# 6장 - 키-값 저장소 설계

**키-값 저장소**는 **키-값 데이터베이스**라고도 불리는 **비 관계형(non-relational) 데이터베이스**이다.
이 저장소에 저장되는 값은
**고유 식별자(identifier)**를 키로 가져야 한다.
키와 값 사이의 연결 관계를
**키-값 쌍(pair)**라 한다.

**키는 유일**해야 하며, 해당 키에 매달린 **값은 키를 통해서만 접근**할 수 있다.

- 이때 키는 유일하기만 하면 된다. (**해시**여도 되고, **일반 텍스트**여도 됨)
  - ex) **해시** : 253DDEC4
  - ex) **일반 텍스트** : last_logged_in_at
- 성능상의 이슈로 **키는 짧을수록 좋음**

**값**은 **문자열**일 수도 있고, **리스트**, **객체** 등등 **다양**한다.

- 값은 무엇이 오든 상관하지 않음

키-값 저장소로 널리 알려진 것은 다음과 같다.

- Amazon DynamoDB
- memcached
- redis
- 등
- `put(key, value)`는 키 값 쌍을 **저장소에 저장**

  `get(key)`는 인자로 주어진 **키의 값을 꺼낸다**.

## 문제 이해 및 설계 범위 확정

> 다음의 특성을 갖는 키-값 저장소 설계할 것임

- 키-값 쌍의 크기는 10KB 이하이다.
- 큰 데이터를 저장할 수 있어야 한다.
- 높은 가용성을 제공해야 한다. 따라서 시스템은 설사 장애가 있더라도 빨리 응답해야 한다.
- 높은 규모 확장성을 제공해야 한다. 따라서 트래픽 양에 따라 자동적으로 서버 증설/삭제가 이루어져야 한다.
- 데이터 이관성 수준은 조정이 가능해야 한다.
- 응답 지연시간(latency)이 짧아야 한다.

# 단일 서버 키-값 저장소

단일 서버의 키-값 저장소는 키-값 쌍 전부를 메모리 해시 테이블로  저장하면 된다.
하지만 이렇게 될 경우 데이터 압축(compression), 메모리 디스크 분리 등에 있어 단점이 많다.

어차피 결국 **많은 데이터를 저장하려면 분산 키-값 저장소(distributed key-value store)**를 만들 필요가 있다.

# 분산 키-값 저장소

분산 해시 테이블이라고도 불리며 키-값 쌍을 여러 서버에 분산시키는 것이다.
분산 시스템을 설계할 때는
**CAP 정리**를 이해해야 한다.

## CAP 정리 (Consistency, Availability, Partition Tolerance theorem)

CAP 정리는 데이터 일관성(consistency), 가용성(availability), 파티션 감내성(partition tolerance)라는 세 가지 요구사항을 **동시에 만족하는** 분산 시스템을 설계하는 것은 **불가능**하다는 점이다.

## 각 요구사항의 의미

- **데이터 일관성(consistency)**
  - 분산 시스템에 접속하는 **모든 클라이언트는** 어떤 노드에 접속했는지 관계없이 **언제나 같은 데이터**를 보게
- **가용성(availability)**
  - 분산 시스템에 접속하는 클라이언트는 일부 노드에 장애가 있어도 **항상 응답을 받을 수 있어야**
- **파티션 감내성(partition tolerance)**
  - 두 노드 사이에 통신 장애가 발생하였음을 의미
  - 파티션 감내는 네트워크에 파티션이 생기더라도 시스템은 계속 동작해야

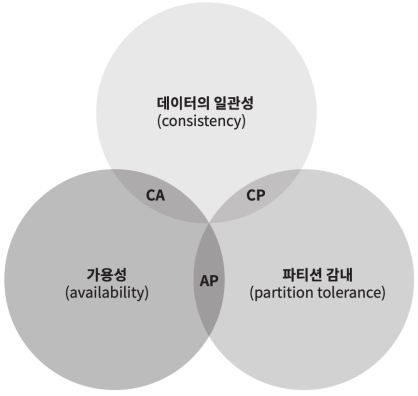
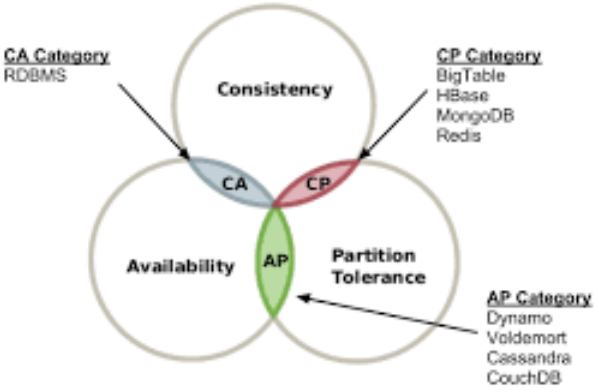



그림 6-1




- **CA** 시스템 : 일관성과 가용성을 지원하는 키-값 저장소
  - 파티션 감내는 지원하지 않음

    하지만 통상 네트워크 장애는 피할 수 없는 일이므로,
    분산 시스템은 반드시 파티션 문제를 감내할 수 있도록 설계되어야 한다.

    **즉, 실세계에 CA 시스템은 존재하지 않는다**.
    - 응..? 그럼 인터넷에 있는 CA 하면서 나오는 RDBMS는 뭐지? - RDB가 나오면 안 됨 (스터디 질문 참고)
- **CP** 시스템 : 일관성과 파티션 감내를 지원하는 키-값 저장소, **가용성 희생**
- **AP** 시스템 : 가용성과 파티션 감내를 지원하는 키-값 저장소, **데이터 일관성 희생**

### *CAP 이론을 이해하기 위한 몇 가지 구체적인 사례*

- 분산 시스템에서 데이터는 보통 여러 노드에 복제되어 보관
  세 대의 복제(replica) 노드 n1, n2, n3에 데이터를 복제하여 보관하는 상황이 다음 그림과 같음

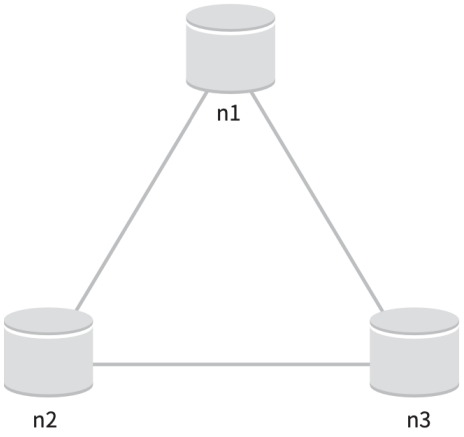



그림 6-2

## 이상적 상태

이상적 환경이라면 네트워크가 파티션되는 상황을 절대로 발생하지 않음.
n1에 기록된 데이터는 자동적으로 n2와 n3에 복제 됨 -
**데이터 일관성**과 **가용성** 만족

## 실세계의 분산 시스템

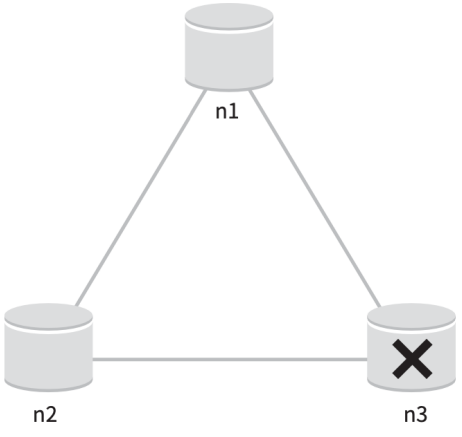



그림 6-3

분산 시스템은 파티션 문제를 피할 수 없음.
따라서 파티션 문제가 발생하면
**일관성과 가용상 사이**에서 **하나를 선택**해야 함
위 그림에서 n3가 장애가 발생했을 때 n1 또는 n2에 기록한 데이터는 n3에 전달되지 않는다.
또한, n3에 기록되었으나 아직 n1 및 n2로 전달되지 않은 데이터가 있다면 n1와 n2는 오래된 사본을 갖게 된다.

- **가용성 대신 일관성을 선택 (CP 시스템, 은행권)**
  - 세 서버 사이에 생길 수 있는 **데이터 불일치 문제를 피하기 위해** n1과 n2에 대해 **쓰기 연산 중단**
    이렇게 하면
    **가용성이 깨진다**.
  - ex) **은행권 시스템**은 보통 **데이터 일관성을 양보하지 않음 (일관성을 선택함)**
    - 온라인 뱅킹 시스템이 계좌 최신 정보를 출력하지 못한다면 큰 문제가 생긴다.
    - 네트워크 파티션 때문에 일관성이 깨질 수 있는 상황이 발생하면 이런 시스템은 상황이 해결될 때까지 오류를 반환하게 됨

- **일관성 대신 가용성을 선택 (AP 시스템, )**
  - 낡은 데이터를 반환할 위험이 있어도 계속 **읽기 연산을 허용**
  - n1과 n2 계속 **쓰기 연산 허용**
  - 파티션 문제가 해결된 뒤에 새 데이터를 n3에 전송

---

# 시스템 컴포넌트

> 키-값 저장소 구현에 사용될 핵심 컴포넌트와 기술들은 다음과 같음

- 데이터 파티션
- 데이터 다중화(replication)
- 일관성(consistency)\
- 일관성 불일치 해소(inconsistency resolution)

- 장애 처리
- 시스템 아키텍처 다이어그램
- 쓰기 경로(wirte path)
- 읽기 경로(read path)
- 이번 절에 다루는 내용은 다이나모, 카산드라, 빅테이블의 사례를 참고했다.

## 데이터 파티션

대규모 애플리케이션의 경우 전체 데이터를 한 대 서버에 욱여넣는 것은 불가능하다.

**가장 단순한 해결책은 데이터를 작은 파티션들로 분할한 다음 여러 대 서버에 저장하는 것이 가능**하다.

데이터를 **파티션 단위로 나눌 때 다음 두 가지 문제** 고려

1. 데이터를 여러 서버에 고르게 분산할 수 있는가
2. 노드에 추가되거나 삭제될 때 데이터의 이동을 최소화할 수 있는가

이전 장(5장)에서 공부한 안정 해시(consistent hash)는 이런 문제를 해결하는데 적합한 기술이다.

▼ 안정 해시의 동작 원리

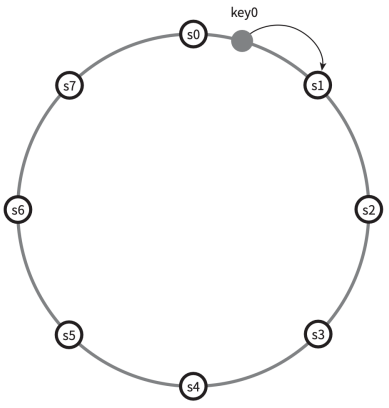



그림 6-4

- 서버를 해시 링(hash ring)에 배치한다.

- 어떤 키-값 쌍을 어떤 서버에 저장할지 결정하려면 우선 해당 키를 같은 링 위에 배치한다.
  그 지점으로부터 링을 시계 방향으로 순회하다 만나는 첫 번째 서버가 바로 해당 키-값 쌍을 저장할 서버

▼ 안정 해시를 사용한 이점

- **규모 확장 자동화(automatic scaling)**
  - 시스템 부하에 따라 서버가 자동으로 추가되거나 삭제되도록 만들 수 있음
- **다양성(heterogeneity)**
  - 각 서버의 용량에 맞게 가상 노드(virtual node) 수를 조정할 수 있음
  - 즉, **고성능 서버는 더 많은 가상 노드를 갖도록** 설정할 수 있음

## 데이터 다중화

높은 가용성과 안정성을 확보하기 위해 데이터를 `N`개 서버에 비동기적으로 다중화(replication)할 필요가 있다.

- `N`은 튜닝 가능한 값

N개 서버를 선정하는 방법은 어떤 키를 링 위에 배치한 후,
그 지점으로부터 시계 방향으로 링을 순회하면서 만나는 첫 N개 서버에 데이터 사본을 보관하는 것이다.

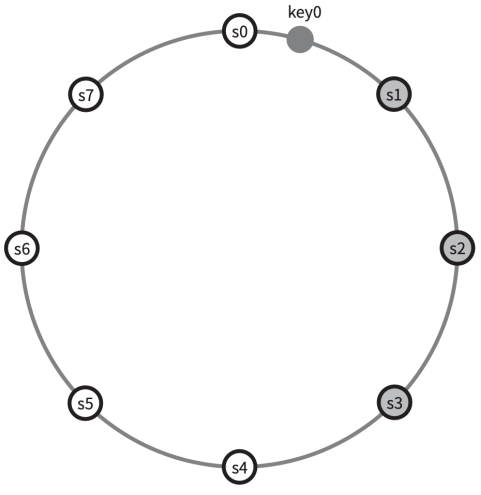



그림 6-5

가상 노드를 사용하다보면 위처럼 N개의 노드가 대응될 실제 물리 서버 개수가 N보다 작아질 수 있다
이 문제를 피하려면 노드를 선택할 때 같은 물리 서버를 중복 선택하지 않도록 해야 한다.

- 이 경우가 이해가 잘 되지 않음.. (98p)

같은 데이터 센터에 속한 노드는 정전, 네트워크 이슈, 자연재해 등의 문제를 동시에 겪을 가능성이 있다.
따라서
**안정성을 담보하기 위해 데이터의 사본은 다른 센터에 보관하고, 센터들은 고속 네트워크로 연결**한다.

## 데이터 일관성

**여러 노드에 다중화된 데이터는 동기화가 되어야** 한다.

**정족수 합의(Quorum Consensus)** 프로토콜을 사용하면 읽기/쓰기 연산 모두에 일관성을 보장할 수 있다.

- **N = 사본 개수**
- **W = 쓰기 연산에 대한 정족수**
  - 쓰기 연산을 성공하려면 **적어도 W개의 서버로부터 쓰기 연산이 성공했다는 응답**을 받아야 함
- **R = 읽기 연산에 대한 정족수**
  - 읽기 연산을 성공하려면 **적어도 R개의 서버로부터 응답을 받아야** 한다.

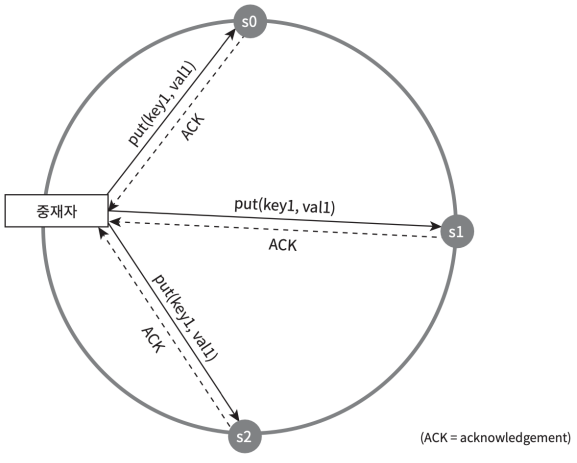



그림 6-6

`N = 3`인 경우

W = 1은 데이터가 한 대 서버에만 기록된다는 뜻이 아니라, 쓰기 연산이 성공했다는 판단하기 위해

**중재자(coordinator)는 최소 한 대 서버로부터 쓰기 성공 응답을 받아야 한다는 뜻**이다.

위 예시로 보면 s1으로부터 성공을 응답을 받았으면 s0, s2로부터 응답을 기다릴 필요가 없음

중재자는 클라이언트와 노드 사이에서 프록시(Proxy) 역할을 함

`W`, `R`, `N`의 값을 정하는 것은 **응답 지연과 데이터 일관성 사이의 타협점을 찾는 전형적인 과정**이다.

- W = 1 or R = 1인 구성인 경우 중재자는 한 대 서버로부터 응답을 받으면 되니 응답 속도는 빠르다.
  하지만 데이터 일관성은 낮아진다. 또한, 1보다 크면 데이터 일관성은 좋아지지만 응답 속도가 느려진다.

`W + R > N`인 경우에는 **강한 일관성(strong consistency)을 보장**한다.

- 일관성을 보증할 초신 데이터를 가진 노드가 최소 하나는 겹치기 때문

> 면접에서 `N`, `W`, `R`의 값을 어떻게 정하면 좋을지

- `R = 1`, `W = N` : 빠른 읽기 연산에 최적화된 시스템
- `W = 1`, `R = N` : 빠른 쓰기 연산에 최적화된 시스템
- `W + R > N` : 강한 일관성이 보장됨 (보통 N=3, W = R = 2)
- `W + R ≤ N` : 강한 일관성이 보장되지 않음
- 요구되는 일관성 수준에 따라 W, R, N의 값을 조정하면 됨

## 일관성 모델(consistency model) - 데이터 일관성의 수준을 결정, 종류 다양

- **강한 일관성(strong consistency)**
  - 모든 읽기 연산은 가장 최근에 갱신된 결과를 반환

    - 클라이언트는 절대로 낡은(out-of-date) 데이터를 보지 못 함
- **약한 일관성(weak consistency)**
    - 읽기 연산은 가장 최근에 갱신된 결과를 반환하지 못할 수 있음
- **최종 일관성(eventual consistency)**
    - 약한 일관성의 한 형태
    - 갱신 결과가 결국에는 모든 사본에 반영(즉, 동기화) 되는 모델

## 강한 일관성

- 모든 사본에 현재 쓰기 연산의 결과가 반영될 때까지 해당 데이터에 대한 읽기/쓰기를 금지
- **새로운 요청의 처리가 중단**되기 때문에 고가용성(HA) 시스템에서는 적합하지 않음

## 최종 일관성 - **다이나모, 카산드라**는 최종 일관성 모델을 따름

- 쓰기 연산이 병렬적으로 발생하면 시스템에 저장된 값의 일관성이 깨질 수 있음.
    - 이 문제는 클라이언트가 해결해야 됨

## 클라이언트가 **일관성이 깨진 데이터를 읽지 않도록**

- **비 일관성 해소 기법 : 데이터 버저닝** (낙관락이랑 비슷?)
    - **버저닝(versioning)**과 **벡터 시계(vector clock)**는 데이터를 다중화했을 때 **사본 간 일관성이 깨지는 문제를 해결**한다.
        - **버저닝** : 데이터를 **변경할 때마다 새로운 버전**을 만듦 (따라서 각 버전 데이터는 immutable, 변경 X)
        - **벡터 시계** : [서버, 버전]의 순서쌍을 데이터에 매단 것
            - 어떤 버전이 선행 버전인지, 후행 버전인지, 다른 버전과 충돌이 있는지 판별하는데 사용
            - `D([S1, v1], [S2, v2], …, [Sn, vn])`와 같이 표현된다면,

`D`는 **데이터**, `vi`는 **버전 카운터**, `Si`는 **서버 번호**

- [Si, vi]가 있으면 vi 증가
  그렇지 않으면 새 항목 [Si, 1]을 생성

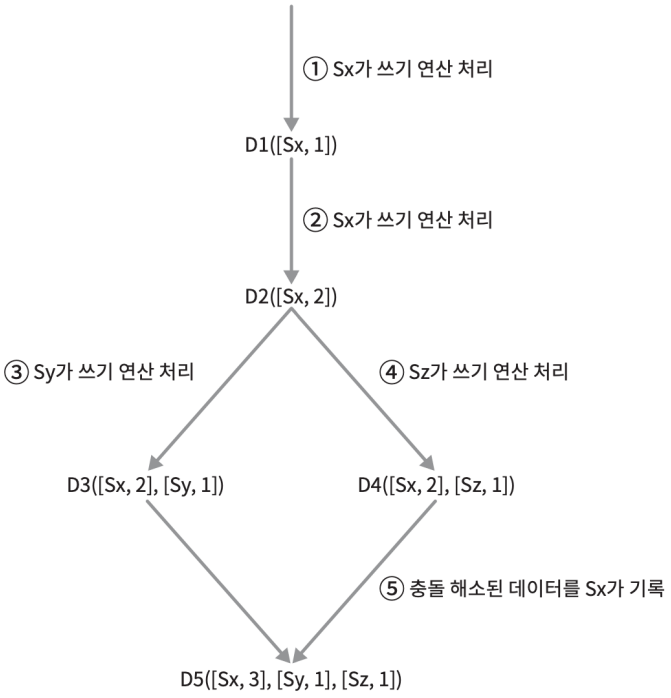



그림 6-9

1. 클라이언트가 데이터 D1을 시스템에 기록
   (쓰기 연산을 처리한 서버는 Sx,
   `벡터 시계` `D1([Sx, 1])`으로 변경
2. 다른 클라이언트가 데이터 D1을 읽고 D2로 업데이트한 다음 기록
   D2는 D1에 대한 변경이므로 D1을 덮어씀
   쓰기 연산은 같은 서버 Sx가 처리,
   `벡터 시계` `D2([Sx, 2])`로 변경
3. 다른 클라이언트가 D2를 읽어 D3로 갱신한 다음 기록
   (쓰기 연산을 처리한 서버는 Sy,
   `벡터 시계` `D3([Sx, 2], [Sy, 1])`로 변경
4. 또 다른 클라이언트가 D2를 읽고 D4로 갱신
   쓰기 연산은 Sz가 처리,
   `벡터 시계` `D4([Sx, 2], [Sz, 1])`
5. 어떤 클라이언트가 D3와 D4를 읽으면 데이터 간 충돌을 알게 됨
   D2를 Sy와 Sz가 각각 다른 값으로 변경했기 때문
   이런 충돌은 클라이언트가 해소한 후 서버에 기록된다.

이 쓰기 연산을 처리한 서버는 Sx였다면, 벡터 시계는 D5([Sx, 3], [Sy, 1], [Sz, 1])로 변경

### 벡터 시계를 사용한 충돌 감지의 단점

1. 충돌 감지 및 해소 로직이 클라이언트에 들어가야 되므로, 클라이언트 구현이 복잡하다.
2. `[서버: 버전]`의 순서쌍 개수가 빨리 증가하게 된다.

- 이 문제를 해결하려면 길이에 임계치(threshold)를 설정하고, 임계치 이상으로 길이가 길어지면 오래된 순서쌍을 벡 터시계에서 제거하면 좋다.
    - 단, 버전 간 선후 관계까 정확하게 결정될 수 없기 때문에 충돌 해소 과정의 효율성이 떨어짐
    - 하지만 아마존 다이나모는 실제 서비스에서 그런 문제가 벌어지는 것을 발견한 적이 없음
    - 따라서 **대부분의 기업에서 벡터 시계를 적용해도 괜찮은 솔루션**이다.

## 장애 처리

**장애 감지(failure detection) 기법**들을 살펴보고 **장애 해소(failure resolution) 전략**을 살펴보자.

### 장애 감지

분산 시스템에서 서버 한 대가 지금 서버 A가 죽었다고 한다고 서버 A를 장애처리하지 않는다.
보통
**두 대 이상의 서버가 똑같이 서버 A의 장애를 보고해야 해당 서버가 실제로 장애가 발생했다고 간주**

다음 그림처럼 **모든 노드 사이에 멀티캐스팅(multicasting) 채널**을 구축하는 것 (네트워크 브로드 캐스트?)

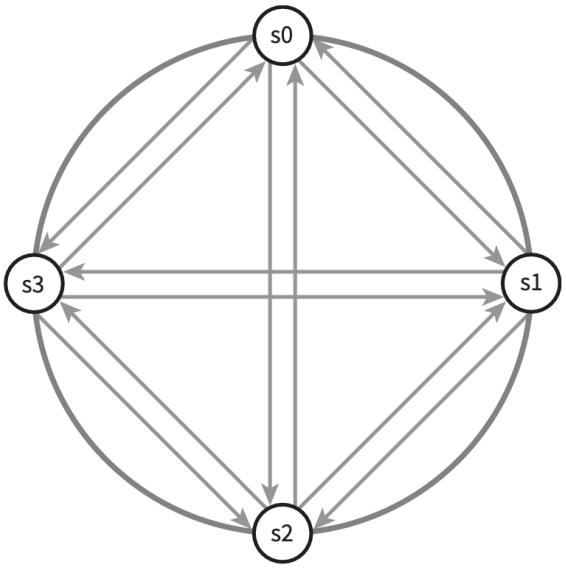



그림 6-10

따라서 **가십 프로토콜(gossip porotocol)** 같은

**분산형 장애 감지(decentralized failure detection) 솔루션을 채택**하는 편이 좋다.

### 가십 프로토콜의 동작 원리

**s0's membership list**

| Member ID | Heartbeat counter | Time |
|---|---|---|
| 0 | 10232 | 12:00:01 |
| 1 | 10224 | 12:00:10 |
| 2 | 9908 | 11:58:02 |
| 3 | 10237 | 12:00:20 |
| 4 | 10234 | 12:00:34 |

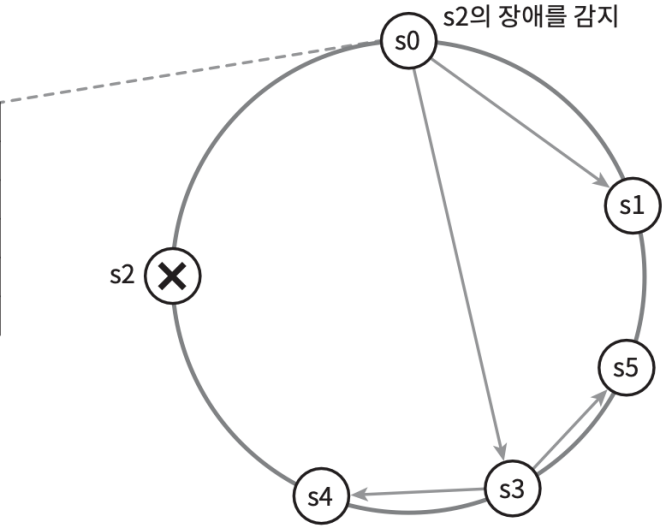



그림 6-11

책 106p

- 각 노드는 **멤버십 목록(membership list)를 유지**한다.
  - 멤버십 목록 : 각 멤버 ID와 박동 카운터(heartbeat counter) 쌍의 목록

- 각 노드는 주기적으로 자신의 박동 카운터를 증가시킨다.
- 각 노드는 무작위로 선정된 노드들에게 주기적으로 자기 박동 카운터 목록을 보낸다.
- 박동 카운터 목록을 받은 노드는 멤버십 목록을 최신 값으로 갱신한다.
- 어떤 멤버의 박동 카운터 값이 지정된 시간 동안 갱신되지 않으면 해당 멤버는 장애(offline) 상태

**일시적 장애 처리** - 시스템 가용성을 보장하기 위해 장애를 감지한 시스템에 대한 조치를 취하는 방법.

- **엄격한 정족수(strict quorum) 접근법**
  - `읽기`와 `쓰기` 연산 **금지**
- **느슨한 정족수(sloppy quorum) 접근법 - 다이나모가 이 방식**
  - 정족수 요구사항을 강제하는 대신,

    `쓰기 연산`을 수행할 **W개의 건강한 서버**와 `읽기 연산`을 수행할 **R개의 건상한 서버**를 **해시 링에서 선택**
  - 이때 장애 상태인 서버는 무시
- 장애 상태인 서버로 요청이 가는 경우 해당 **서버가 복구되었을 때 일괄 반영하여 데이터 일관성을 보장**한다.
  - 이를 위해 임시로 쓰기 연산을 처리한 서버에는 그에 관한 hint를 남겨둔다.
  - 따라서 이런 장애 처리 방은 **단서 후 임시 위탁(hinted handoff) 기법**이라 부름

**영구 장애 처리**

**반 엔트로피(anti-entropy) 프로토콜**을 구현하여 사본을 동기화하면 영구적인 노드 장애 상태를 처리할 수 있음

- **반 엔트로피(anti-entropy) 프로토콜** : 사본들을 비교하여 최신 버전으로 갱신하는 과정을 포함
- 사본 간의 일관성이 망가진 상태를 탐지하고 전송 데이터의 양을 줄이기 위해 **머클(Merkle) 트리**를 사용
  - 머클 트리 : 해시 트리라고도 불림,
  - 각 노드에 자식 노드들에 보관된 값의 해시(자식 노드가 종단 leaf 노드인 경우), 또는 자식 노드들의 레이블로부터 계산된 해시 값을 레이블로 붙여두는 트리

- 해시 트리를 사용하면 대규모 자료 구조의 내용을 효과적이면서도 보안상 안전한 방법으로 검증할 수 있음

**데이터 센터 장애 처리**

정전, 네트워크 장애, 자연재해 등 다양한 이유로 발생할 수 있음
데이터 센터 장애에 대응할 수 있는 시스템을 만들려면 데이터를
**여러 데이터 센터에 다중화**하는 것이 중요

## 장애 해소

???????? 이부분이 나왔었나 ???????, 일시적 장애 처리, 영구 장애 처리, 데이터 센터 장애 처리 이런건가?

## 시스템 아키텍처 다이어그램

- 클라이언트는 키-값 저장소가 제공하는 두 가지 단순한 API, 즉 get(key) 및 put(key, value)와 통신
- 중재자(coordinator)는 클라이언트에게 키-값 저장소에 대한 프록시(Proxy) 역할을 하는 노드
- 노드는 안정 해시(consistenct hash)의 해시 링(hash ring) 위에 분포

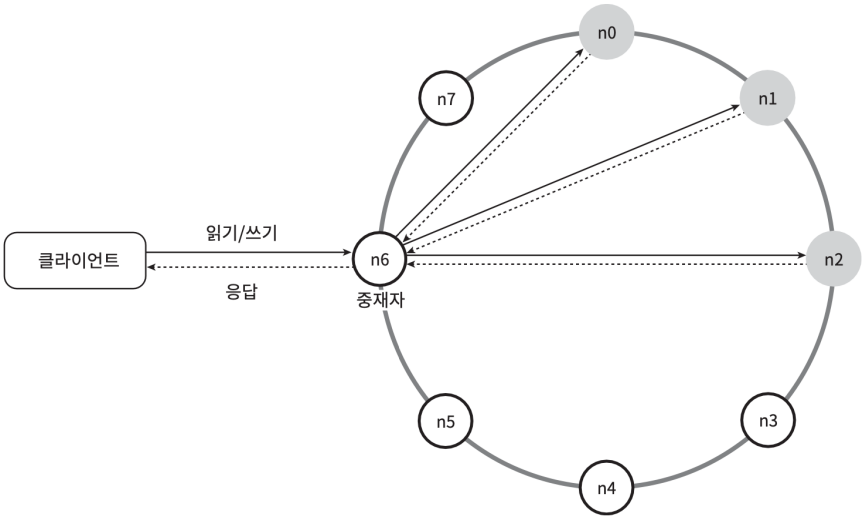



그림 6-17

- 노드를 자동으로 추가 또는 삭제할 수 있또록, 시스템은 완전히 분산(decentralized)

- 데이터는 여러 노드에 다중화
- 모든 노드가 같은 책임을 지므로, SPOF는 존재하지 않음

> 완전히 분산된 설계를 골랐으면 **모든 노드는 다음 기능 전부를 지원**해야 됨

- 클라이언트 API
- 장애 감지
- 데이터 충돌 해소
- 장애 복구 매커니즘
- 다중화
- 저장소 엔진
- 등..

## 쓰기 경로

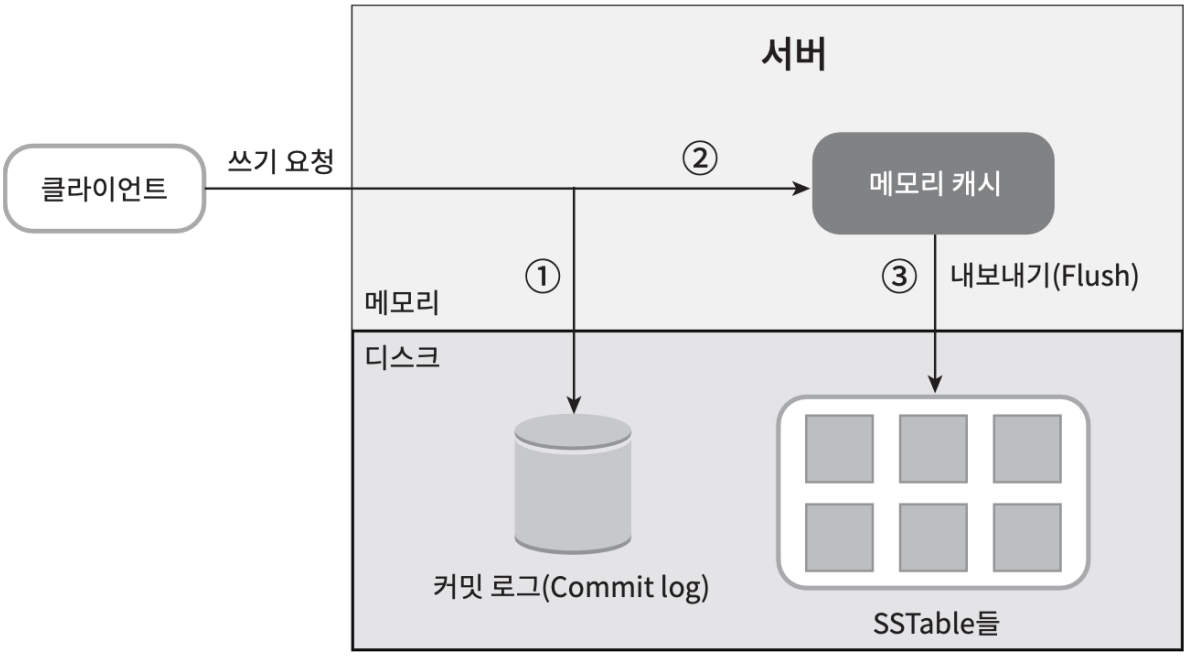



**그림 6-19**

카산드라의 사례를 참고한 것

1. 쓰기 요청이 커밋 로그(commit log) 파일에 기록.
2. 데이터가 메모리 캐시에 기록
3. 메모리 캐시가 가득차거나 사전에 정의된 어떤 임계치에 도달하면 데이터는 디스크에 있는 **SSTable**에 기록
    - `SSTable` : Sorted-String Table의 약어, <키, 값>의 순서쌍을 정렬된 리스트 형태로 관리하는 테이블

## 읽기 경로

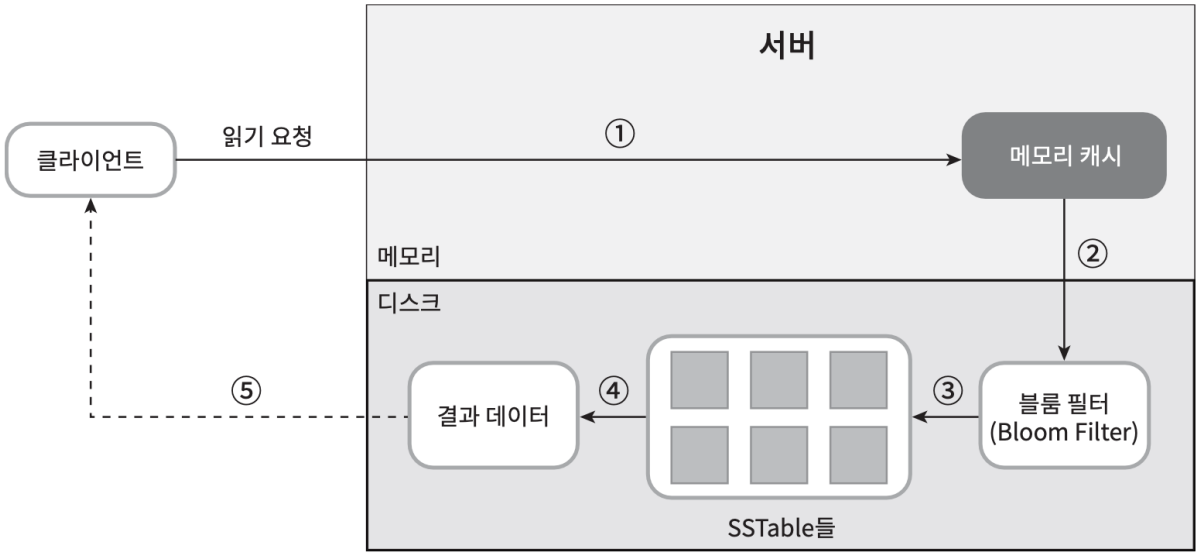



그림 6-21

1. 데이터가 메모리에 있는지 검사. (메모리에 없으면 2로 이동)
2. 데이터가 메모리에 없으므로 **블룸 필터(Bloom Filter)**를 검사
    - `블룸 필터` : 어느 SSTalbe에 찾는 키가 있는지 알아내는 효율적인 방법을 위해 사용
3. 블룸 필터를 통해 어떤 SSTable에 키가 보관되어 있는지 알아냄
4. SSTable에서 데이터를 가져옴
5. 해당 데이터를 클라이언트에게 반환

# 요약

- 분산 키-값 저장소가 가져야 하는 기능과 그 기능 구현에 이용되는 기술 정리

| 목표/문제 | 기술 |
| --- | --- |
| **대규모 데이터 저장** | 안정 해시를 사용해 서버들에 부하 분산 |
| **읽기 연산에 대한 높은 가용성 보장** | 데이터를 여러 데이터센터에 다중화 |
| **쓰기 연산에 대한 높은 가용성 보장** | 버저닝 및 벡터 시계를 사용한 충돌 해소 |
| **데이터 파티션** | 안정 해시 |
| **점진적 규모 확장성** | 안정 해시 |
| **다양성(heterogeneity)** | 안정 해시 |
| **조절 가능한 데이터 일관성** | 장족수 합의(quorum consensus) |
| **일시적 장애 처리** | 느슨한 정족수 프로토콜(sloppy quorum)과<br>단서 후 임시 위탁(hinted handoff) |
| **영구적 장애 처리** | 머클 트리(Merkle tree) |
| **데이터 센터 장애 대응** | 여러 데이터 센터에 걸친 데이터 다중화 |

---

# Chapter 6: DESIGN A KEY-VALUE STORE

1. Amazon DynamoDB
2. memcached
3. Redis
4. Dynamo: Amazon’s Highly Available Key-value Store
5. Cassandra
6. Bigtable: A Distributed Storage System for Structured Data
7. Merkel tree
8. Cassandra Architecture
9. SSTable
10. Bloom filter