6️⃣

231001 6. 키-값 저장소 설계

6. 키-값 저장소 설계

키-값 저장소

비 관계형 데이터베이스

고유 식별자를 키로 가지고, 키와 값 사이의 연결 관계를 “키-값” 쌍이라고 지칭

해당 키에 매달린 값은 키를 통해서만 접근할 수 있음

키에는 일반 텍스트나 해시 값이 들어가는데 성능상의 이유로 짧을수록 좋음

▼ 문제 이해 및 설계 범위 확정

완벽한 설계는 없음

읽기, 쓰기 그리고 메모리 사용량 사이에 어떤 균형을 찾고

데이터의 일관성과 가용성 사이에서 타협적 결정을 내린 설계가 쓸만한 답안

▼ 단일 서버 키-값 저장소

가장 직관적인 방법

키-값 쌍 전부를 메모리에 해시 테이블로 저장

빠른 속도를 보장하지만 모든 데이터를 메모리 안에 두는 것은 불가능할 수도 있음

⇒ 개선책으로 **데이터 압축**이나 **자주 쓰이는 데이터만 메모리에 두고 나머지는 디스크에 저장**하는 방식이 있음

위에 언급된 두 방식으로 개선한다고 해도 서버 한 대로는 부족한 때가 찾아옴

많은 데이터를 저장하기 위해서 **분산 키-값 저장소**를 만들 필요가 있음

▼ 분산 키-값 저장소

(= 분산 해시 테이블)

키-값 쌍을 여러 서버에 분산

▼ CAP 정리

데이터의 일관성, 가용성, 파티션 감내라는 세 가지 요구사항을 동시에 만족하는 분산 시스템을 설계하는 것은 불가능하다는 정리

데이터 일관성 consistency

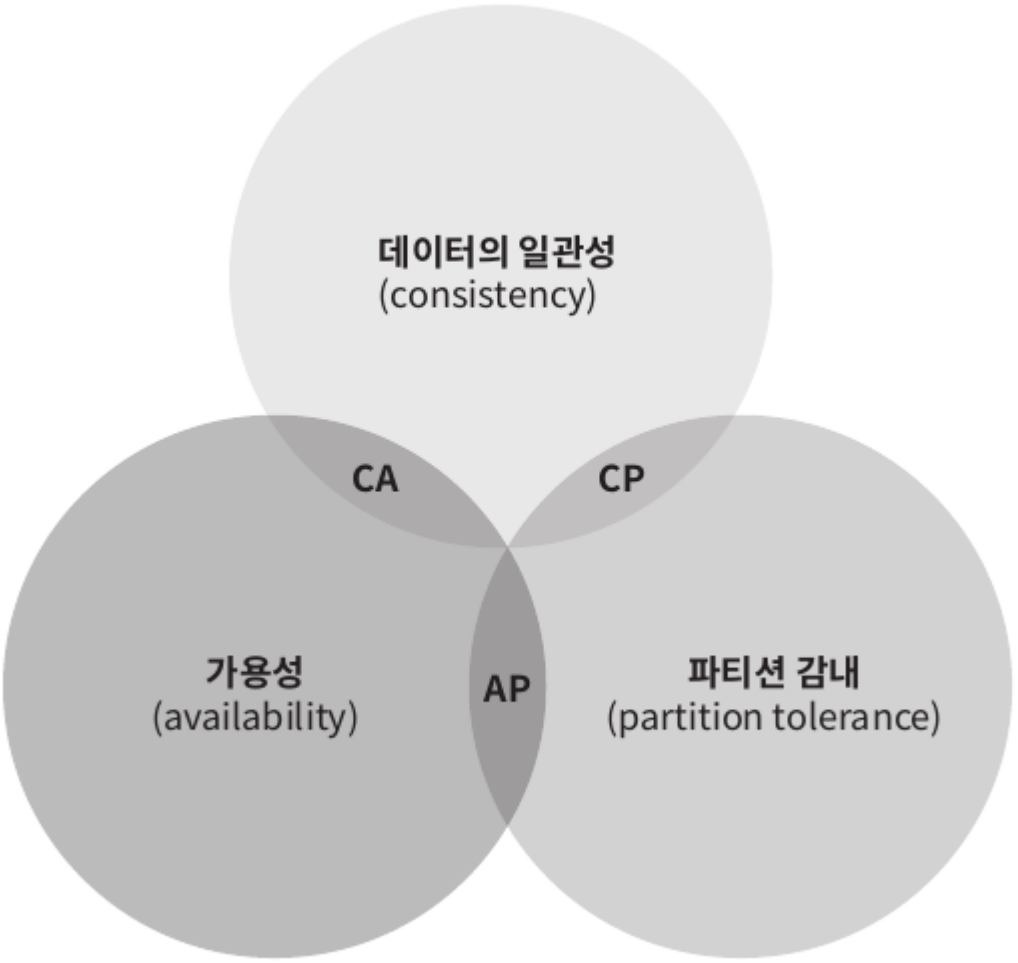
분산 시스템에 접속하는 모든 클라이언트는 어떤 노드에 접속했느냐에 관계없이 언제나 같은 데이터를 보게 되어야 함

가용성 availability

분산 시스템에 접속하는 클라이언트는 일부 노드에 장애가 발생하더라도 항상 응답을 받을 수 있어야 함

파티션 감내 partition tolerance

네트워크에 파티션(두 노드 사이에 통신 장애가 발생)이 생기더라도 시스템은 계속 동작해야 함



두 가지를 충족하려면 나머지 하나는 반드시 희생되어야 함

- CP
  - 일관성과 파티션 감내를 지원
  - 가용성을 희생
- AP
  - 가용성과 파티션 감내를 지원
  - 데이터 일관성을 희생
- CA
  - 일관성과 가용성을 지원
  - 파티션 감내를 희생
  - 통상 네트워크 장애는 피할 수 없는 일로 여겨지므로, 분산 시스템은 반드시 파티션 문제를 감내할 수 있도록 설계되어야 함

- 실세계에서 CA는 존재하지 않음

1. 이상적 상태
   - 네트워크가 파티션되는 상황은 절대로 일어나지 않음
   - 데이터 일관성과 가용성도 만족
2. 실세계의 분산 시스템
   - 파티션 문제를 피할 수 없음
   - 파티션 문제가 발생하면 일관성과 가용성 사이에서 하나를 선택해야 함
     - 일관성 선택
       - 세 서버 사이에 생길 수 있는 데이터 불일치 문제를 피하기 위해 쓰기 연산을 중단시켜야 하는데 이럴 경우 가용성이 깨짐
       - 상황이 해결될 때까지 오류 반환
     - 가용성 선택
       - 낡은 데이터를 반환할 위험이 있더라도 읽기 연산을 허용
       - 멀쩡한 서버에서는 쓰기 연산도 허용
       - 파티션 문제가 해결된 뒤에 데이터 일관성 맞추는 작업 진행

요구사항에 맞도록 CAP 정리를 적용해야 함

이 문제에 대해 면접관과 상의하고, 결론에 따라 시스템 설계

▼ 시스템 컴포넌트

키-값 저장소 구현에 사용될 핵심 컴포넌트 및 기술

▼ 데이터 파티션

대규모 애플리케이션의 경우 전체 데이터를 한 대 서버에 욱여넣는 것은 불가능

데이터를 작은 파티션들로 분할한 후 여러 대 서버에 저장하는 것이 가장 단순한 해결책

데이터를 파티션 단위로 나눌 때 따져봐야 할 문제

- 데이터를 여러 서버에 고르게 분산할 수 있는가

- 노드가 추가되거나 삭제될 때 데이터의 이동을 최소화할 수 있는가

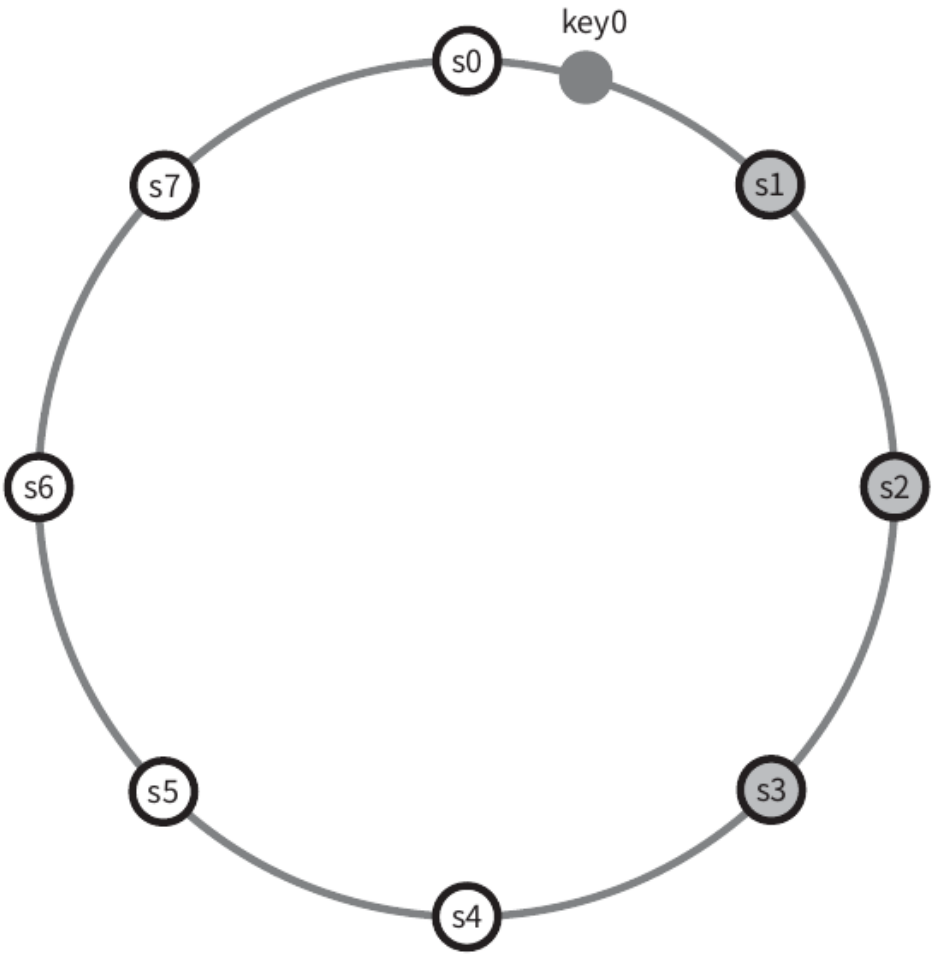
⇒ **안정 해시** 사용

안정 해시를 사용해서 데이터를 파티션하면 좋은 점

- 규모 확장 자동화
  - 시스템 부하에 따라 서버가 자동으로 추가되거나 삭제되도록 만들 수 있음
- 다양성
  - 각 서버의 용량에 맞게 가상 노드의 수 조정 가능
  - 고성능 서버는 더 많은 가상 노드를 갖도록 설정 가능

▼ 데이터 다중화

높은 가용성과 안정성을 확보하기 위해서 데이터를 N개 서버에 비동기적으로 다중화할 필요가 있음



어떤 키를 해시 링 위에 배치하고 그 지점으로부터 시계 방향으로 링을 순회하면서 만나는 첫 N개 서버에 데이터 사본을 보관

가상 노드를 사용하면 실제 물리 서버의 개수가 N보다 작아질 수 있기 때문에 노드를 선택할 때 같은 물리 서버를 중복 선택하지 않도록 해야 함

같은 데이터 센터에 속한 노드는 문제를 동시에 겪을 가능성이 있음

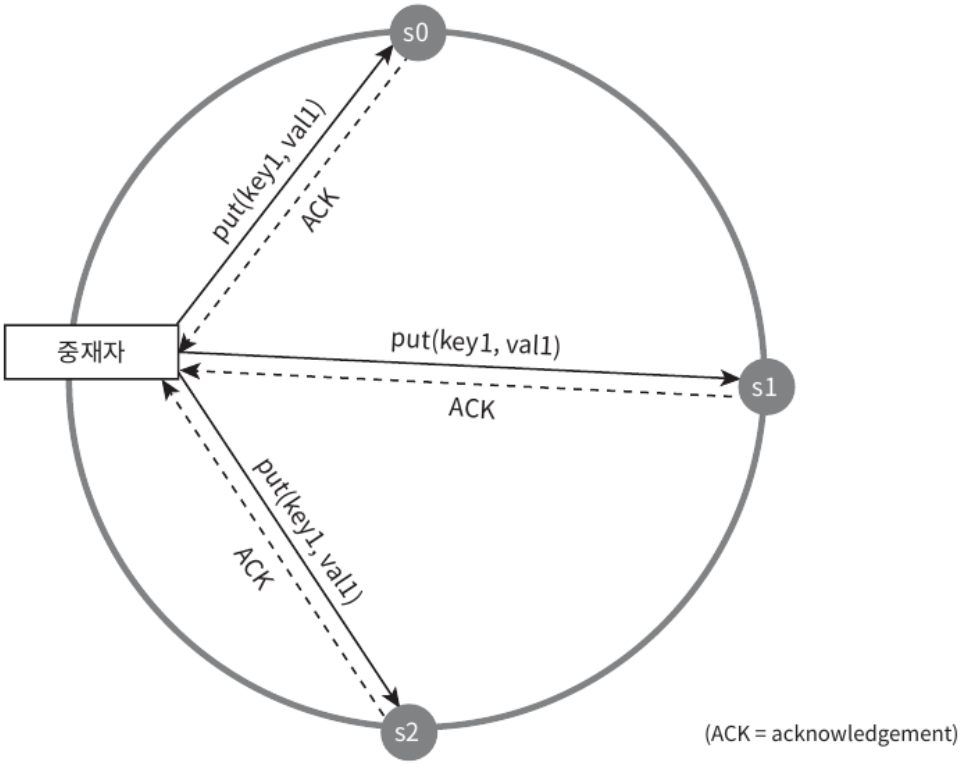
안정성을 담보하기 위해 데이터의 사본은 다른 센터의 서버에 보관하고, 센터들을 고속 네트워크로 연결하는 것이 좋음

▼ 데이터 일관성

여러 노드에 다중화된 데이터는 적절히 동기화 되어야 함

정족수 합의 프로토콜을 사용하면 읽기/쓰기 연산 모두에 일관성을 보장할 수 있음

```
N = 사본 개수
W = 쓰기 연산에 대한 정족수
R = 읽기 연산에 대한 정족수
```



W, R, N의 값을 정하는 것은 **응답 지연과 데이터 일관성 사이의 타협점을 찾는 과정**

요구되는 **일관성 수준**에 따라 W, R, N의 값을 조정하면 됨

- R = 1, W = N
  - 빠른 읽기 연산에 최적화된 시스템
- W = 1, R = N
  - 빠른 쓰기 연산에 최적화된 시스템
- W + R > N
  - 강한 일관석이 보장됨
  - 보통 N = 3, W = R = 2

- W + R ≤ N
    - 강한 일관성이 보장되지 않음

▼ 일관성 모델

데이터 일관성의 수준을 결정함

강한 일관성

모든 읽기 연산은 가장 최근에 갱신된 결과를 반환

클라이언트는 절대로 낡은 데이터를 보지 못함

강한 일관성 달성

- 모든 사본에 현재 쓰기 연산의 결과가 반영될 때까지 해당 데이터에 대한 읽기/쓰기 금지
- 고가용성 시스템에는 적합하지 않음
- 새로운 요청의 처리가 중단됨

약한 일관성

읽기 연산은 가장 최근에 갱신된 결과를 반환하지 못할 수 있음

최종 일관성

약한 일관성의 한 형태

갱신 결과가 결국에는 모든 사본에 반영(동기화)되는 모델

쓰기 연산이 병렬적으로 발생하면서 시스템에 저장된 값의 일관성이 깨질 수 있는데 이는 클라이언트가 해결해야 함 ⇒ 데이터 버저닝 기법 사용

▼ 비 일관성 해소 기법: 데이터 버저닝

데이터를 다중화하면 가용성은 높아지지만 사본 간 일관성이 깨질 가능성이 높아짐

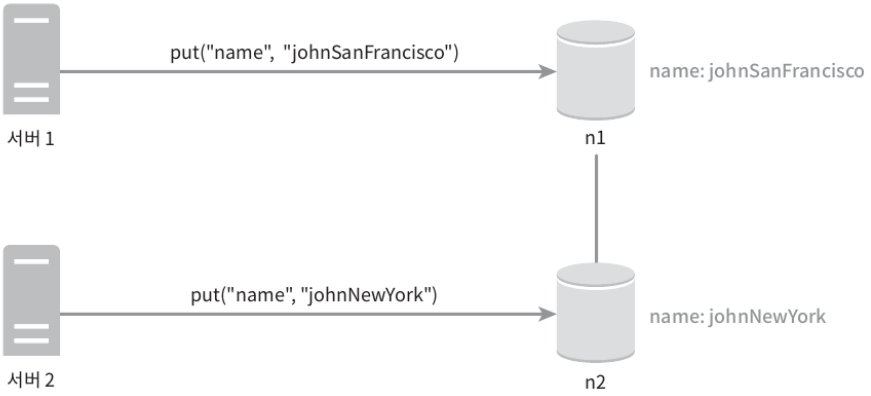
⇒ 이 문제를 해소하기 위해 **버저닝**과 **벡터 시계** 등장

버저닝

데이터를 변경할 때마다 해당 데이터의 새로운 버전을 만드는 것

각 버전의 데이터는 변경 불가능

데이터 일관성이 깨지는 과정



1. 서버 1은 name 값을 “johnSanfrancisco”로 변경
2. 서버 2는 name 값을 “johnNewYork”이라고 변경
3. 두 개의 연산은 동시에 이루어졌기 때문에 충돌하는 두 값을 가지게 됨
4. 충돌을 해소하기 어려워 보임

⇒ 충돌을 발견하고 자동으로 해결해 낼 **버저닝 시스템**이 필요

⇒ **벡터 시계**는 이러한 문제를 푸는 사용되는 보편적인 기술

▼ 벡터 시계

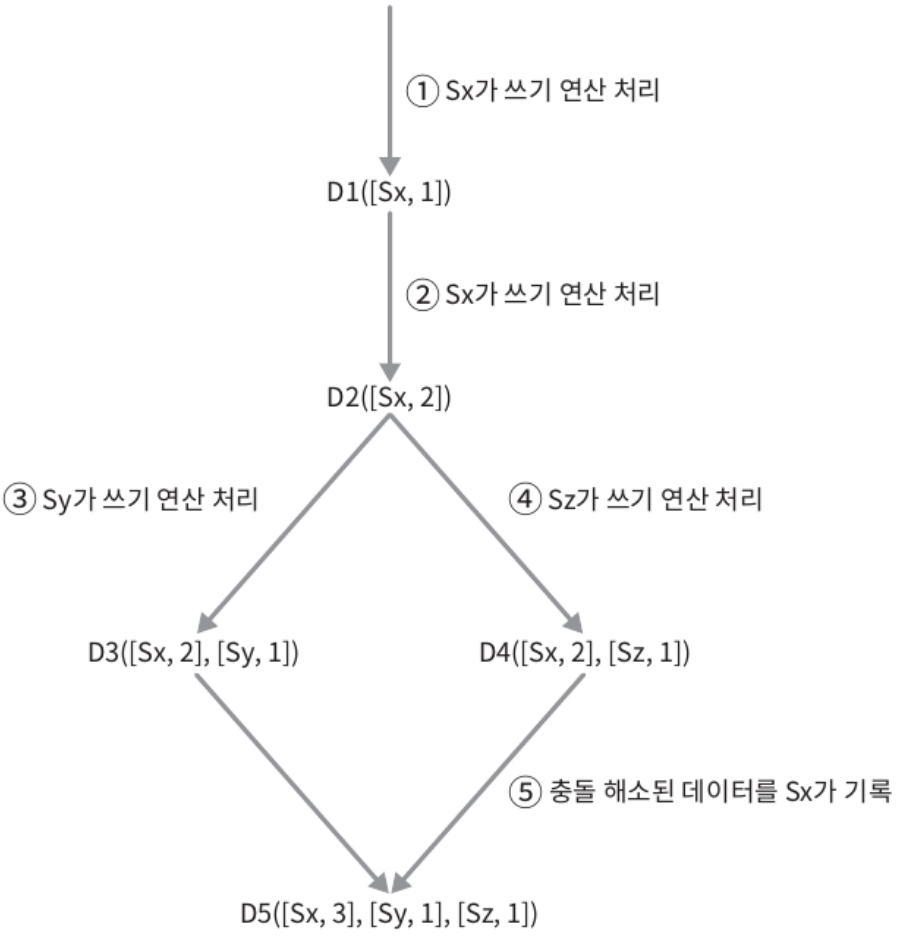
[서버, 버전]의 순서쌍을 데이터에 매단 것

어떤 버전이 선행 버전인지, 후행 버전인지, 다른 버전과 충돌이 있는지 판별할 때 사용

D([S1, v1], [S2, v2] ... [Sn, vn])와 같이 표현한다고 가정

- D = 데이터
- vi = 카운터
- si = 서버 번호

- 만일 데이터 D를 서버 Si에 기록할 경우 시스템은 아래 작업 가운데 하나를 수행해야 함
  - [Si, vi]가 있다면 vi를 증가
  - 그렇지 않으면 새 항목[Si, 1]를 만듦



1. 클라이언트가 데이터 D1을 시스템에 기록
2. 이 쓰기 연산을 처리한 서버는 Sx이다. 따라서 벤터 시계는 D1[(Sx, 1)]으로 변경
3. 다른 클라이언트가 데이터 D1을 읽고 D2로 업데이트 한 다음 기록
4. D2는 D1에 대한 변경이므로 D1을 덮어씀
5. 이 때 쓰기 연산은 같은 서버 Sx가 처리한다고 가정

6. 벡터 시계는 D2([Sx, 2])로 변경
7. 다른 클라이언트가 D2를 읽어 D3로 갱신한 다음 기록
8. 이 쓰기 연산은 Sy가 처리한다고 가정
9. 벡터 시계 상태는 D3([Sx, 2], [Sy, 1]])로 변경
10. 또 다른 클라이언트가 D2를 읽고 D4로 갱신한 다음 기록
11. 이 때 쓰기 연산은 서버 Sz가 처리한다고 가정
12. 벡터 시계는 D4([Sx, 2], [Sz, 1])로 변경
13. D2를 Sy와 Sz가 각기 다른 값으로 바꾸었기 때문에 어떤 클라이언트가 D3와 D4를 읽으면 데이터 간 충돌이 있다는 것을 알게 됨
14. 이 충돌은 클라이언트가 해소한 후에 서버에 기록
15. 이 쓰기 연산을 처리한 서버는 Sx였다고 가정
16. 벡터 시계는 D5([Sx, 3], [Sy, 1], [Sz, 1])로 변경

단점

- 충돌 감지 및 해소 로직이 클라이언트에 들어가야 하기 때문에 클라이언트 구현이 복잡해짐
- [서버:버전] 의 순서쌍 개수가 굉장히 빨리 늘어남
  - 임계치를 설정하고 임계치 이상으로 길이가 길어지면 오래된 순서쌍을 벡터 시계에서 제거하도록 해야 함
  - 버전 간 선후 관계가 정확하게 결정될 수 없기 때문에 충돌 해소 과정의 효율성이 낮아지게 됨
  - 하지만 실제 서비스에서 그런 문제가 벌어지는 것을 발견한 적이 없다고 하기 때문에 사용해도 될 듯

▼ 장애 처리

대규모 시스템에서 장애는 그저 불가피하기만 한 것이 아니라 아주 흔하게 벌어지는 사건

장애 처리는 굉장히 중요한 문제

▼ 장애 감지

분산 시스템에서는 보통 두 대 이상의 서버가 똑같이 특정 서버의 장애를 보고해야 해당 서버에 실제로 장애가 발생했다고 간주

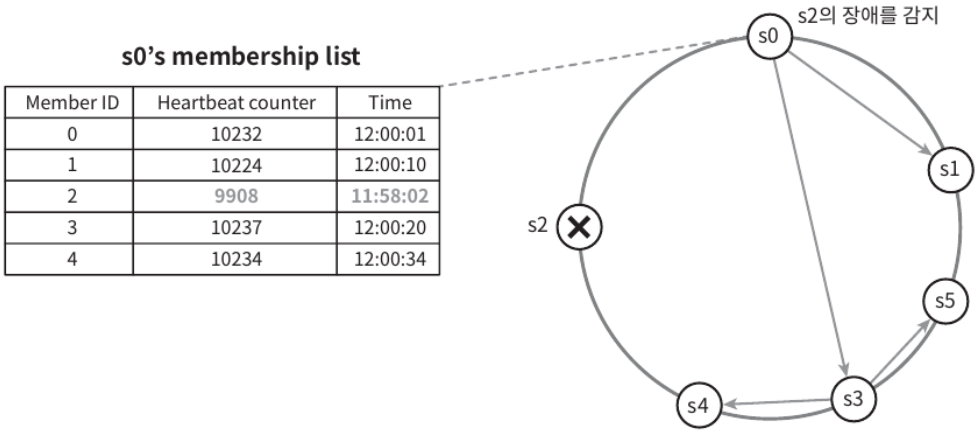
멀티캐스팅 채널

서버 장애를 감지하는 가장 손쉬운 방법

서버가 많을 때는 비효율적임

분산형 장애 감지 솔루션: 가십 프로토콜

서버가 많을 때는 분산형 장애 감지 솔루현을 채택하는 것이 효율적임



1. 각 노드는 멤버십 목록을 유지
2. 각 노드는 주기적으로 자신의 박동 카운터 증가
3. 각 노드는 무작위로 선정된 노드들에게 주기적으로 자기 박동 카운터 전송
4. 박동 카운터 목록을 받는 노드는 멤버십 목록을 최신 값으로 갱신
5. 어떤 멤버의 박동 카운터 값이 지정된 시간 동안 갱신되지 않으면 해당 멤버는 장애 상태인 것으로 간주

▼ 일시적 장애 처리

가십 프로토콜로 장애를 감지한 시스템은 가용성을 보장하기 위해 필요한 조치를 해야 함

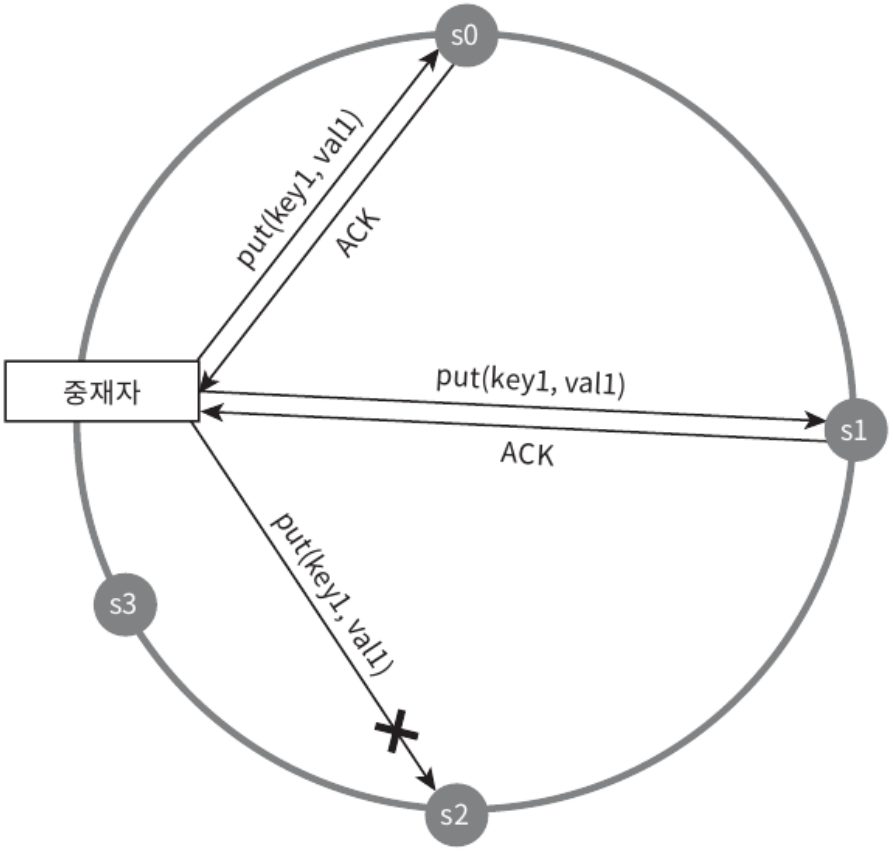
- 엄격한 정족수 접근법

    - 읽기/쓰기 연산 금지
- 느슨한 정족수 접근법
    - 엄격한 정족수 접근법의 조건을 완화하여 가용성을 높임
    - 정족수 요구사항을 강제
    - 쓰기 연산을 수행할 W개의 건강한 서버와 읽기 연산을 수행할 R개의 건강한 서버를 해시 링에서 선정
    - 장애 상태인 서버는 무시

장애 상태인 동안 발생한 변경사항은 해당 서버가 복구되었을 때 일관 반영

이를 위해 임시로 쓰기 연산을 처리한 서버에 그에 관한 단서를 남겨둠

⇒ 단서 후 임시 위탁



▼ 영구 장애 처리

영구적인 장애 처리를 위해 **반-엔트로피 프로토콜** 구현

반-엔트로피 프로토콜

사본들을 비교하여 최신 버전으로 갱신하는 과정을 포함

사본 간의 일관성이 망가진 상태를 탐지하고 전송 데이터의 양을 줄이기 위해 **머클 트리** 사용

머클 트리

각 노드에 그 자식 노드들에 보관된 값의 해시 또는 자식 노드들의 레이블로부터 계산된 해시 값을 레이블로 붙여두는 트리

대규모 자료 구조의 내용을 효과적이면서도 보안상 안전한 방법으로 검증 가능

머클 트리를 사용하면 동기화해야 하는 데이터의 양은 실제로 존재하는 차이의 크기에 비례할 뿐, 두 서버에 보관된 데이터의 총량과는 무관해 짐

하지만 실제로 쓰이는 시스템의 경우 버킷 하나의 크리가 꽤 크다는 것을 알아두어야 함

두 머클 트리 비교

1. 루트 노드의 해시 값을 비교하는 것으로 시작
2. 루트 노드의 해시 값이 일치하면 두 서버는 같은 데이터를 가지고 있는 것
3. 값이 다를 경우에는 왼쪽 자식 노드의 해시 값을 비교
4. 그 다음에는 오늘쪽 자식 노드의 해시 값을 비교
5. 이런 식으로 아래쪽으로 탐색해 나가다 보면 다른 데이터를 가지는 버킷을 찾을 수 있음
6. 다른 데이터를 가지는 버킷들만 동기화하면 됨

▼ 데이터 센터 장애 처리

데이터 센터 장애는 정전, 네트워크 장애, 자연재해 등 다양한 이유로 발생

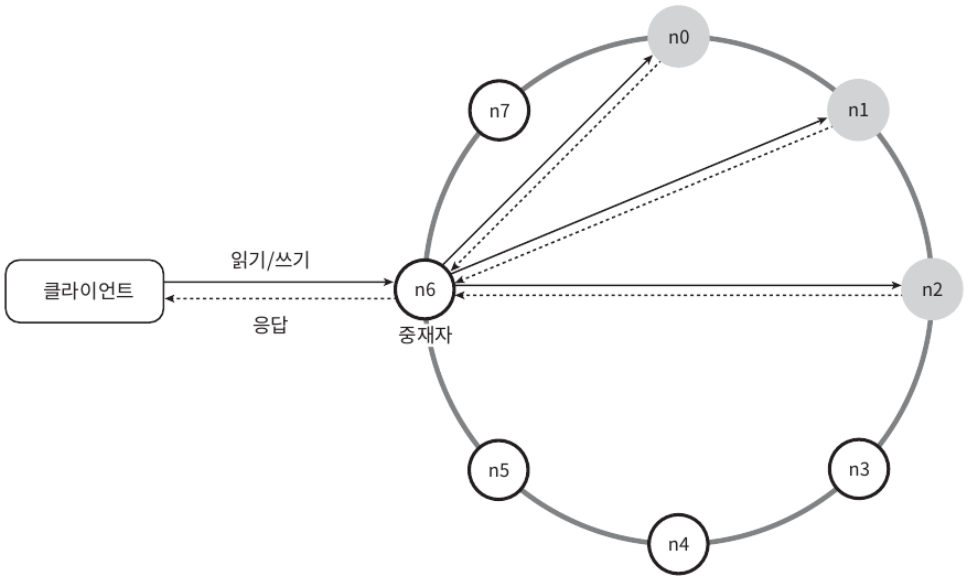
한 데이터 센터가 완전히 망가져도 사용하는 다른 다이터 센터에 보관된 데이터를 이용할 수 있도록 **데이터를 여러 데이터 센터에 다중화**하는 것이 중요

▼ 시스템 아키텍처 다이어그램

아키텍처의 주된 기능

- 클라이언트는 키-값 저장소가 제공하는 두 가지 단순한 API(get, put)와 통신함

- 중재자는 클라이언트에게 키-값 저장소에 대한 프록시 역할을 하는 노드
- 노드는 안정 해시의 해시 링 위에 분포

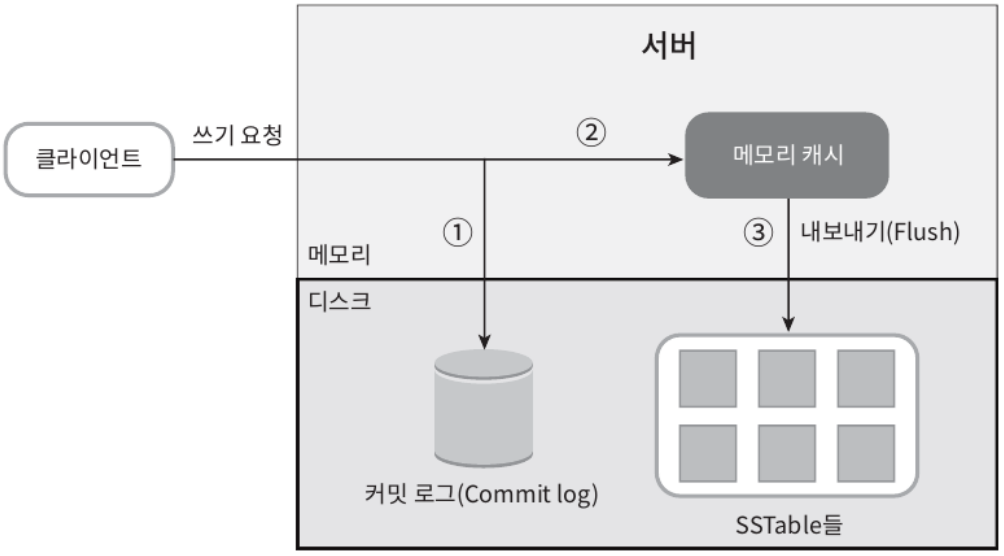


- 노드를 자동으로 추가, 삭제할 수 있도록 시스템은 완전히 분산됨
- 데이터는 여러 노드에 다중화
- 모든 노드가 같은 책임을 지기 때문에 SPOF는 존재하지 않음

완전히 분산된 설계를 채택했기 때문에 **모든 노드는 제시된 기능을 전부 지원해야 함**

- 클라이언트 API
- 데이터 충돌 해소
- 다중화
- 장애 감지
- 장애 복구 메커니즘
- 저장소 엔진

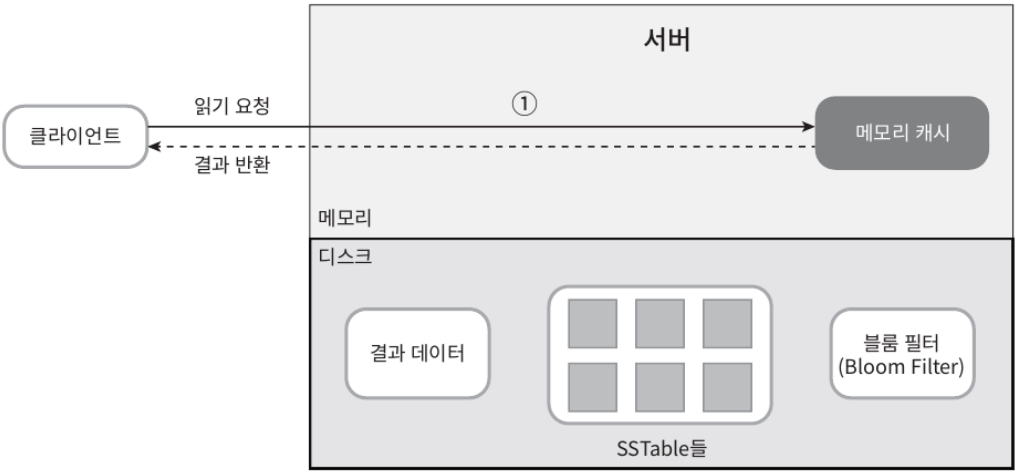
▼ 쓰기 경로



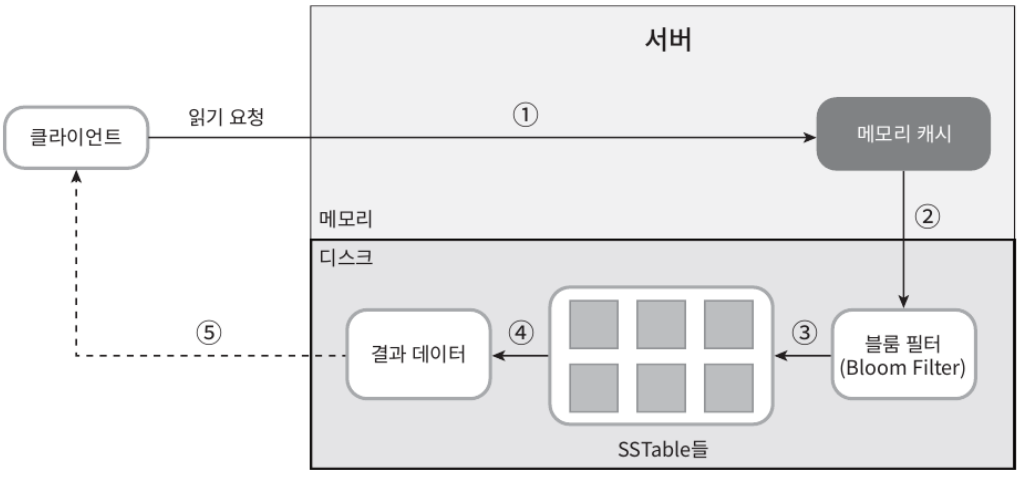
1. 쓰기 요청이 커밋 로그 파일에 기록
2. 데이터가 메모리 캐시에 기록
3. 메모리 캐시가 가득차거나 사전에 정의된 어떤 임계치에 도달하면 데이터는 디스크에 있는 SSTable(Sorted-String Table, 키-값의 순서쌍을 정렬된 리스트 형태로 관리하는 테이블)에 기록

▼ 읽기 경로

1. 데이터가 메모리에 있는지 검사



2. 있으면 메모리에서 데이터를 추출해서 클라이언트에게 반환



2. 데이터가 메모리에 없으면 블룸 필터를 검사
3. 블룸 필터를 통해 어떤 SSTable에 키가 보관되어 있는지 확인
4. SSTable에서 데이터 추출
5. 해당 데이터를 클라이언트에 반환

▼ 토론

분산 키-값 저장소는 위에서 언급된 방법들로 데이터의 일관성과 안전성을 보장하는 것 같은데 그럼 추가적인 트랜잭션 처리를 할 필요가 없는 건가?

- 그런 것 같음
- 일부 분산 키-값 저장소는 ACID (Atomicity, Consistency, Isolation, Durability) 트랜잭션을 완전히 지원하지만, 다른 저장소는 범위가 제한된 트랜잭션 또는 커밋의 일부분만 지원할 수도 있습니다.
- 따라서 분산 키-값 저장소를 사용할 때 트랜잭션 처리에 대한 특별한 주의가 필요하며, 저장소 시스템의 문서 및 지원 자료를 참고하여 해당 저장소의 트랜잭션 처리 기능을 이해하고 활용해야 합니다