6장 : 키-값 저장소 설계

난이도	⭐⭐⭐⭐
학습날짜	@2025년 12월 15일

키-값 저장소 (Key-Value Store)

키-값 데이터베이스라고도 불리며, 비 관계형(Non-relational) 데이터베이스이다.

- 저장되는 값은 고유한 식별자인 고유 식별자를 가져야 한다.
  - 해당 키를 통해서만 접근 가능해야 한다.
- 이런 키와 값 사이의 연결 관계를 키-값 쌍이라고 지칭한다.

> 다이나모, memcached, 레디스 등이 있다.

🗣️ **완벽한 설계**란 없다.
읽기, 쓰기, 그리고 메모리 사용량 사이에 어떤 **균형을 찾고**, 데이터의 일관성과 가용성 사이에서 **타협적 결정을 내린 설계를 만들었다면** 쓸만한 답안일 것이다.

🤔 비관계형 데이터베이스의 **등장 배경**과 **사용 이유**
1970년대부터 주류였던 관계형 데이터베이스(RDBMS)가 해결하지 **못한 지점들을 보완하며 발전**해왔다.

1. 등장 배경: RDBMS의 한계

과거에는 데이터가 정형화되어 있고 그 양이 예측 가능했지만, 2000년대 중반 웹 2.0 시대로 접어들며 상황이 변했습니다.

- **빅데이터의 출현:** 소셜 미디어, IoT 센서, 로그 파일 등 처리해야 할 **데이터의 양**이 기하급수적으로 늘어났습니다.
- **비정형 데이터의 증가:** 텍스트, 이미지, 비디오 등 고정된 표(Table) 형식에 담기 어려운 **비정형 데이터**가 많아졌습니다.
- **수평적 확장(Scale-out)의 한계:** RDBMS는 주로 서버의 성능을 높이는 '수직적 확장'에는 유리하지만, 여러 대의 저가형 서버로 분산 처리하는 '수평적 확장'은 복잡하고 비용이 많이 듭니다.
- **유연한 스키마 필요성:** 서비스 개발 속도가 빨라지면서 데이터 구조(Schema)를 미리 정의하고 엄격하게 지키는 것이 오히려 걸림돌이 되기도 했습니다.

2. NoSQL의 주요 사용 이유 (장점)

NoSQL은 "관계형 모델을 사용하지 않는다"는 특징 덕분에 다음과 같은 강력한 이점을 제공합니다.

① 유연한 데이터 모델 (Schema-less)

데이터 구조를 미리 정의할 필요가 없습니다. 같은 컬렉션 안에서도 데이터마다 다른 필드를 가질 수 있어, 서비스 요구사항이 자주 바뀌는 **스타트업이나 신규 프로젝트**에 유리합니다.

② 높은 확장성 (Scalability)

대부분의 NoSQL은 처음부터 분산 환경을 염두에 두고 설계되었습니다. 데이터 양이 늘어나면 서버를 단순히 추가하는 방식(Scale-out)으로 성능을 선형적으로 확장할 수 있습니다.

③ 고성능과 빠른 속도

복잡한 `JOIN` 연산을 배제하고 데이터가 물리적으로 가까운 곳에 저장되도록 설계되어, **단순 읽기/쓰기 작업에서 압도적인 속도**를 자랑합니다. 캐싱이나 실시간 분석에 자주 사용되는 이유입니다.

④ 가용성 (Availability)

데이터를 여러 서버에 복제하여 저장하기 때문에, 특정 서버에 장애가 발생해도 서비스가 중단되지 않고 계속 운영될 수 있는 높은 가용성을 제공합니다

3. 대표적인 NoSQL 유형

데이터를 어떻게 저장하느냐에 따라 크게 4가지로 나뉩니다.

유형	특징	대표 사례
Document	JSON과 유사한 문서 형태로 저장. 가장 범용적임.	MongoDB, CouchDB
Key-Value	키와 값의 쌍으로 저장. 속도가 가장 빠름.	Redis, Amazon DynamoDB
Wide-Column	행마다 다른 컬럼을 가질 수 있어 대용량 처리에 특화.	Cassandra, HBase
Graph	데이터 간의 관계를 노드와 간선으로 연결하여 표현.	Neo4j, JanusGraph

4. 요약: 언제 NoSQL을 써야 할까?

- 데이터의 구조가 확정되지 않았거나 자주 변경될 때
- 막대한 양의 데이터를 읽고 써야 하는 실시간 서비스 (SNS, 이커머스 장바구니 등)
- 서버를 여러 대 가동하여 시스템을 중단 없이 확장해야 할 때

NoSQL이 RDBMS를 완전히 대체하는 것은 아니다.
데이터의 **일관성과 무결성**이 절대적으로 중요한 금융 시스템 등에는 여전히 RDBMS가 사용됩니다. 따라서 현재는 두 방식을 적재적소에 혼합하여 사용하는 유연함이 필요하다.

CAP 정리

분산 키-값 저장소는 키-값 쌍을 여러 서버에 분산시키기 때문에 **분산 해시 테이블**이라고도 한다.

이때, **분산 시스템을 설계할 때는 CAP 정리**에 대해 숙지하고 있어야 한다.

Consistency (일관성)

정의: 모든 노드가 같은 시점에 같은 값을 반환

- 분산 시스템에 접속하는 모든 클라이언트는 **어떤 노드에 접속했느냐에 관계없이 언제나 같은 데이터**를 보게 되어야 한다.

Availability (가용성)

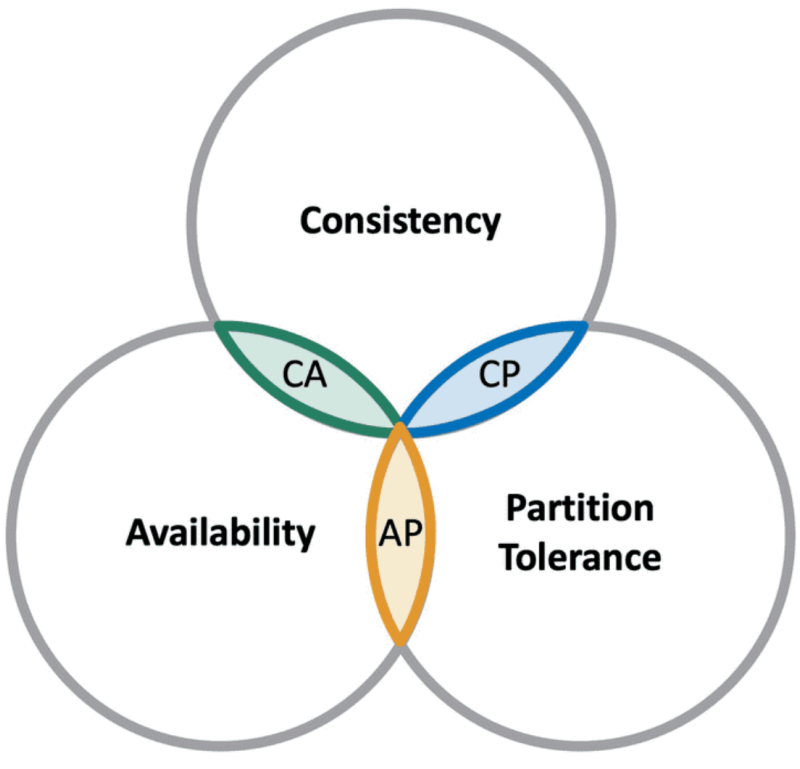
정의: 모든 요청은 (성공/실패 상관없이) 반드시 응답을 받는다

- 분산 시스템의 일부 노드에 **장애가 발생했더라도 항상 응답**을 받을 수 있어야 한다.

Partition Tolerance (파티션 감내)

정의: 네트워크 분할(Network Partition)이 발생해도 시스템이 동작해야 함

- 파티션 : 두 노드 사이에 통신 장애가 발생하였음을 의미



CAP 정리는 3가지 모두를 동시에 충족시키기에는 어렵고, 이들 중 2가지를 충족하려면 나머지 하나는 반드시 희생하여야 한다.

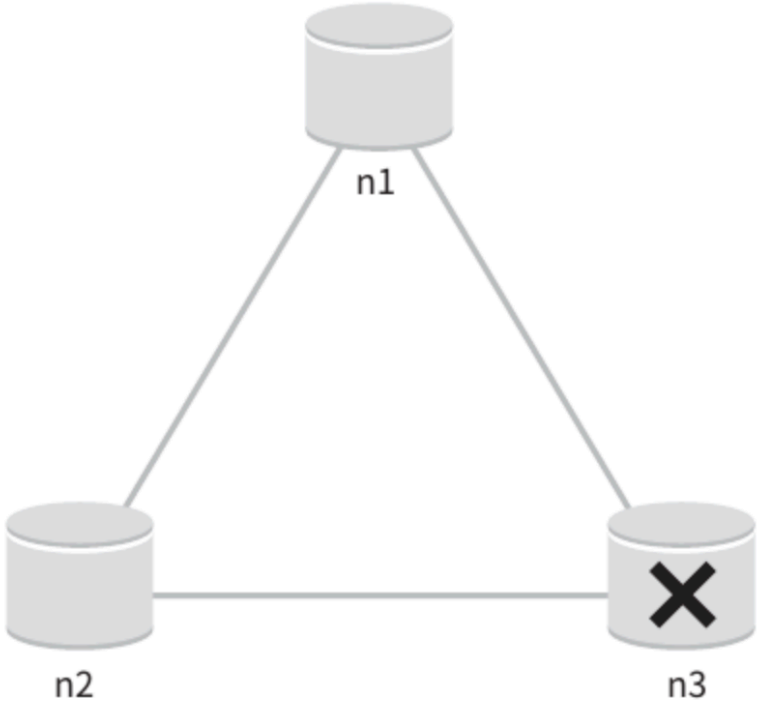
종류	정의
CP 시스템	일관성과 파티션 감내를 지원하는 키-값 저장소. 가용성을 희생한다.
AP 시스템	가용성과 파티션 감내를 지원하는 키-값 저장소. 데이터 일관성을 희생한다.
CA 시스템	일관성과 가용성을 지원하는 키-값 저장소. 이때, 통상 **네트워크 장애는 피할 수 없는 일**로 여겨지므로, 분산 시스템은 반드시 파티션 문제를 감내할 수 있도록 설계되어야 한다.

> 파티션 문제가 발생하면, 일관성과 가용성 사이에서 하나를 선택하여야 한다.

분산 시스템에서 가용성 vs 일관성에 대해 선택해야 하는 상황을 이해하기 위해 예시로 이해해보자.

아래는 분산 시스템에서 3개의 노드로 분할된 시스템을 나타낸다. 이때, n3가 나머지 두 노드와의 **파티션 문제**가 발생했다고 가정해보자.

- 파티션 문제 : 네트워크 장애
  - 파티션 문제는 노드간 네트워크 장애가 발생했다는 점으로, 구별해야 할 점은 **노드가 죽은 것은 아니**라는 점이다!



n3에 대한 파티션 문제가 발생

이때, 데이터의 쓰기와 읽기에 대한 일관성과 가용성에 대해 생각해보자.

1. **일관성 보장 (CP 시스템)**

   항상 동일한 데이터를 반환해야 하므로, 파티션 문제가 발생하면 쓰기 요청을 중단하는 방법으로 동일하지 않은 데이터가 저장되는 상황을 방지할 수 있다.

   → n1,n2 혹은 n3에 쓰기 작업을 막으면 동일한 데이터를 반환할 것이다.

2. **가용성 보장 (AP 시스템)**

   항상 데이터를 반환해야 하므로, 파티션 문제가 발생하였어도 각 노드에 GET 요청이 들어오면 반환해야 한다.

   → 이는, 파티션 문제가 발생한 이후 데이터에 쓰기 작업이 발생하여 서로 다른 값을 지니고 있어도 반환하게 된다는 것을 의미한다.

🤔 각 노드는 서로 다른 데이터를 저장하는 것으로 **안정 해시** 챕터에서 배웠는데, 각 노드별로 중복된 데이터를 가지고 있을 때가 문제인것으로 보인다. **중복된 값이 여러 노드에 걸쳐져 있는 이유는 무엇일까?**

- 만약 특정 Key를 가진 데이터가 오직 노드 A에만 있다면, 노드 A가 다운되는 순간 그 데이터는 서비스 불능 상태가 된다.
- 이를 막기 위해 안정 해시에서는 Key의 위치를 결정한 후, 링 위에서 시계 방향으로 만나는 **다음 N개의 노드에도 복제본(Replica)을 저장**한다.
  이렇게 하면 **노드 하나가 사라져도 복제본을 가진 다른 노드가 즉시 응답할 수 있다.**

📌 현업에서 사용하는 노드 복제 방법

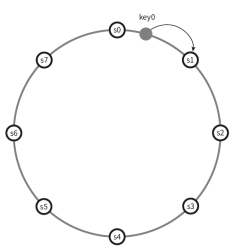
정리

🤔 공부 Flow

- 분산 시스템에 대한 이해
- 안정 해시 : 분산 시스템에서 부하를 적절하게 분리시키는 방법
- CAP 정리 : 분산 시스템에서 일관성 / 가용성을 설계하는 방법

시스템 컴포넌트 (키-값 저장소의 구현 방법)

1. 데이터 파티션



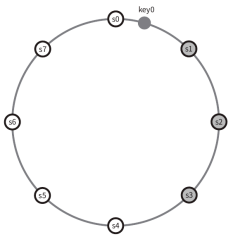
→ 데이터를 분리하여 저장하는 방법

- **안정 해시 및 해시 링, 가상 노드**

대규모 애플리케이션은 데이터가 많아지므로 모두 한 대 서버에 욱여넣는 것은 불가능하다.

따라서, 데이터를 작은 파티션들로 분할한 다음 여러 대 서버에 저장하는 방식을 사용할 수 있다.

2. 데이터 다중화

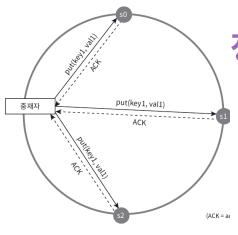


→ 하나의 데이터를 여러 군데에 중복하여 저장하는 방법

높은 가용성과 안정성을 확보하기 위해, 데이터를 N개 서버에 비동기적으로 다중화 (Replication)하여 저장할 수 있다.

- N : 각 서비스에 맞게 튜닝하는 값

3. 데이터 일관성



N = 3인 경우의 정족수 프로토콜

→ 여러 노드에 다중화된 데이터를 동기화하는 방법.

정족수 (Quorum Consensus) 프로토콜을 사용하여 읽기/쓰기 연산 모두에 일관성을 보장하도록 한다.

- N : 사본의 개수
- W : 쓰기 연산에 필요한 정족수. 쓰기 연산이 성공한 것으로 간주되려면 적어도 W개의 서버로부터 쓰기 연산이 성공했다는 응답을 받아야 한다.
- R : 읽기 연산에 필요한 정족수. 읽기 연산이 성공한 것으로 간주되려면 적어도 R개의 서버로부터 응답을 받아야 한다.

> N, W, R의 값을 정하여 **응답 지연과 데이터 일관성 사이의 타협점**을 찾아야 한다.

ex.

- R =1 , W = N : 빠른 읽기 연산에 최적화된 시스템
- W = 1, R = N : 빠른 쓰기 연산에 최적화된 시스템
- W + R > N : 강한 일관성
- W + R ≤ N : 강한 일관성이 보장되지 않음

3.1. 데이터 일관성 모델

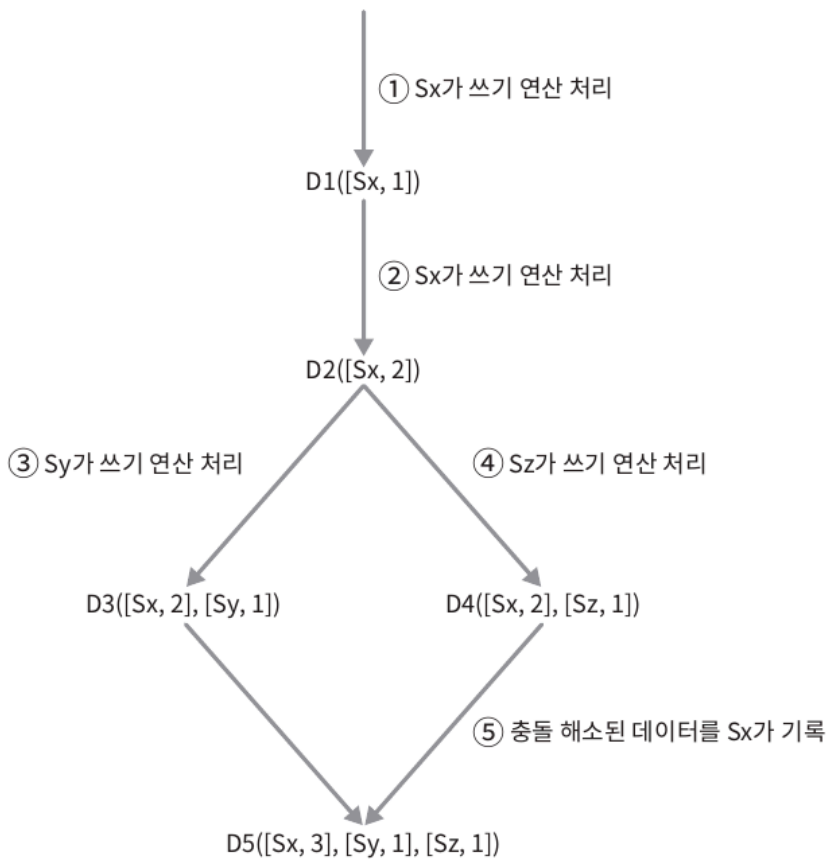
데이터의 일관성 수준을 결정하는 것.

종류	설명	구현 방법
강한 일관성 (Strong Consistency)	모든 읽기 연산은 **가장 최근에 결정된 결과**를 반환한다. 다시 말해서 클라이언트는 절대로 낡은 데이터를 보지 못한다	모든 사본에 현재 쓰기 연산의 결과가 반영될 때까지 해당 데이터에 대한 읽기/쓰기를 금지 - 시스템 처리가 중단되기 때문에, 고가용성 시스템에 적합하지 않음
약한 일관성 (Weak Consistency)	읽기 연산은 **가장 최근에 갱신된 결과를 반환하지 못할 수 있다.**	
결과적 일관성 (Eventual Consistency)	약한 일관성의 한 형태로, 갱신 결과가 결국에는 모든 사본에 반영(동기화)되는 모델	

3.2. 비 일관성 해소 기법 : 데이터 버저닝

버저닝과 벡터시계를 이용한 데이터 일관성을 위한 방법. 병렬성을 허용하였기 때문에 발생한 문제

- **버저닝** : 데이터를 변경할 때마다 해당 데이터의 새로운 버전을 만드는 것
- **벡터 시계** : **[서버, 버전]**의 순서쌍을 데이터에 매단 것
  - 선행 / 후행 버전을 판별하는데 쓰임



버저닝

📌 벡터 시계는 **충돌을 감지하기** 위한 방법으로, **충돌을 해결하지는 못한다.**
시스템이 **"이 데이터는 서로 모순되니 누군가 정리를 해야 해!"**라고 판단할 수 있는 **근거**를 제공하는 것까지가 벡터 시계의 역할

Q. 그럼 감지된 충돌을 누가 해결하나요?

충돌이 탐지된 후, 일관성을 다시 맞추는 작업(Resolution)은 보통 다음 두 가지 주체 중 하나가 담당합니다.
[클라이언트 해결 방법]

- 시스템(서버)은 충돌된 두 데이터(Sibling)를 모두 보관합니다.
- 사용자가 데이터를 읽으려고 할 때, 서버는 두 버전을 모두 던져주며 "충돌이 났으니 네가 합쳐서 다시 알려줘"라고 요청합니다.
    - 예: 쇼핑몰 장바구니 서비스에서 두 기기로 각각 품목을 담아 충돌이 났을 때, 두 장바구니의 품목을 합쳐서(Merge) 보여주는 방식.

▼ 구체적으로 클라이언트가 어떻게 해결하는지
1. 충돌 데이터(Sibling) 수신
사용자가 장바구니 페이지를 열면, 서버는 벡터 시계를 비교해보고 "아, 이 데이터는 충돌 상태구나"라고 판단합니다. 서버는 하나를 선택해 버리는 대신, **충돌된 데이터 덩어리들(Siblings)**을 모두 클라이언트에 보냅니다.

- **데이터 A (버전 [S1:1]):** `{"items": ["맥북"], "vclock": {"S1": 1}}`
- **데이터 B (버전 [S2:1]):** `{"items": ["아이폰"], "vclock": {"S2": 1}}`

서버는 이 두 벌을 리스트 형태 `[Data A, Data B]`로 묶어서 클라이언트에게 응답합니다.

2. 클라이언트의 병합(Merge) 로직 실행
클라이언트 앱(또는 브라우저) 코드에는 이 충돌을 어떻게 합칠지에 대한 **비즈니스 규칙**이 코딩되어 있습니다.

- **비즈니스 규칙 예시:** "장바구니 항목은 단순 합집합(Union)으로 처리한다."
- **연산 과정:**
    1. 데이터 A의 아이템 리스트 추출: `["맥북"]`
    2. 데이터 B의 아이템 리스트 추출: `["아이폰"]`
    3. 두 리스트 합치기: `["맥북", "아이폰"]`
    4. 새로운 벡터 시계 생성: 두 시계를 합친 후 자신의 카운터 업데이트 (예: `{"S1": 1, "S2": 1, "Client": 1}`)

3. 해결된 데이터의 재전송 (Read Repair)
클라이언트는 합쳐진 최종 데이터와 새로운 벡터 시계를 다시 서버로 보냅니다.

> "서버야, 아까 준 충돌 데이터들 내가 정리했어. 이게 최종본이고, 버전은 이거야!"

서버는 클라이언트가 보낸 이 '정리된 데이터'를 저장하고, 이전에 가지고 있던 충돌 데이터(Siblings)를 삭제합니다. 이 과정을 보통 **읽기 수리(Read Repair)**라고 부릅니다.

4. 왜 굳이 이렇게 복잡하게 할까? (핵심 이유)
서버는 데이터가 **'무엇'**인지 모르기 때문입니다.

- **만약 은행 잔고라면?** 단순히 합치는 게 아니라 숫자를 더하거나 빼야 합니다.
- **만약 설정 파일이라면?** 나중에 바뀐 설정을 우선시하거나, 사용자에게 "어떤 설정을 유지할까요?"라고 팝업을 띄워 물어봐야 합니다.
- **만약 삭제된 아이템이라면?** "삭제"라는 정보(Tombstone)가 있는 버전을 우선시해야 할 수도 있습니다.

이런 세세한 결정은 **사용자 경험(UX)과 직결**되기 때문에, 서버가 멋대로 판단하지 않고 클라이언트(애플리케이션 계층)에 위임하는 것입니다.

요약: 클라이언트 해결 프로세스

1. **Read:** 클라이언트가 데이터를 요청함.
2. **Conflict:** 서버가 충돌된 버전(Siblings) 목록을 보냄.
3. **Merge:** 클라이언트가 비즈니스 로직(합집합, 최신순 등)에 따라 데이터를 하나로 합침.

4. **Write:** 합쳐진 데이터를 서버에 다시 저장하여 충돌 상태를 해소함.

[서버 해결 방법]

- **LWW (Last Write Wins):** 가장 단순하게, 타임스탬프가 가장 최근인 것을 선택하고 나머지는 버립니다. (데이터 유실 위험이 있음)
- **CRDT (Conflict-free Replicated Data Type):** 데이터 구조 자체가 충돌이 발생하지 않도록 설계된 특수 타입을 사용하여 수학적으로 자동 병합합니다.

단점

1. **충돌 감지 및 해소 로직이 클라이언트**에 들어가야 하므로, 클라이언트 구현이 복잡함

🤔 클라이언트에 충돌 해소 로직이 들어가있는 이유
시스템(스토리지 계층)은:

- **의미(semantic)**를 모름
- 두 값 중 무엇이 "옳은지" 판단 불가

예를 들어:

```
A: "I love cats"
B: "I love dogs"
```

- 시스템은 병합 규칙을 모름
- "더 나은 문장" 같은 건 **도메인 지식**임

👉 그래서 **결정권을 클라이언트로 넘김**

🤔 FE 개발을 하면서 지금까지 충돌 해소 로직을 개발해보지 못했는데.. 경험 부족일까? 불필요한 작업일까?

1. **백엔드가 이미 강한 일관성을 강제함**
   지금까지의 백엔드 설계 자체가 강한 일관성을 강제하도록 설계되어서, FE에서 관리하지 않아도 되었음.
2. **FE가 아닌, 사용자에게 행동을 위임함**
   - 새로고침 / 재시도 유도
     - ex. "최신 데이터로 다시 시도해주세요"
   - 수정 중 경고
     - ex. "이 페이지를 오래 열어두셨습니다"
3. 일관성 문제가 발생하는 문제가 없었음.
   - 보통, 실시간 서비스를 사용하는 경우 일관성 문제가 발생함
     - ex. 노트, 메모 앱, 협업 툴, 구글 Docs 등

2. **[서버:버전]**의 순서쌍 개수가 굉장히 빨리 늘어난다

4. 장애 처리

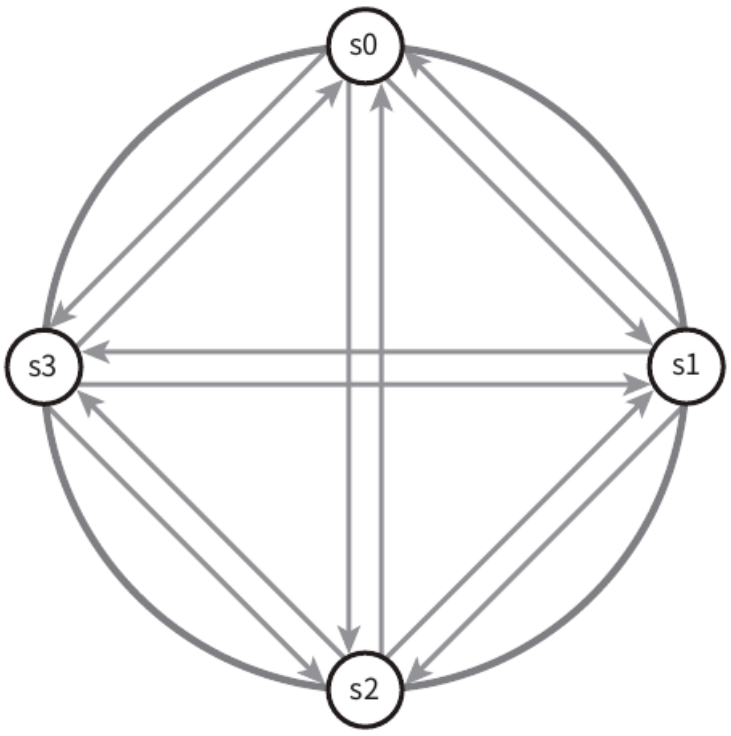
장애를 어떻게 처리할 것이냐는 굉장히 중요한 문제로, 장애 감지 (Failure Detection) 기법과 장애 해소 (Failure Resolution) 전략들을 짚어 보아야 한다.

4.1. 장애 감지

보통, **두 대 이상의 서버**가 똑같이 서버 A의 장애를 보고해야 해당 서버에 실제로 장애가 발생했다고 간주한다.

이를 위해, 보통 다음과 같은 멀티캐스팅 (multicasting) 채널을 구축하는 것이 가장 쉬운 방법이다.

- 서버가 많을 때는 커넥션이 너무 많아져서 비효율적 (n * (n-1) / 2 개)



multicasting

가십 프로토콜 (Gossip Protocol)

가십 프로토콜은 분산형 **장애 감지 솔루션**으로, 사람들이 소문을 퍼뜨리는 방식 (가십)과 유사하다고 하여 붙여진 이름으로, 전염병이 퍼지는 방식과도 비슷하여 에피데믹 프로토콜 (`Epidemic Protocol`)이라고도 불린다.

[작동 원리]

1. 각 노드는 멤버십 목록이 존재한다
    - 멤버십 목록 : 멤버 ID + 박동 카운터 쌍
2. **각 노드는 주기적으로 자신의 박동 카운터를 증가**시킨다
3. 각 노드는 **무작위**로 선정된 노드들에게 주기적으로 **자기 박동 카운터 목록을 보낸다**

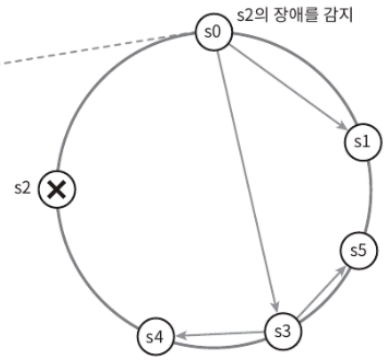
    > 네트워크에서의 Distance Vector 방식과 유사하다.

4. 박동 카운터 목록을 받은 노드는 멤버십 목록을 최신 값으로 갱신한다.
5. 🔥🔥 이때, 어떤 멤버의 박동 카운터 값이 **지정된 시간 동안 갱신되지 않으면 해당 멤버는 장애 (Offline) 상태**인 것으로 간주한다.

예시 설명

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



- 노드 s0은 그림 좌측의 테이블과 같은 멤버십 목록을 가진 상태이다.
- 노드 s0은 노드 s2(멤버ID=2)의 박동 카운터가 오랫동안 증가되지 않았다는 것을 발견한다.
- 노드 s0은 노드 s2를 포함하는 박동 카운터 목록을 무작위로 선택된 다른 노드에게 전달한다.
- 노드 s2의 박동 카운터가 오랫동안 증가되지 않았음을 발견한 모든 노드는 해당 노드를 장애 노드로 표시한다.

📌 다른 노드들한테 받는다. 일정 시간 다른 노드로부터 받지 않으면 장애가 났다고 감지 vs 박동 카운터를 받지 않은
→ 무작위 노드로 멤버십 리스트를 보내야 하기 때문에 필요하다.

4.2. 장애 해소

가십 프로토콜로 장애를 감지한 이후에는, 이에 걸맞는 장애 처리 방안을 마련해두어야 한다.

일시적 장애 해소

가십 프로토콜로 장애를 감지한 시스템은 가용성을 보장하기 위한 조치를 해야 한다.

엄격한 정족수 접근법을 쓴다면 읽기와 쓰기 연산을 금지하는 등...

느슨한 정족수 접근법은 이 조건을 완화하여 가용성을 높인다.

정족수 요구사항을 강제하는 대신, 쓰기 연산을 수행할 W개의 건강한 서버와 읽기 연산을 수행할 R개의 건강한 서버를 해시 링에서 고른다. 이 때 장애 상태인 서버는 무시한다.

장애인 서버로 가는 요청은 다른 서버에서 처리한다.

그동안의 변경사항은 해당 서버 복구 시 일괄로 반영하여 일관성을 높인다.

이를 위해 임시로 쓰기 연산을 처리한 서버에서 hint를 남긴다. 이를 단서 후 임시 위탁 (`hinted handoff`)이라 한다.

영구 장애 해소

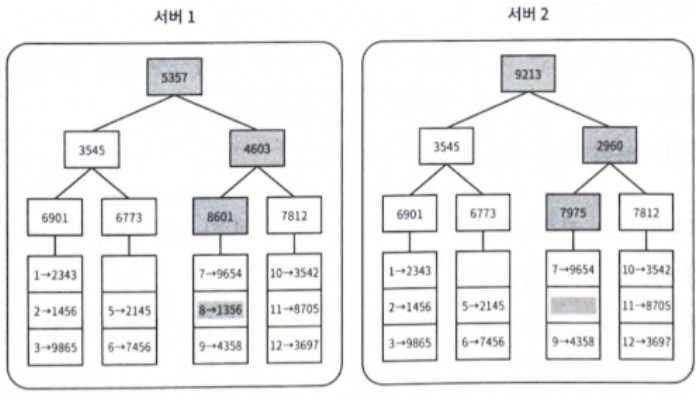
[반 - 엔트로피 프로토콜]

반-엔트로피 프로토콜은 **사본들을 비교하여 최신 버전으로 갱신하는 과정**을 포함한다.

사본 간의 일관성이 망가진 상태를 탐지하고 전송 데이터의 양을 줄이기 위해서는 **머클(Merkle)트리**를 사용할 것이다.

- 머클 트리 : 해시 트리라고도 불리며 각 노드에 그 자식 노드들에 보관된 값의 해시, 또는 자식 노드들의 레이블로부터 계산된 해시 값을 레이블로 붙여두는 트리이다.

해당 머클 트리를 사용하면 동기화해야 하는 데이터의 양은 실제로 존재하는 차이의 크기에 비례할 뿐, 두 서버에 보관된 데이터의 총량과는 무관해진다. 하지만 실제로 쓰이는 시스템의 경우 버킷 하나의 크기가 꽤 크다는 것은 알아두어야 한다.



4개의 버킷으로 나눠진 예제

구현 방법

1. ‘키 공간을 여러 개의 버킷으로 나눈다
2. 버킷에 포함된 각각의 키에 균등 분포 해시 (Unform Hash) 함수를 적용하여 해시 값을 계산한다.
3. 버킷별로 해시값을 계산한 후, 해당 해시 값을 레이블로 갖는 노드를 만든다
4. 자식 노드의 레이블로부터 새로운 해시 값을 계산하여, 이진 트리를 상향식으로 구성한다.

장애 탐지 및 동기화:

루트 노드에서 아래쪽으로 탐색해 나가다 보면 다른 데이터를 갖는 버킷을 찾을 수 있다.
해당 버킷들만 동기화하면 된다.

📌 반-엔트로피 프로토콜은 **버킷을 동기화시키는 방식**으로 일관성을 보장한다. 즉, 동기화해야할 버킷을 찾는 방법이다.
Q. 왜 데이터를 해시하는가?

- 데이터의 크기를 줄이기 위해
  - 원본 1TB >> 사본 1B

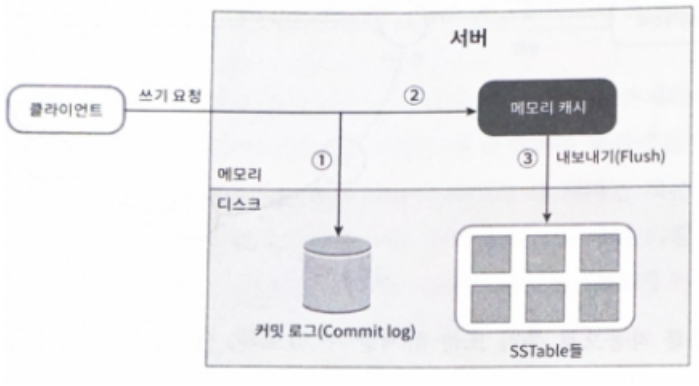
데이터 센터 장애 처리

데이터 센터 또한 다양한 원인(정전, 네트워크 장애, 자연재해 등)으로 인해 장애가 발생할 수 있다.

따라서, 데이터 센터 또한 다중화하는 것이 중요하다.

분산 시스템의 읽기 / 쓰기 구현 방법

쓰기 경로



1. 쓰기 요청이 **커밋 로그 (Commit Log) 파일에 기록**된다.
2. 데이터가 메모리 캐시에 기록된다
3. 메모리 캐시가 가득 차거나 사전에 등록된 임계치에 도달하면 데이터는 디스크에 있는 SSTable에 기록된다.
    - SSTable은 `Sorted-String Table`의 약어로, <키, 값>의 순서쌍을 정렬된 리스트의 형태로 관리하는 테이블

📌 SSTable 이란?
분산 시스템(특히 Cassandra, Bigtable 등)의 읽기/쓰기 시스템에서 **SSTable(Sorted Strings Table)**은 디스크에 저장되는 **데이터의 최종 보관 형태**를 말합니다.
쉽게 비유하자면, SSTable은 한 번 쓰면 수정할 수 없는, 가나다순으로 정렬된 단어장과 같습니다.

1. 핵심 특징

- **Sorted (정렬됨):** 모든 데이터(Key-Value)가 키를 기준으로 정렬되어 있습니다. 덕분에 특정 데이터를 찾을 때 '이진 탐색'처럼 아주 빠르게 찾을 수 있고, 범위 검색(Range Scan)도 매우 효율적입니다.
- **Immutable (불변성): 가장 중요한 특징입니다.** 한 번 디스크에 쓰인 SSTable은 절대 수정되지 않습니다. 데이터를 수정하거나 삭제하고 싶다면 기존 파일을 고치는 게 아니라, 새로운 SSTable 파일에 "이 값은 이제 A가 아니라 B야" 혹은 "이 값은 삭제됐어(Tombstone)"라는 내용을 새로 씁니다.
- **Persistence (지속성):** 메모리(MemTable)에 있던 데이터가 가득 차면 디스크로 옮겨져 영구 저장된 상태입니다.

2. 왜 이렇게 복잡하게 저장하나요? (LSM 트리 방식)

SSTable은 주로 **LSM(Log-Structured Merge) 트리** 구조에서 사용됩니다. 일반적인 DB(B-Tree 방식)처럼 파일 중간을 찾아가서 수정하는 방식은 디스크 I/O 비용이 매우 큽니다. 반면 SSTable 방식은 다음과 같은 흐름을 가집니다.

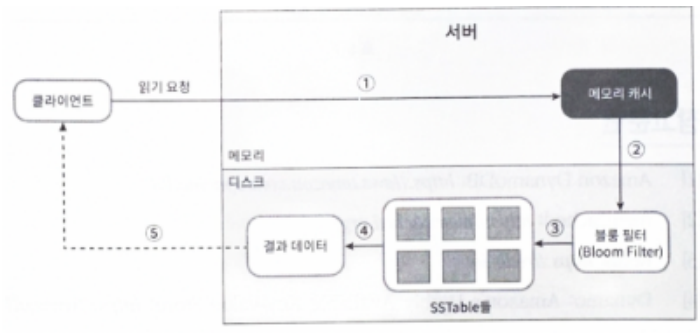
1. **쓰기:** 메모리(**MemTable**)에 먼저 정렬해서 저장합니다. (매우 빠름)
2. **플러시(Flush):** 메모리가 꽉 차면 통째로 디스크에 던집니다. 이게 하나의 **SSTable**이 됩니다.
3. **읽기:** 메모리에 없으면 디스크에 있는 여러 개의 SSTable을 뒤집니다.
4. **병합(Compaction):** SSTable이 너무 많아지면 백그라운드에서 하나로 합치면서, 중복된 값이나 삭제된 데이터를 정리합니다.

3. 읽기 시스템에서 SSTable을 찾는 과정

데이터를 읽을 때, 시스템은 여러 개의 SSTable 중 어디에 내가 찾는 값이 있는지 알아내야 합니다. 이를 위해 다음과 같은 보조 도구들을 함께 사용합니다.

- **Bloom Filter (블룸 필터):** "이 SSTable에 네가 찾는 키가 확실히 없니?"라고 물어보는 장치입니다. 없다는 대답은 100% 확실해서, 불필요한 디스크 읽기를 획기적으로 줄여줍니다.
- **Index File:** 각 SSTable마다 키가 어디 위치에 있는지 알려주는 목차 파일입니다.
- **Tombstone (비석):** 데이터를 삭제했다는 표시입니다. 읽기 시스템은 여러 SSTable을 뒤지다가 이 '비석'을 만나면 해당 데이터가 삭제되었음을 인지합니다.

읽기 경로



1. 데이터가 메모리 캐시에 있는 경우에는 데이터를 반환한다.
2. 데이터가 메모리 캐시에 없는 경우에는 *어느 SSTable에 찾는 키가 있는지* 알아내어야 한다. 이 문제를 풀기 위해 불룸 필터 (Bloom Filter)가 흔히 사용된다.
    - 데이터가 메모리에 있는지 검사한다.
    - 데이터가 메모리에 없으므로 불룸 필터를 검사한다.
    - 불룸 필터를 통해 어떤 SSTable에 키가 보관되어 있는지 알아낸다
    - SSTable에서 데이터를 가져온다
    - 해당 데이터를 클라이언트에게 반환한다.

📌 **불룸 필터?**

> **없다고 하면 진짜 없고, 있으면 있을 수도 있다.**

특정 원소가 집합에 **포함되어 있는지 여부를 확인**하기 위해 사용하는 아주 가볍고 효율적인 **확률적 자료구조.**
→ 분산 시스템이나 DB에서 SSTable을 뒤지기 전에, "네가 찾는 데이터가 이 파일 안에 혹시라도 있을까?"를 먼저 물어보는 **'문지기'** 역할

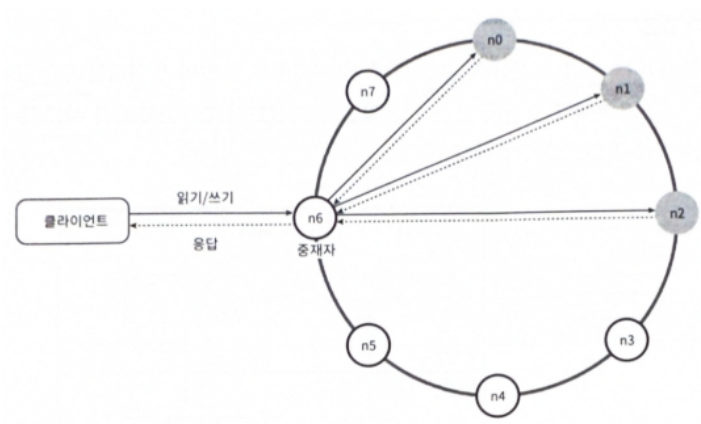
작동 원리: 비트 배열과 해시 함수

블룸 필터는 아주 작은 메모리 공간(비트 배열)만 사용합니다.

1. **초기 상태:** 000000... 처럼 모든 칸이 0인 비트 배열을 만듭니다.
2. **데이터 저장:** 데이터(예: "UserA")를 여러 개의 해시 함수에 통과시켜 나온 숫자 자리에 **1**을 표시합니다.
3. **데이터 확인:** "UserB"가 있는지 물어보면, 똑같이 해시 함수를 돌려봅니다.
    - 만약 해당 자리 중 **하나라도 0**이면? → "UserB는 절대 없음!"
    - 모든 자리가 **1**이면? → "UserB가 있을 가능성이 높음!" (다른 데이터들 때문에 우연히 1이 채워졌을 수도 있음)

즉, 블룸 필터는 "데이터가 여기 없으니까 헛수고하지 마!"라고 알려주는 데 특화되어 있습니다.

최종 시스템 아키텍처 다이어그램



1. 클라잉너트는 키-값 저장소가 제공하는 두 가지 단순한 API, get(key) / put(key,value)와 통신한다.
2. 중재자는 클라이언트에게 키-값 저장소에 대한 프락시 역할을 하는 노드이다
3. 노드는 안정 해시의 해시 링 위에 분포한다.
4. 노드는 자동으로 추가 또는 삭제될 수 있도록 완전히 분산되어야 한다
5. 데이터는 여러 노드에 다중화된다
6. 모든 노드가 같은 책임을 지므로, SPOF가 존재하지 않는다.

시스템(스토리지 계층)은:

- *의미(semantic)**를 모름
- 두 값 중 무엇이 “옳은지” 판단 불가

예를 들어:

```
A: "I love cats"
B: "I love dogs"
```

- 시스템은 병합 규칙을 모름
- “더 나은 문장” 같은 건 **도메인 지식**임

👉 그래서 **결정권을 클라이언트로 넘김**