6장 키-값 저장소 설계

- 키-값 저장소
  - 비 관계형 데이터베이스
  - 저장되는 값은 고유 식별자(identifier)를 키로 가져야 한다.
  - 키-값 쌍에서 키는 유일해야 한다
  - 해당 키에 매달린 값은 키를 통해서만 접근할 수 있다

문제 이해 및 설계 범위 확정

- 키-값 쌍의 크기는 10KB 이하
- 큰 데이터 저장 가능
- 높은 가용성 제공
- 높은 규모 확장성 제공
- 데이터 일관성 수준 조정 가능
- 응답 지연시간(latency) 짧아야한다.

단일 서버 키-값 저장소

- 한 대의 서버만 사용 → 키-값 쌍 전부를 메모리에 해시 테이블로 저장
- 빠른 속도, BUT 메모리 부족 → 데이터 압축 or 일부는 디스크에 저장 → 그래도 부족해...

분산 키-값 저장소 (분산 해시 테이블)

CAP 정리

- **데이터 일관성(Consistency)**, **가용성(Availability)**, **파티션 감내(Partition Tolerance)** 라는 세 가지 요구사항을 **동시에 만족하는** 분산 시스템을 설계하는 것은 **불가능하다**
- **데이터 일관성** : 분산 시스템에 접속하는 모든 클라이언트는 어떤 노드에 접속했느냐에 관계 없이 **언제나 같은 데이터**를 보게 되어야 한다.
- **가용성** : 분산 시스템에 접속하는 클라이언트는 일부 노드에 장애가 발생하더라도 **항상 응답을 받을 수 있어야** 한다.
- **파티션 감내** : 네트워크에 **파티션(두 노드 사이에 통신 장애가 발생하였음)이 생기더라도** 시스템은 계속 동작하여야 한다.

> 💡 네트워크 장애는 피할 수 없는 일로 여겨지므로, 분산 시스템은 반드시 파티션 문제를 감내할 수 있도록 설계되어야 한다. 즉, 실제로 CA 시스템은 존재하지 않는다.

시스템 컴포넌트

📍 데이터 파티션

- 대규모 애플케이션에서 전체 데이터를 한 서버에 저장하기 어렵기 때문
- 고려사항
  - 데이터를 여러 서버에 고르게 분산할 수 있는가
  - 노드가 추가되거나 삭제될 때 데이터의 이동을 최소화할 수 있는가
- **안정해시** (Recap. 5장)
  - 규모 확장 자동화 : 시스템 부하에 따라 서버가 자동으로 추가/삭제
  - 다양성 : 각 서버의 용량에 맞게 가상 노드 수 조절 가능

📌 데이터 다중화

- 높은 가용성과 안정성 확보를 위해 데이터를 N개 서버에 **비동기적으로 다중화 (replication)** 필요
- N개 서버를 선정하는 방법
  - 어떤 키를 해시 링 위에 배치
  - 그 지점으로부터 시계 방향으로 링을 순회하면서 만나는 첫 N개 서버에 데이터 사본을 보관

  (노드를 선택할 때 같은 물리 서버를 중복 선택하지 않아야 한다)

📌 데이터 일관성

- 여러 노드에 **다중화된** 데이터는 **동기화** 필요
- **Quorum Consensus 프로토콜** → 읽기/쓰기 연산 모두에 일관성 보장
  - 중재자 : 클라이언트와 노드 사이에서 proxy 역할
  - N = 사본 개수
  - W = 쓰기 연산이 성공한 것으로 간주되려면 적어도 W개의 서버로부터 쓰기 연산이 성공했다는 응답을 받아야 한다.
  - R = 읽기 연산이 성공한 것으로 간주되려면 적어도 R개의 서버로부터 응답을 받아야 한다.
  - W, R, N 값을 정하는 방법 → **응답 지연과 데이터 일관성 사이의 타협점**
    - R = 1, W = N : 빠른 읽기 연산에 최적화된 시스템
    - W = 1, R = N : 빠른 쓰기 연산에 최적화된 시스템
    - W + R > N : 강한 일관성이 보장
    - W + R ≤ N : 강한 일관성이 보장되지 않음

- 일관성 모델
  - **강한 일관성** : 모든 읽기 연산은 가장 최근에 갱신된 결과를 반환 (→ 클라이언트는 절대 낡은 데이터 보지 못함)
  - **약한 일관성** : 읽기 연산은 가장 최근에 갱신된 결과를 반환하지 못할 수 있다.
  - **결과적 일관성** : 약한 일관성의 한 형태로, 갱신 결과가 결국에는 모든 사본에 반영(동기화)되는 모델

- 비 일관성 해소 기법 : 데이터 버저닝(versioning)
  - **버저닝** : 데이터를 변경할 때마다 해당 데이터의 새로운 버전을 만드는 것 (→ 각 버전의 데이터는 변경 불가능)
  - **벡터 시계** : [서버, 버전]의 순서쌍을 데이터에 매단 것. 어떤 버전이 선행/후행인지, 다른 버전과 충돌 여부 판단에 사용
    - 데이터 `D`, 버전 카운터 `vi`, 서버 번호 `Si`일 때 벡터 시계는 `D([S1, v1], [S2, v2], … , [Sn, vn])` 같이 표현
    - 데이터 D를 서버 Si에 기록하면, 시스템은 [Si, vi]가 있으면 vi를 증가하고, 없다면 새 항목 [Si, 1]를 만든다

- 장애 감지 (failure detection)
  1. 모든 노드 사이에 멀티캐스팅(multicasting) 채널을 구축 → 쉽지만 비효율적
  2. 가십 프로토콜(gossip protocol) 같은 분산형 장애 감지 솔루션 채택
     a. 각 노드는 멤버십 목록(각 멤버 ID와 heartbeat 카운터 쌍의 목록)을 유지
     b. 각 노드는 주기적으로 자신의 heartbeat 카운터 증가
     c. 각 노드는 무작위로 선정된 노드들에게 주기적으로 자기 heartbeat 카운터 목록을 보낸다.
     d. heartbeat 카운터 목록을 받은 노드는 멤버십 목록을 최신 값으로 갱신
     e. 어떤 멤버의 heartbeat 카운터 값이 지정된 시간 동안 갱신되지 않으면 해당 멤버는 장애(offline) 상태인 것으로 간주

- 일시정 장애 처리
  - strict quorum → 읽기와 쓰기 연산 금지
  - sloppy quorum → quorum 요구사항을 강제하는 대신, 쓰기 연산을 수행할 W개의 건강한 서버와 읽기 연산을 수행할 R개의 건강한 서버를 해시 링에서 고른다.
  - **단서 후 임시 위탁(hinted handoff)**
    - 네트워크나 서버 문제로 장애 문제인 서버로 가는 요청은 다른 서버가 잠시 맡아 처리
    - 그동안 발생한 변경사항은 해당 서버가 복구되었을 때 일괄 반영하여 데이터 일관성 보존

      → 이를 위해 임시로 쓰기 연산을 처리한 서버에 그에 관한 단서(hint) 남겨둠

- 영구 장애 처리
  - 반-엔트로피(anti-entropy) 프로토콜 구현 → 사본 동기화
    - 사본들을 비교하여 최신 버전으로 갱신
    - 사본 간의 일관성이 망가진 상태를 탐지하고 전송 데이터의 양을 줄이기 위해 **머클(Merkle) 트리** 사용
  - 머클 트리(해시 트리)
    - 각 노드에 그 자식 노드들에 보관된 값의 해시, 또는 자식 노드들의 레이블로부터 계산된 해시값을 레이블로 붙여두는 트리
    - 머클 트리를 만드는 방법
      1. 키 공간을 버킷으로 나눈다.
      2. 버킷에 포함된 각각의 키에 균등 분포 해시 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 구성해 나간다

시스템 아키텍처 다이어그램

- **클라이언트는** 키-값 저장소가 제공하는 두 가지 단순한 API(`get(key)`, `put(key, value)`))와 통신한다.
- **중재자(coordinator)**는 클라이언트에게 키-값 저장소에 대한 **프락시(proxy)** 역할을 하는 노드다
- 노드는 **안정 해시의 해시 링** 위에 분포한다.
- 노드를 자동으로 추가 또는 삭제할 수 있도록, 시스템은 **완전히 분산**된다

  → 각 노드는 클라이언트API, 장애 감지, 데이터 중돌 해소, 다중화, 장애 복구 메커니즘, 저장소 엔진 기능 지원!
- 데이터는 여러 노드에 **다중화**된다.
- 모든 노드가 같은 책임을 지므로, SPOF는 존재하지 않는다.

쓰기 경로

1. 쓰기 요청이 커밋 로그(commit log) 파일에 기록된다.
2. 데이터가 메모리 캐시에 기록된다.
3. 메모리 캐시가 가득차거나 사전에 정의된 어떤 임계지에 도달하면 데이터는 디스크에 있는 SSTable에 기록된다.

읽기 경로

1. 데이터가 메모리에 있는지 검사한다. 없으면 2로 간다.
2. 데이터가 메모리에 없으므로 블룸 필터를 검사한다.
3. 블룸 필터를 통해 어떤 SSTable에 키가 보관되어 있는지 알아낸다.
4. SSTable에서 데이터를 가져온다.
5. 해당 데이터를 클라이언트에게 반환한다.

7장. 분산 시스템을 위한 유일 ID 생성기 설계

1단계 : 문제 이해 및 설계 범위 확정

- ID는 유일해야 한다.
- ID는 숫자로만 구성되어야 한다.
- ID는 64비트로 표현될 수 있는 값이어야 한다.
- ID는 발급 날짜에 따라 정렬 가능해야 한다.
- 초당 10,000개의 ID를 만들 수 있어야 한다.

2단계 : 개략적 설계안 제시 및 동의 구하기

다중 마스터 복제

- **다음 ID의 값을 구할 때 k(= 현재 사용중인 데이터 베이스 서버의 수)만큼 증가시킨다.**
- 규모 확장성 문제를 어느 정도 해결할 수 있지만, 여러 단점 존재
  - 여러 데이터 센터에 걸쳐 규모를 늘리기 어렵다
  - ID의 유일성은 보장되지만, 시간 흐름에 맞춰 커지도록 보장할 수 없다
  - 서버를 추가/삭제할 때 잘 동작하도록 만들기 어렵다

UUID(Universally Unique Identifier)

- 컴퓨터 시스템에 저장되는 정보를 유일하게 식별하기 위한 128비트짜리 수
- 장점 - 만드는 것이 단순, 서버 사이 조율이 필요 없으므로 동기화 이슈 없음, 규모 확장 쉬움
- 단점 - 요구사항보다 길고, 시간순 정렬 불가, 숫자가 아닌 값이 포함될 수 있음

티켓 서버

- auto_increment 기능을 갖춘 데이터베이스 서버, 즉 티켓 서버를 중앙 집중형으로 하나만 사용
- 장점 - 유일성이 보장되는 숫자로만 구성된 ID를 쉽게 만듬, 구현이 쉽고 중소 규모에 적합

- 단점 - 티켓 서버가 SPOF가 된다.

트위터 스노플레이크(twitter snowflake) 접근법

- **사인 비트** : 1비트 할당. 음수와 양수 구별에 사용
- **타임 스탬프** : 41비트 할당. 기원 시각(epoch) 이후로 몇 밀리초가 경과했는지 나타내는 값
- **데이터 센터 ID** : 5비트 할당
- **서버 ID** : 5비트 할당
- **일련변호** : 12비트 할당. 각 서버에서 ID를 생성할 때마다 이 일련번호를 1만큼 증가. 1밀리초가 경과할 때마다 0으로 초기화

3단계 : 상세 설계

타임스탬프

- 시간이 흐름에 따라 점점 큰 값 → ID를 시간순으로 정렬 가능하도록

일련번호

- 12비트 → 4096개의 값. 어떤 서버가 같은 밀리초 동안 하나 이상의 ID를 만들어 낸 경우에만 0보다 큰 값

4단계 : 마무리

- **시계 동기화** : 하나의 서버가 여러 코어에서 실행될 경우나 서버가 물리적으로 독립도니 여러 장비에서 실행될 경우, ID 생성 서버들이 전부 같은 시계를 사용하지 않을 수 있다.
- **각 절(section) 길이 최적화**
- **고가용성**