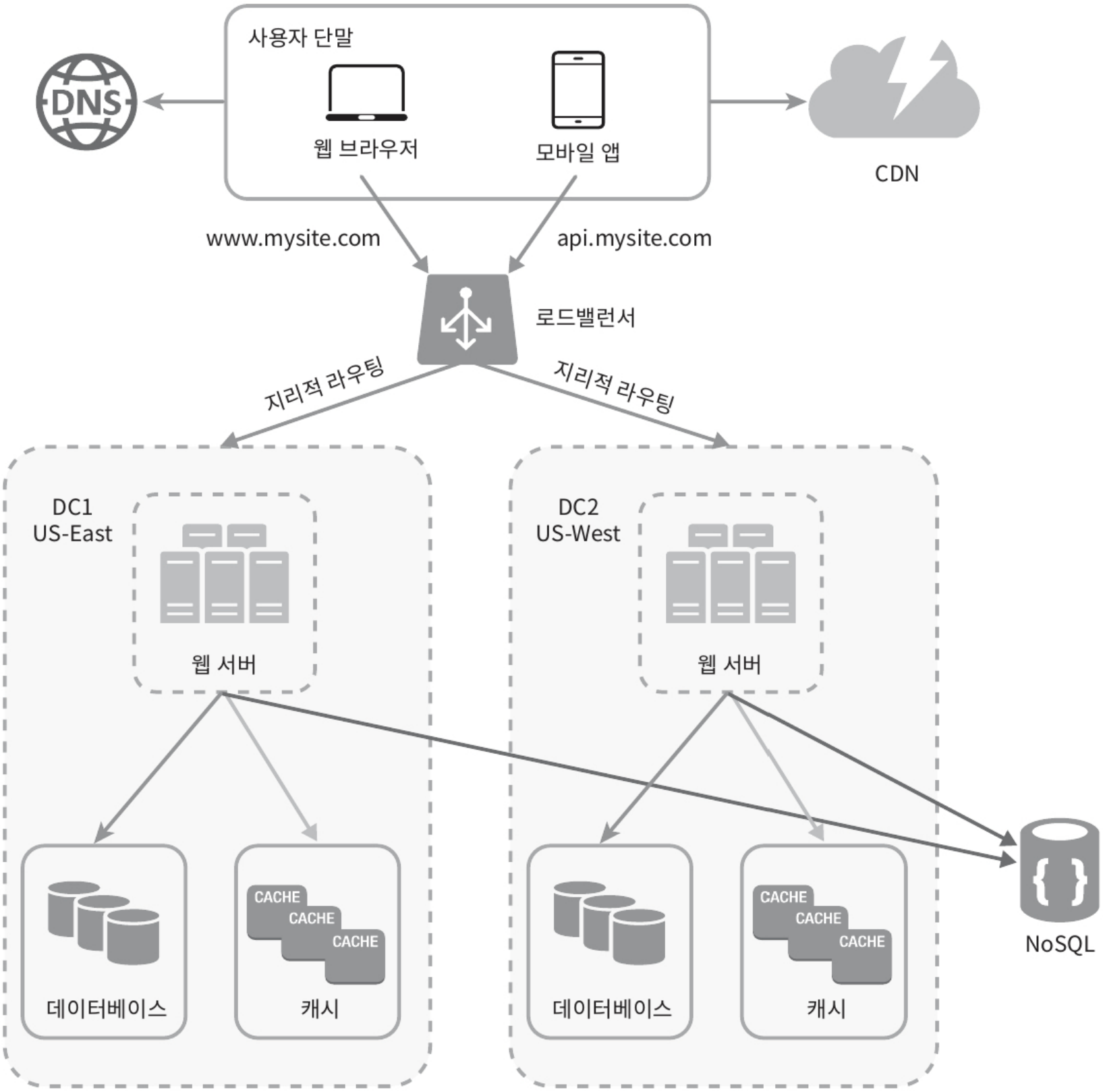

그림 1-15

위 사례는, 두 개의 데이터 센터를 이용하여 지리적 라우팅 (가까운 데이터센터에 요청) 을 하는 방식이다.

LB를 통해서, EAST, WEST에 자연스럽게 라우팅 된다.

> 만약 여기서 한 데이터 센터에 문제가 생긴다면? 아래 그림처럼 문제가 발생하지 않은 하나의 데이터 센터에 모든 트래픽이 갈 것이다.

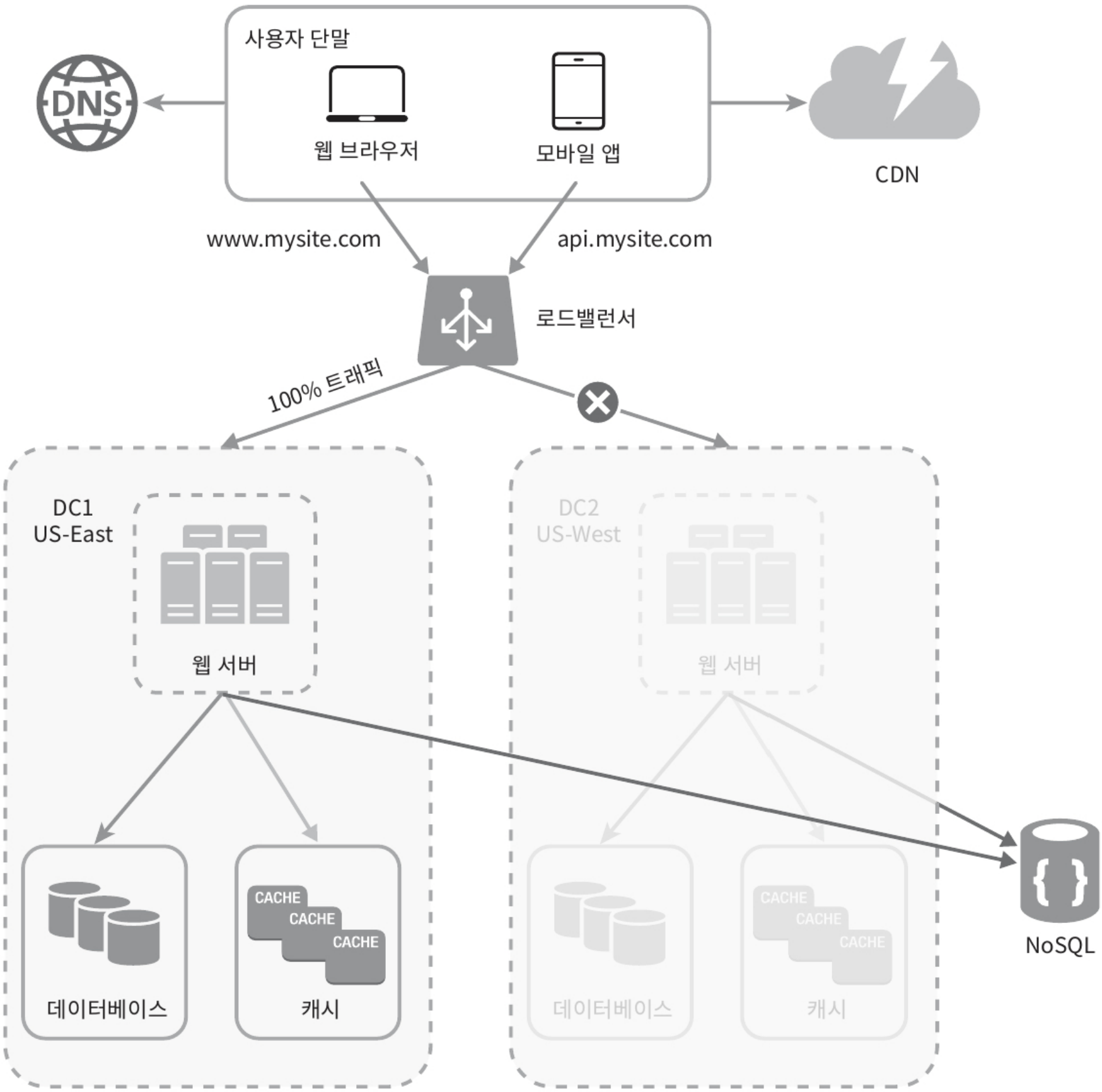

그림 1-16

고려해야 할 것

- 트래픽 우회
  - 올바른 데이터 센터로 보내는 효과적인 방법을 사용해야 함.
- 데이터 동기화
  - 각 데이터 센터간의 데이터를 어떻게 동기화해야할지 고민해봐야함.
- 테스트와 배포
  - 여러 데이터 센터를 사용할 경우, 서비스를 여러 위치에서 테스트 해봐야 함.

메시지 큐

시스템을 더 크게 확장하기 위해서는 메시지 큐를 통해서, 각 서비스 간의 결합도를 낮춰 독립적으로 확장할 수 있도록하여야한다.

메시지큐는 메시지의 무손실을 보장하는 컴포넌트로, 비동기 통신을 지원한다.

- 버퍼 역할을 하여,비동기적으로 전송
- producer 가 메시지를 발행하여 메시지 큐에 전송하면, subscriber가 consume하는 역할을 한다.
- 각 서비스간의 결합이 느슨해져서, 규모 확장이 쉬워지고 애플리케이션을 안정적으로 서비스 할 수 있다.

그림 1-17

아래 사례를 통해 알아보자.

- 웹 서버에서 보정 작업을 위한 메시지를 발행한다.
- 보정 작업 프로세스는 메시지 큐에서 적절히 꺼내서 소비한다.
- 비동기적으로 작업을 완료한다.

그림 1-18

> 이러한 방식을 통해, 각 서비스는 독립적으로 확장 가능해졌다. 각 서비스는 서비스가 아닌 메시지큐를 바라보고 있고, 이러한 것을 통해 한 서버에 문제가 생겨도 다른 서비스까지 오류가 전파되지 않는다.

> 또한, 보정 작업 프로세스는 메시지 큐의 사용률에 따라 적절히 프로세스를 줄이고 늘릴 수 있을 것이다.

로그 메트릭

다중화된 서비스가 되었다면, 이제는 로그 메트릭을 잘 수집하는 것도 중요하다.

한 대의 서비스라면, 로그 메트릭이 필수적이지 않다.

규모가 커지고 나면, 필수적이다.

> 하나의 서버로 이루어진 서비스를 서빙한다면, 그 서버에서 바로 로그를 본다거나 하는 방식으로 로그를 추적하거 메트릭을 볼 수 있겠지만.. 다중화된 서비스에서는 쉽지 않다. 로그 메트릭이 필수적이다.

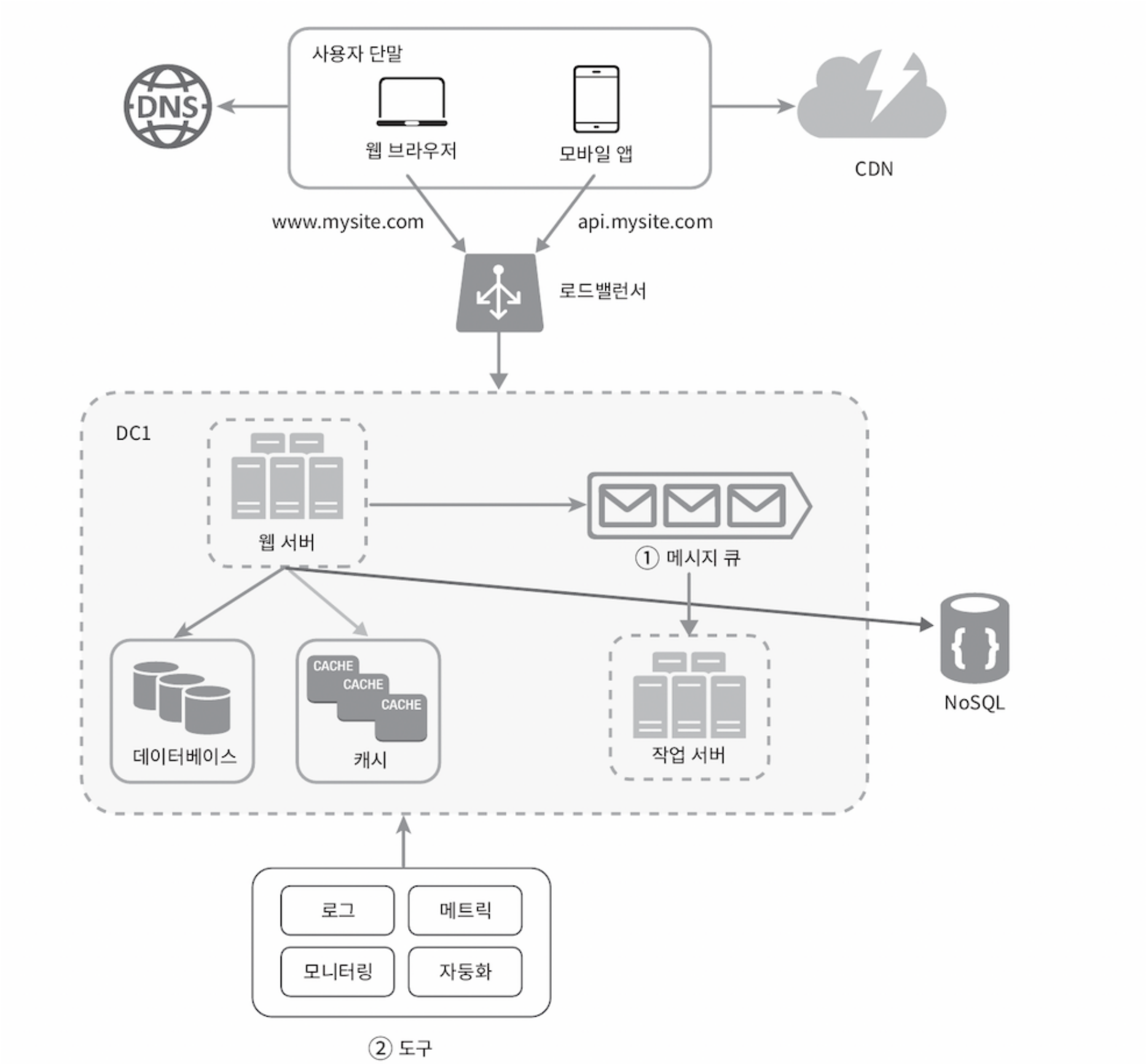

그림 1-19

- 로그
  - 시스템의 오류와 문제를 모니터링 할 수 있다.
- 메트릭
  - 메트릭을 잘 수집하면, 비즈니스에 유용한 정보를 얻을 수 있다.
  - 호스트 단위 메트릭, 종합 메트릭, 핵심 비즈니스 메트릭을 잘 살펴보자.
- 자동화
  - 시스템이 크고 복잡해지면, 자동화 도구를 통해 생산성을 올리자.
  - CI/CD 를 도입하는 것이 대표적인 자동화 도구 도입 방식이다.

데이터베이스의 확장

데이터가 많아지면, 데이터베이스의 부하도 증가.

이제는 데이터베이스를 증설하는 방안을 고려해보자.

Scale up

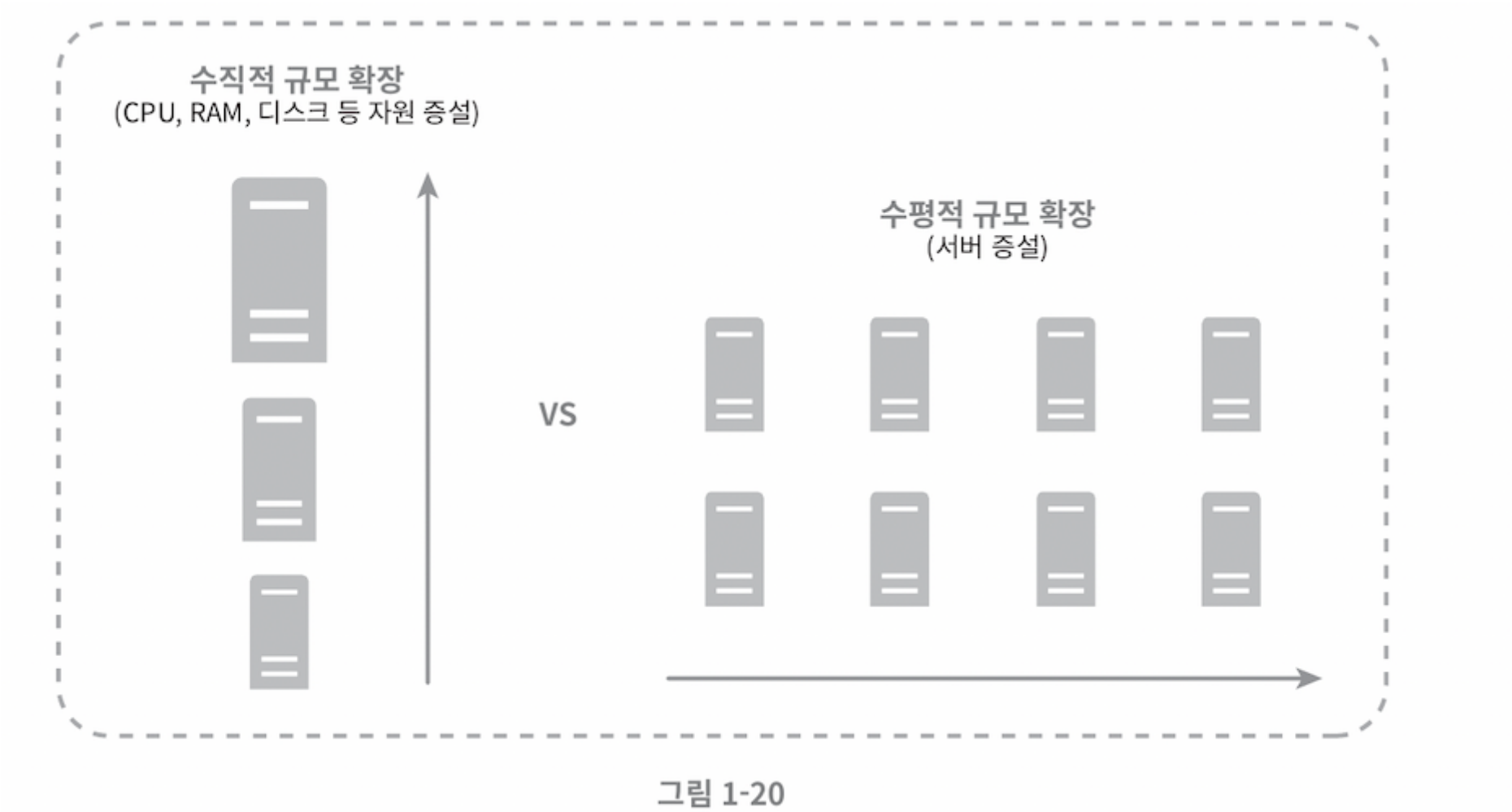

그림 1-20

- 수직적 확장. 하나의 데이터베이스의 자원을 고성능으로 증설하는 방법.
- 하지만, 하드웨어의 성능은 한계가 있음.
- SPOF 위험성이 크다.
- 비용이 많이든다.

Scale out

Scale up의 단점을 알아 봤으니, 이걸 대응하기 위한 Scale out 방식을 고려해보자.

샤딩

샤딩은 데이터베이스를 샤드라는 작은 단위로 분할하는 기술이다.

모든 샤드는 같은 스키마를 쓰지만, 데이터 사이에 중복이 없다.

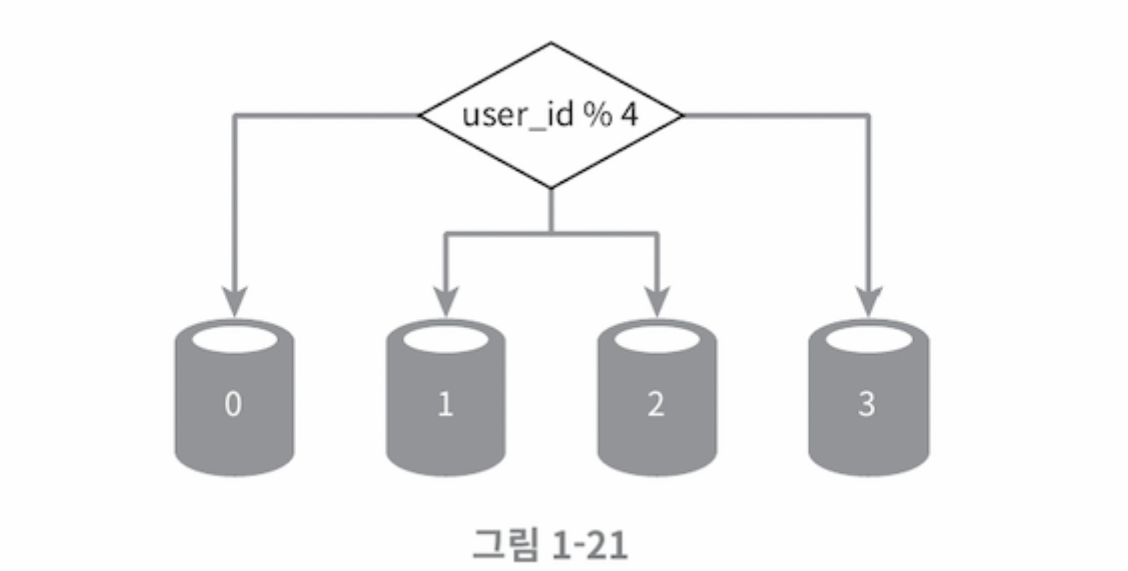

그림 1-21

보통 위와 같이, 적절한 해시 함수를 통해 데이터 베이스에 분산된 데이터를 저장한다.

실제 user_id % 4 를 사용했을 경우, 저장되는 데이터는 아래와 같다.

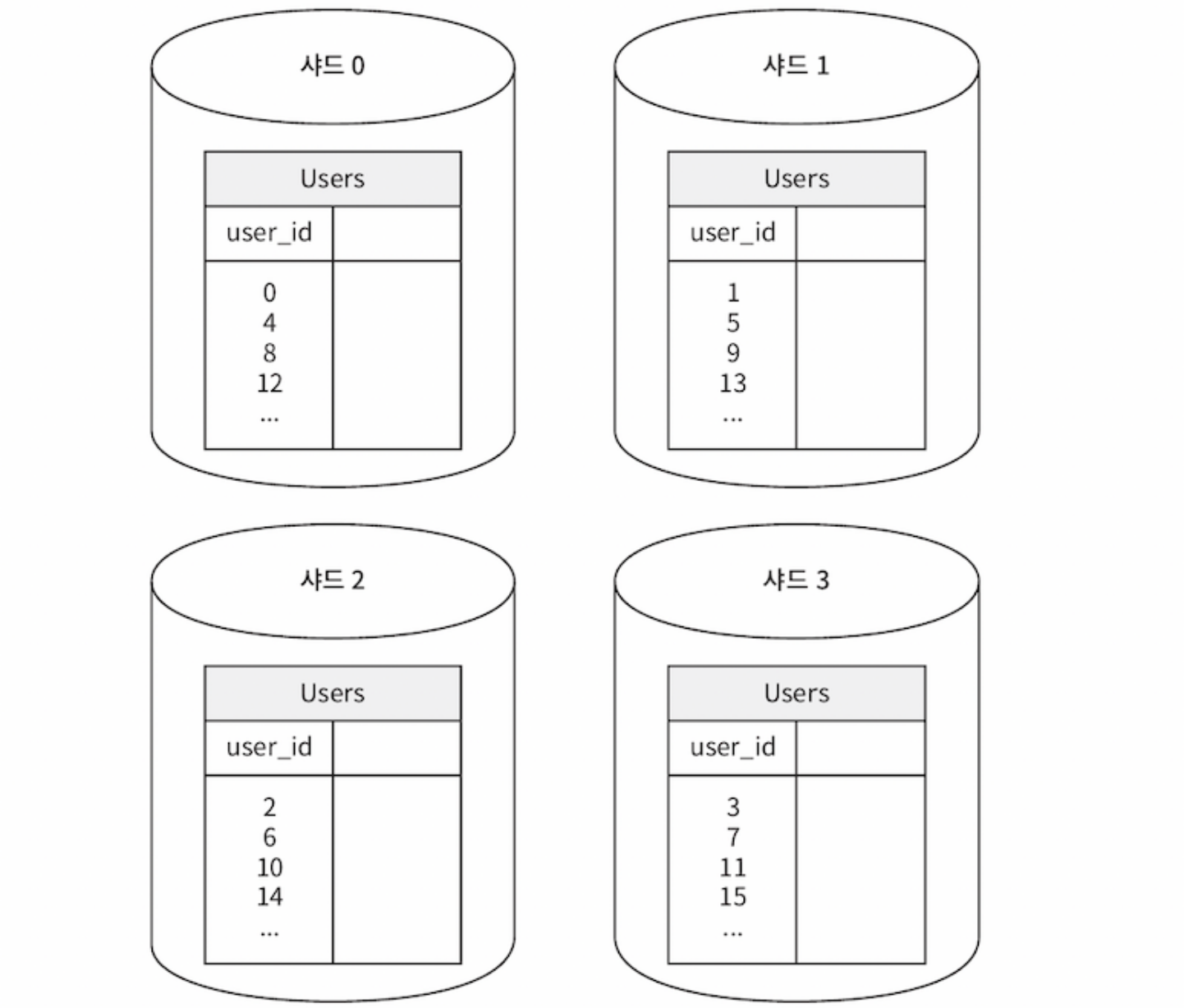

그림 1-22

샤딩시 고려해야 할 문제

- 샤딩 키를 어떻게 정해야하나
  - 데이터를 고르게 분할 할 수 있도록 정의하는 것이 가장 중요.
- 데이터의 재 샤딩
  - 하나의 샤드로는 감당이 어려울 때
  - 샤드 간의 데이터 분포가 균등하지 못하여 어떤 샤드만 다른 곳 보다 소모가 빨리 될 때. (샤드 소진)

    - 이러한 현상에서는 샤드키를 계산하는 함수를 변경하고 데이터를 재배치해야한다.
- 유명인사 문제
    - 예를 들어, 한 유저가 엄청난 유명인사라면 하나의 샤드에 데이터가 몰리게 될 것이다.
    - 이러한 문제를 풀려면, 각 유명인사 마다 샤드를 하나씩 할당해야할 수도 있고 여러 샤드에 한 유저에 대한 정보를 저장해야할 수도 있다.
- 조인과 비정규화
    - 하나의 데이터베이스를 여러 샤드로 쪼갰기 때문에, 여러 샤드의 데이터를 조인하기 쉽지않다.
    - 이러한 상황에서 해결하는 방식은 데이터를 비정규화하여, 하나의 테이블에서 질의가 수행될 수 있도록 하는 것이다.

최종 결과

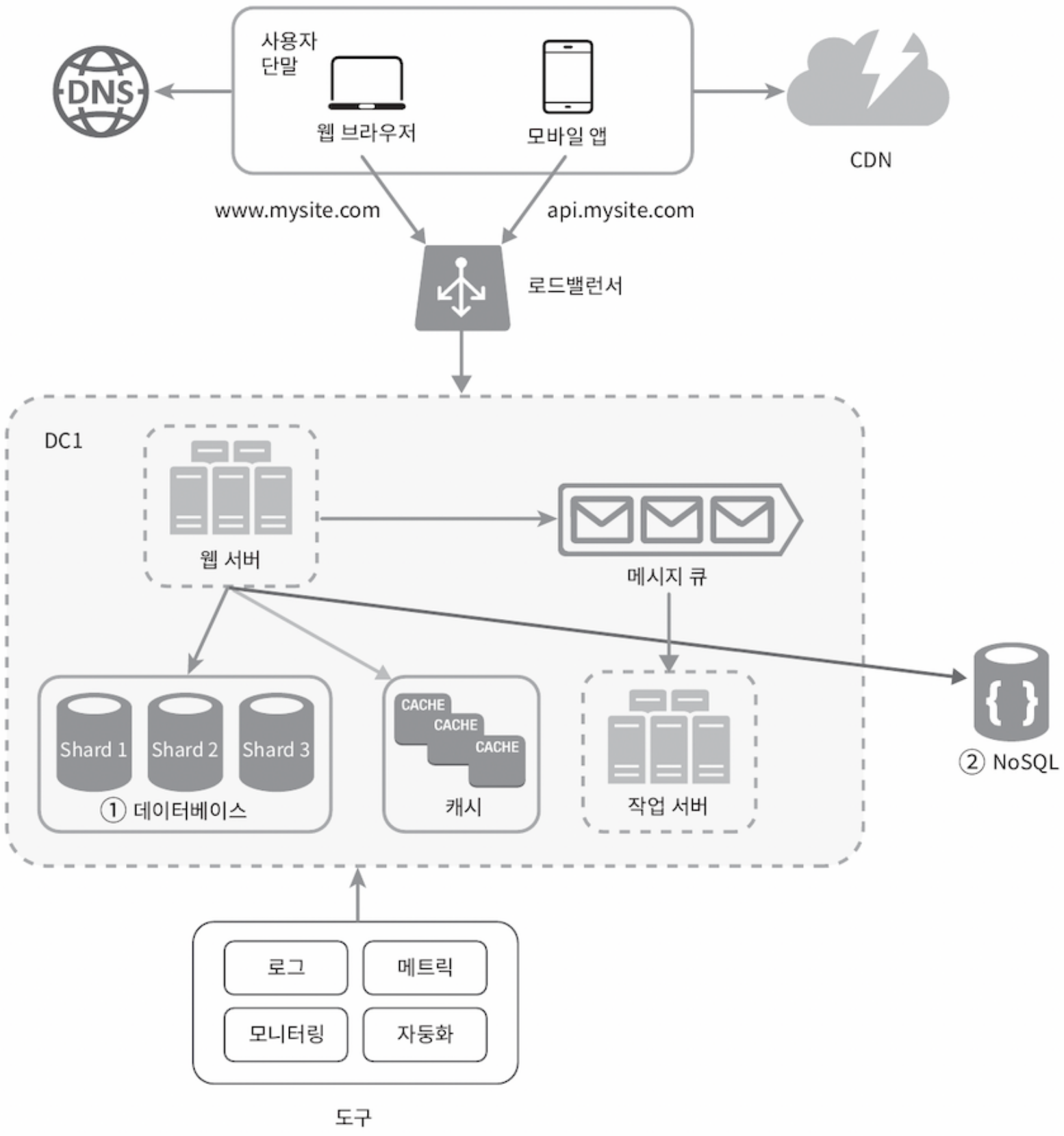

그림 1-23

- 이것이 최종적인 결과이다.
- 우리는 아래와 같은 사항을 적용했다.
  - 웹서버를 무상태로 설계하여, NoSQL을 도입했고 수평적인 확장이 가능하게 했다.
  - CDN을 통해 정적인 콘텐츠를 빠르게 전달할 수 있게 했다.
  - 데이터베이스 샤딩과 master-slave 구조를 적용하여, 수평적 확장을 진행했다.
  - 캐시 계층을 두어 데이터베이스의 비싼 연산 혹은 자주 참조되는 데이터를 빠르게 응답할 수 있게 되었다.
  - 로그 메트릭 도구를 통해, 서버 모니터링을 손쉽게 할 수 있게 되었다.
  - 메시지 큐를 도입하여, 작업서버와의 결합도를 줄이고 비동기적인 통신을 할 수 있게 되었다.
  - 웹서버와 데이터센터 다중화를 통해 안정적인 서비스가 가능해졌고, 로드 밸런서를 통해 서비스하게 되었다.