# 4장 데이터베이스와 아키텍처 구성



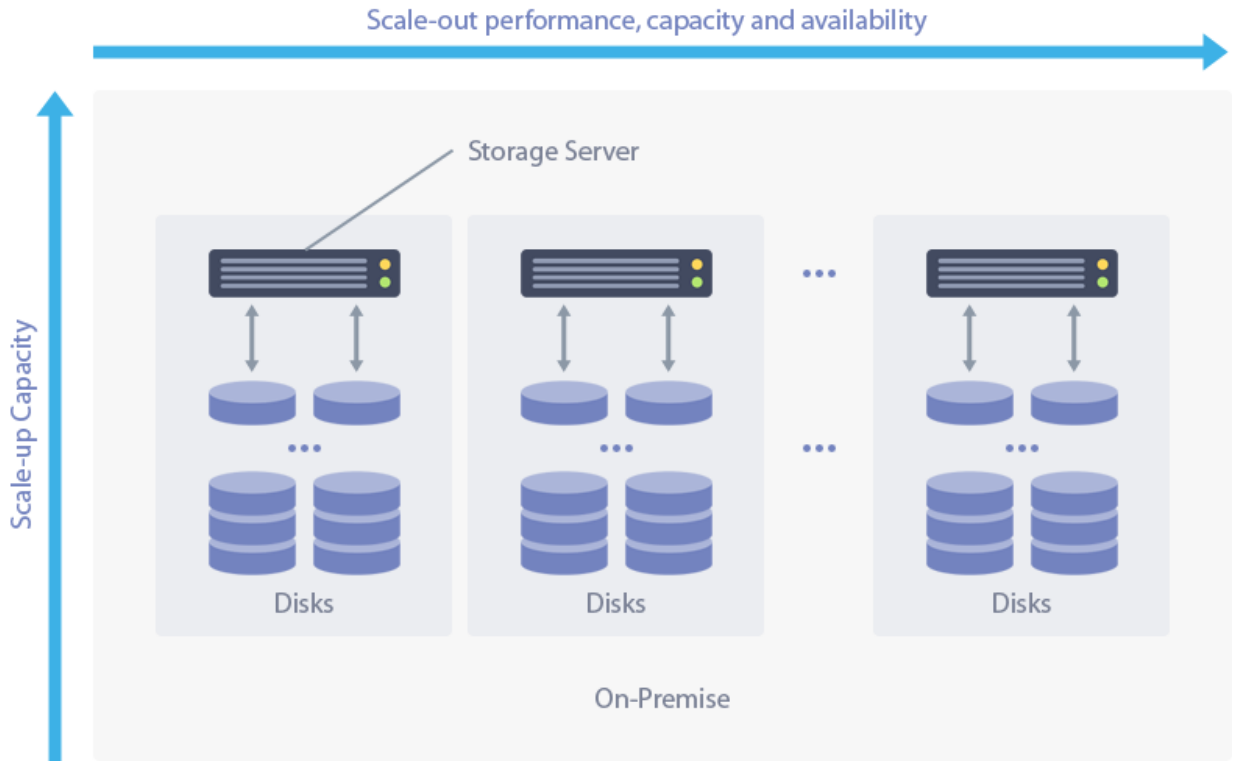
## 다중화? 가용성? 확장성?

다중화, 고가용성

- DB 서버가 2대 있다면 1대가 고장 난다 하더라도 나머지 1대가 동작하는 경우 서비스 정지를 막을 수 있다.

가용성

- 시스템이 서비스 제공시간에 장애 없이 서비스를 계속 지속할 수 있는 비율




`확장성 - Scale up`

- 기존의 하드웨어를 보다 높은 사양으로 업그레이드하는 것
- 수직 스케일링

`확장성 - Scale out`

- 장비를 추가해서 확장하는 방식
- 비슷한 사양의 서버를 연결해 추가된 분만큼 용량이 증가시키거나 워크로드를 분담해 성능을 높이는 병렬 컴퓨팅을 구현할 수 있다
- 수평 스케일링

## 데이터베이스 아키텍쳐

1. stand-alone

- 데이터베이스가 동작하는 머신이 LAN 이나 인터넷 등의 네트워크에 접속하지 않고 독립되어 동작하는 구성
- DBMS 와 애플리케이션 소프트웨어가 같은 DB 서버에서 동작하는 상황.
- 물리적으로 떨어진 공간에서 접속할 수 없으며, 복수 사용자가 동시에 사용할 수 없다. 가용성(한대가 죽으면 끝난다), 확장성이 낮다(머신 자체를 좋은 머신으로 바꾸는 방법뿐이다)
- 보안상 좋다

2. 클라이언트/서버
    - 복수 사용자가 동시에 작업할 수 있고, 원격지에서 사용할 수 있다.
    - DB 서버에 DBMS 를 띄우고, 개인 PC에서 네이티브 애플리케이션을 설치해서 DB 서버에 접속하는 클라이언트로 사용하는 방식
    - 인터넷에서 직접 DB 서버에 접속하기에 보안상 위험하다.
    - 네이티브 어플리케이션의 문제점: 갱신 후 동작하지 않는 문제, 버전 선택의 문제

3. WEB 3계층
    - 클라이언트 - ( 인터넷 ) - 웹서버 - 애플리케이션 서버 - DB 서버
    - 클라이언트와 데이터베이스 계층 사이에 웹 서버 계층과, 애플리케이션 계층이 추가된 형태
    - 웹 서버

      클라이언트로부터 HTTP 요청을 직접 받아 그 처리를 뒷단의 애플리케이션 계층에 넘기고 결과를 클라이언트에 반환

      아파치, Nginx, IIS, ...
    - 애플리케이션 서버

      비즈니스 로직을 구현한 애플리케이션이 동작하는 층

      웹 서버로부터 받은 요청을 처리하고 필요하면 DB 서버에 접속해서 데이터를 추출하고 이를 가공한 결과를 웹 서버에게 반환함

      톰캣, 네티, 제티, ...

- 복수 사용자가 동시에 작업할 수 있도록, 원격지에서 사용할 수 있도록 하면서 네이티브 어플리케이션의 문제를 해결한 아키텍처
- 허나 여전히 가용성과 확장성이 낮다는 문제가 있다.

## 가용성과 확장성의 확보

WEB 3계층 구조를 사용하면

- 복수 사용자가 동시에 DB에 작업 할 수 있고
- 원격지에서도 DB 를 사용할 수 있다.

그렇지만 여전히 서버가 1대이기 때문에

- 가용성이 낮고 (서버가 1대뿐이기 때문에 그 서버에 장애가 일어나면 서비스가 정지)
- 확장성이 부족하다 (서버가 1대인 상황에서 서버의 성능의 한계에 도달하면 부품을 업그레이드 하는 방법 뿐)

가용성을 높이는 2가지 전략

1. 심장 전략 (고품질, 소수 전략) → scale up ?

   견고한 신뢰성을 갖춘 컴포넌트를 사용해서 죽지 않게 하자!

2. 신장 전략 (저품질, 다수 전략) → scale out ?

   어차피 사물은 언젠간 망가진다. 여러개의 여분을 마련해두자!

클러스터

- 동일한 기능의 컴포넌트를 여러개 준비해 병렬화 하는 것
- 클러스터 구성으로 시스템의 가동률을 높이는 것 (가동률? 가용성? Availability?)

  = 여유도(Redundancy)를 확보한다

  = 다중화
- 서버 대수와 가동률은 처음에 큰 폭으로 증가하다가 점차 증가 폭이 줄어드는 증가 그래프를 그린다.

(가동률 증가 폭 9% → 0.9% → ... )

단일 장애점 SPOF

- 단일 장애점의 신뢰성이 시스템 전체의 가용성을 결정한다
- 쇠사슬의 강도는 가장 약한 고리의 강도로 결정된다.

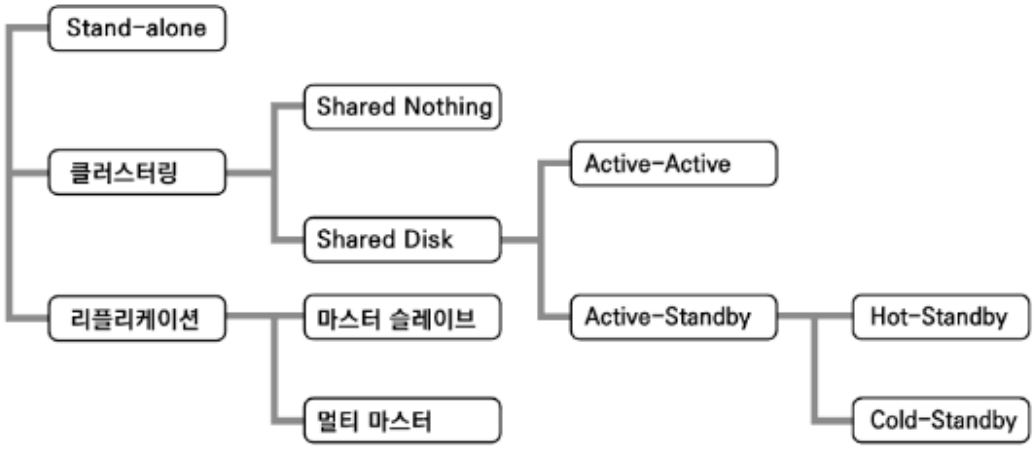
신뢰성

- 하드웨어나 소프트웨어가 고장나는 빈도나 고장 기간을 나타내는 개념

가용성

- 사용자 입장에서 볼때 시스템을 어느 정도 사용할 수 있는지를 나타내는 개념
- 99%의 가용성? 1년 중 3일 15시간 29분은 서비스 다운이 일어날 수 있다는 뜻

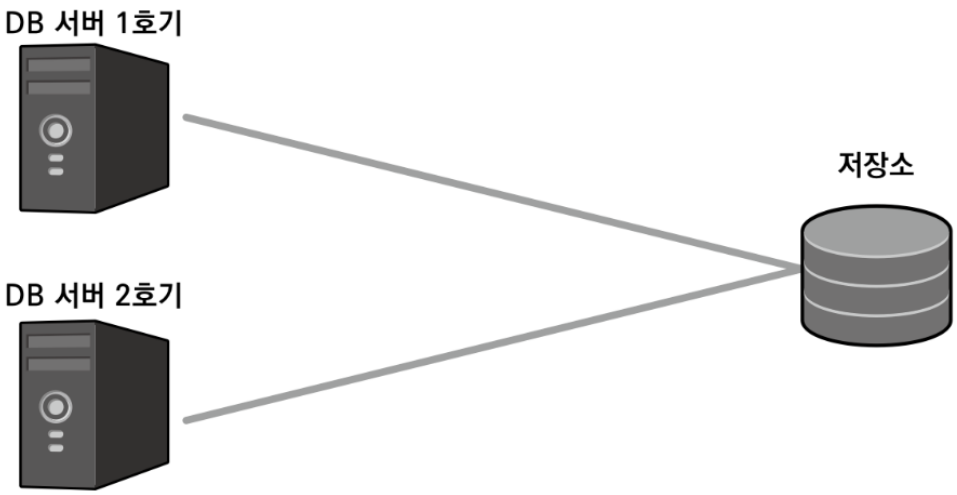
## DB 서버의 다중화 - 클러스터링




데이터베이스의 아키텍처 패턴

- DB 서버는 데이터를 보존하는 영속 계층이라 다중화에 대해 고려할 부분이 많다.
- 데이터는 항상 갱신되기 때문에 다중화를 유지하는 도중 데이터의 정합성을 의식해야 한다.

가장 단순한 다중화 구성

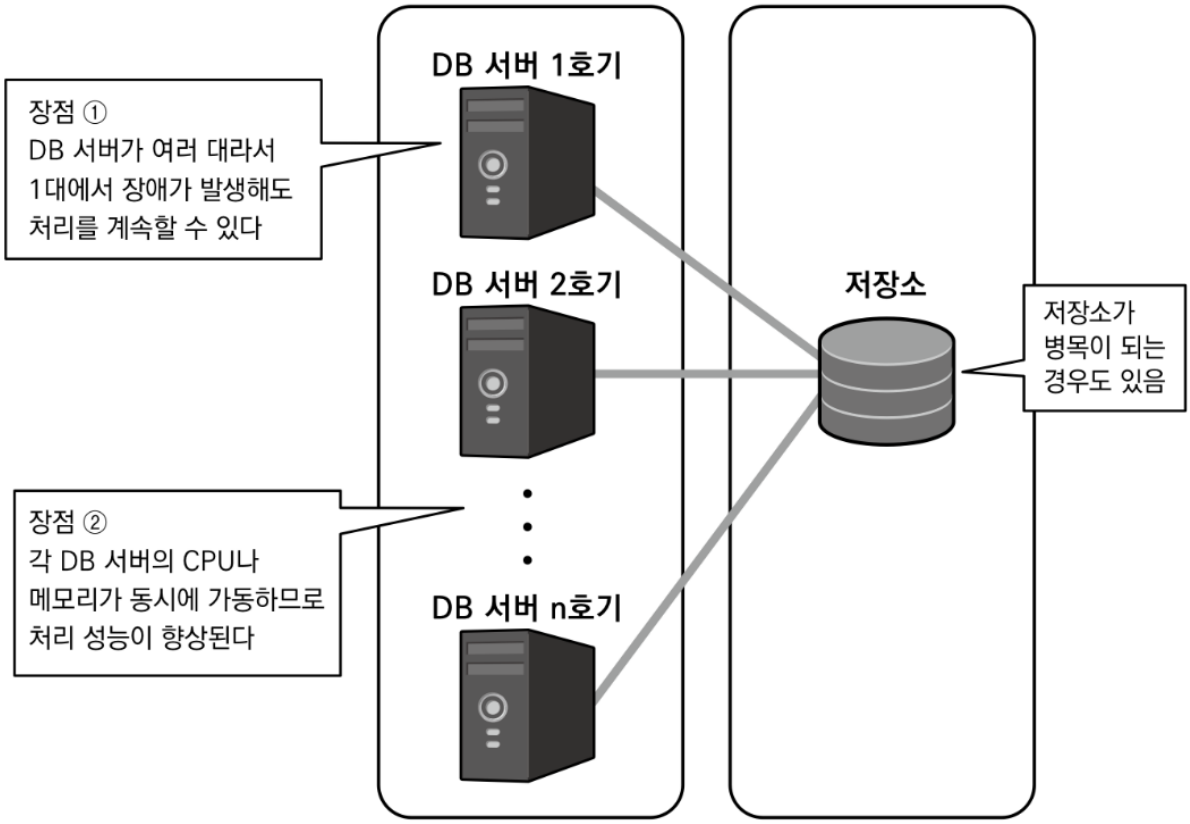



- DB 서버만을 다중화, 저장소는 하나만 사용
- 이 경우 저장소가 하나이기 때문에 데이터 정합성을 고려 할 필요 없음

Active, Standby 클러스터 구성

- 여러대의 DB 서버가 있을 때 여러 DB 서버가 동시에 동작하게끔 할 것인가?

`Active-Active`



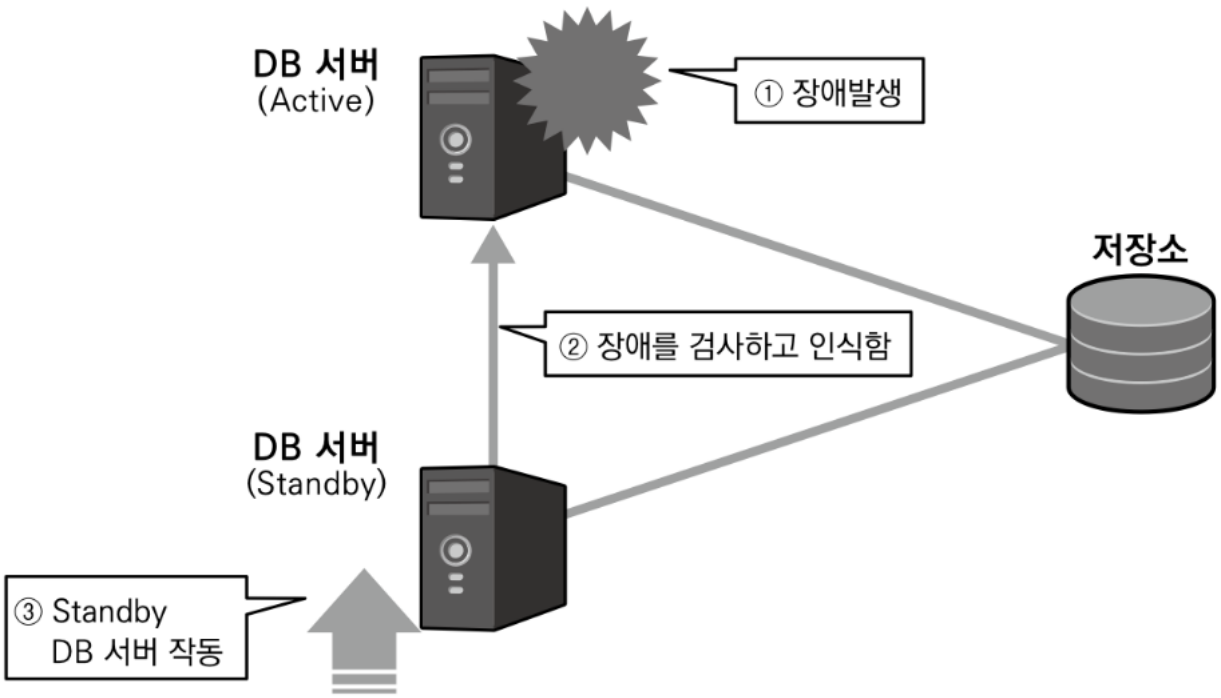

- 클러스터를 구성하는 컴포넌트를 동시에 가동.

장점

- 시스템 다운 시간이 짧다. (한 서버가 죽어도 동시에 가동중인 다른 서버가 처리해줄 수 있다.)
- 성능이 좋다. (동시에 가동하는 CPU나 메모리도 증가하기 때문에 성능이 증가한다)

  단 저장소가 병목이 되어 기대했던 만큼 성능 증가 효과를 보지 못할 수는 있다.

`Active-Standby`




- 보통 Active DB 서버만 작동하다가 Active 서버에 장애가 생기면 Standby DB 서버가 작동한다.

단점

- Active DB 가 죽고, Standby DB 가 작동될 때까지 그 텀 동안 다운 상태가 된다.

Heartbeat?

- Standby DB 서버가 일정 간격으로 Active DB에 이상이 없는지를 조사하기 위한 통신
- Active DB 가 죽으면 이 heart beat 가 끊기기에 Active 가 죽었다는 것을 알게 되고, Standby DB 서버가 작동하게 된다

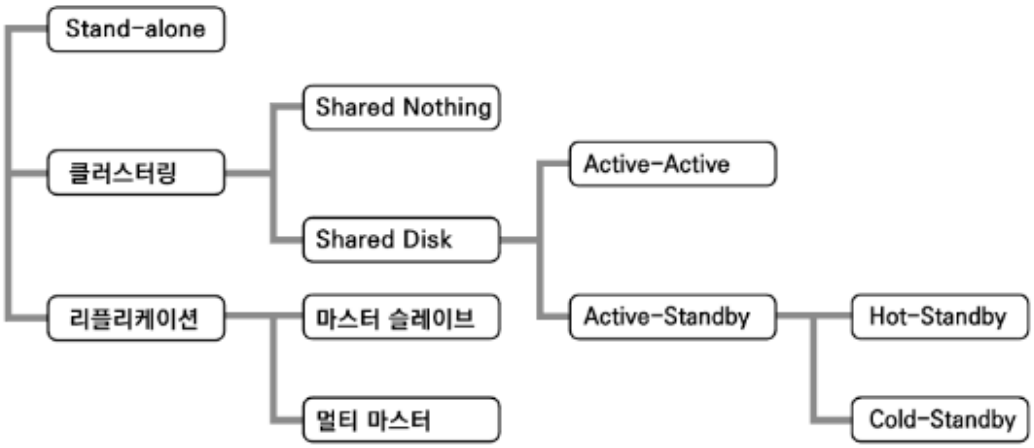
1. Cold Standby

   평소에는 Standby DB 가 작동하지 않다가 Active DB 가 다운된 시점에만 동작

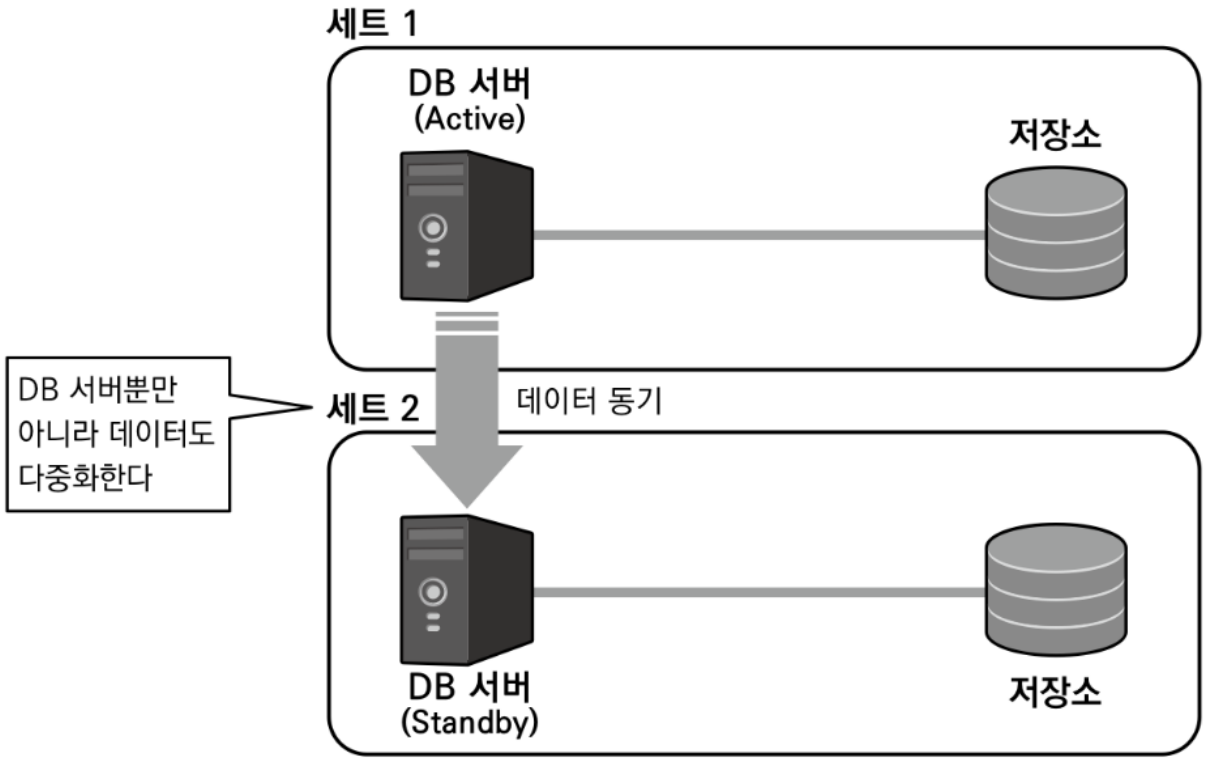
2. Hot Standby

   평소에도 Standby DB 가 작동하는 구성

## DB 서버의 다중화 - 리플리케이션




데이터베이스의 아키텍처 패턴




- Active Standby 클러스터 구성은 저장소 부분은 다중화 하지 않지만

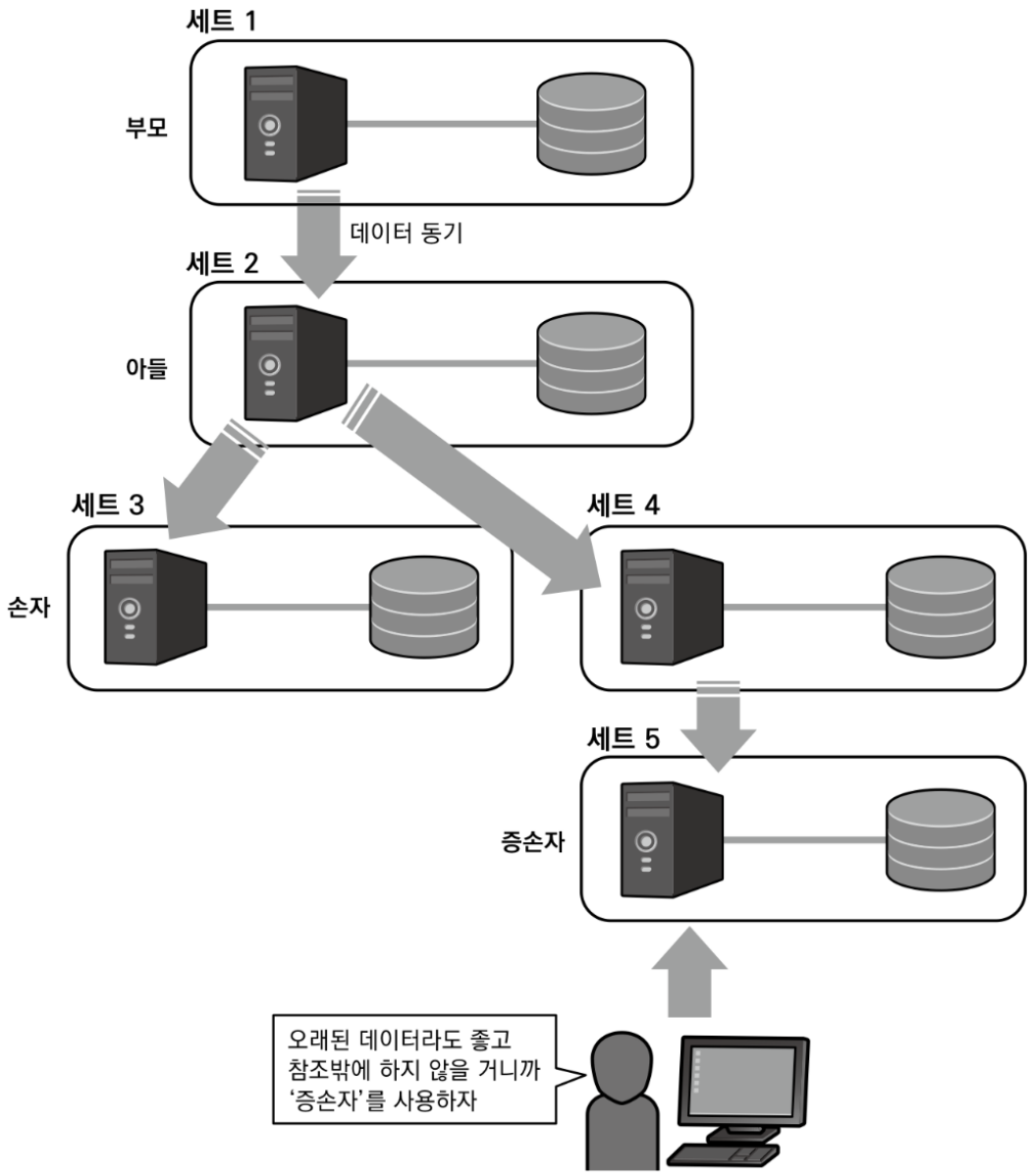
- 리플리케이션은 DB 서버뿐만 아니라 저장소도 다중화한다.
- 데이터 센터에 장애가 생기는 경우 저장소가 부서지는 경우를 대비! (+ 부하 분산의 효과)
- 보통 저장소도 내부 컴포넌트가 다중화되어 있긴 함 - `RAID 기술`

주의할 점

- Active 측 저장소의 데이터는 항상 클라이언트로부터 갱신될 수 있지만,

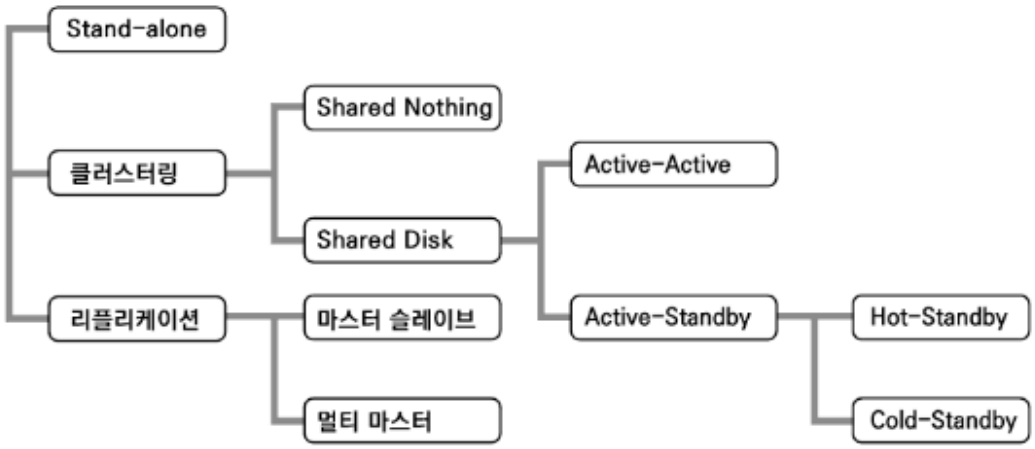
  Standby 측 DB 는 주기적인 텀을 가지고 Active DB 에서 데이터를 갱신한다.
- Standby 측 DB 의 데이터의 최신화를 잘 고려해야 한다 - `동기화`
- `데이터 정합성`을 맞춰줘야 한다는 것

리플리케이션 - 피라미드 형태

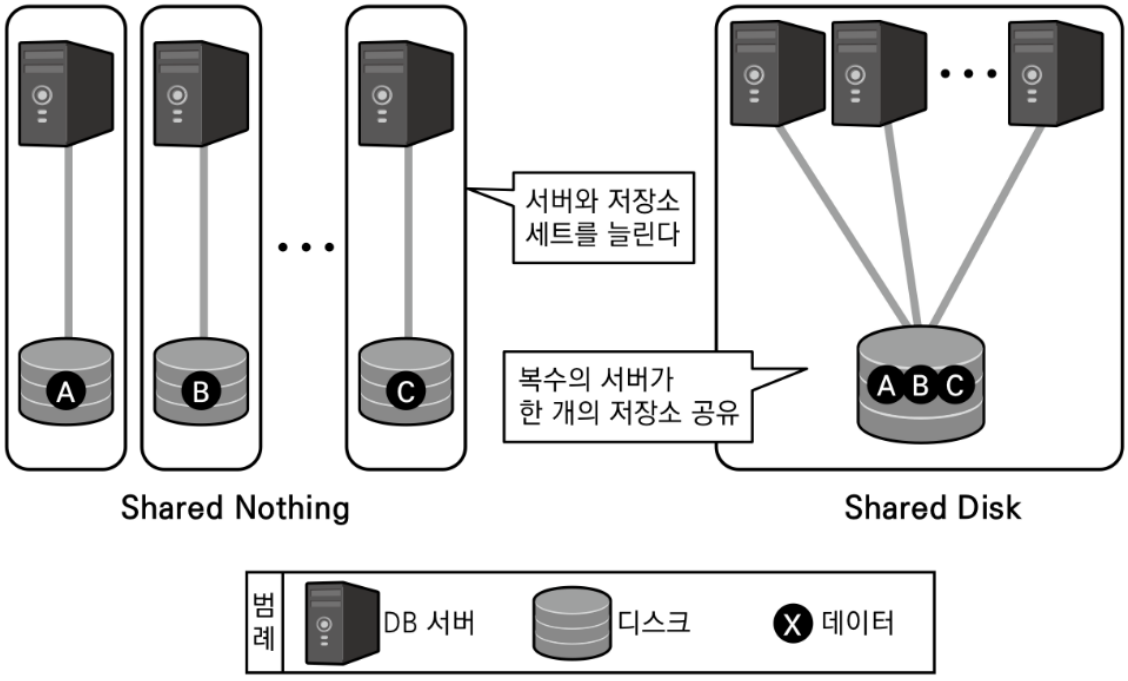



- 오래된 데이터라도 괜찮고 쓰기가 아니라 참조만 해도 괜찮은 경우
- 부모 DB 에 걸리는 부하를 분산 시킬 수 있다.

## 성능을 추구하기 위한 다중화 - Shared Nothing




데이터베이스의 아키텍처 패턴

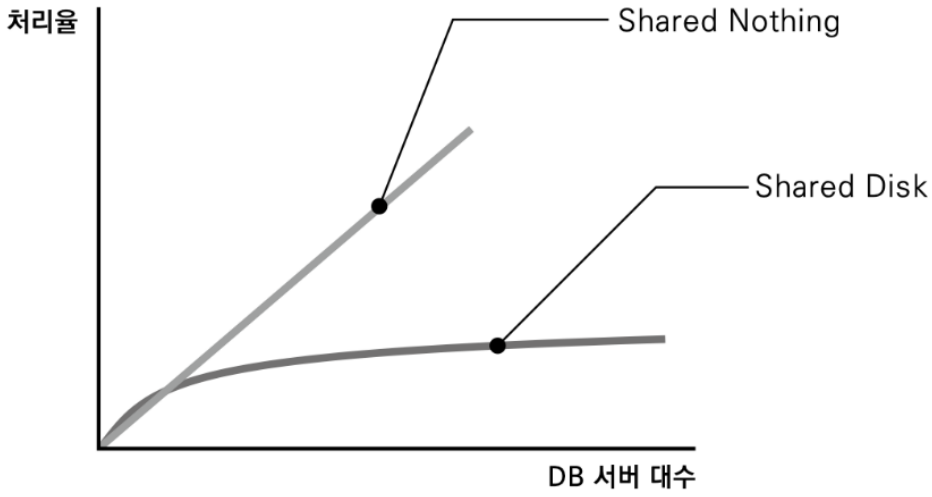



Shared Disk

- 복수의 서버가 1대의 디스크를 사용
- DB 서버를 늘려도 처리율이 무한으로 늘지 않고 어딘가 한계점이 존재한다.
- 저장소가 공유자원이므로 DB 서버간 정보 공유를 위한 오버헤드가 크기 때문

Shared Nothing

- 저장소를 공유하는 Shared Disk 와 달리, 네트워크 이외의 자원을 모두 분리하는 방식.
- 서버와 저장소의 세트를 늘리면 병렬 처리로 인해 성능이 선형적으로 증가함.




- Google - `Sharding`
- Shared Disk 방식은 복잡한 동기화 구조를 사용하기에 구축이 복잡하지만 Shared Nothing 방식은 같은 구성의 DB 를 횡으로 나열하기에 구조가 간단하며 DB 서버수에 비례해 저장소가 늘어간다.
- 단, 각각의 DB 서버가 동일한 1개의 데이터에 액세스 할 수 없다는 단점이 있다.

  한 DB 서버가 다운 될 때 다른 DB 서버가 이를 이어받아 처리할 수 있는 `커버링 구성`을 고려해야 한다.