CP5. 복제

Part 2. 분산 데이터

▼ 개요

- Part 1 에서는 단일 장비에서 데이터를 저장할 때 적용하는 데이터 시스템 측면을 다뤘다.
  Part 2 에서는 저장소와 데이터 검색에 여러 장비가 관여하면 무슨일이 발생할까? 주제 중심으로 다룬다.
- 여러 장비 간 분산된 DB를 필요하는 이유는 여러가지다.
    - 확장성
      데이터 볼륨, 읽기 부하, 쓰기 부하가 단일 장비에서 다룰 수 있는 양보다 커지면 부하를 여러 장비로 분배할 수있다.
    - 내결함성/고가용성
      장비 하나(또는 여러 장비나 네트워크, 전체 데이터센터)가 죽더라도 애플리케이션이 계속 동작해야 한다면 여러 장비를 사용해 중복성을 제공할 수 있다. 장비 하나가 실패하면 다른 하나가 이어 받는다. (fail over)
    - 지연시간
      전 세계 사용자가 있다면 사용자와 지리적으로 가까운 곳의 데이터센터에서 서비스를 제공하기 위해 전 세계 다양한 곳에 서버를 두고 싶을 것이다. 이를 통해 사용자는 네트워크 패킷이 지구를 반 바퀴 돌아서 올 때까지 기다릴 필요 없다.
- **고부하로 확장**
    - **공유 메모리 아키텍쳐**
        - 고부하 확장이 필요하다면 더 강한 장비를 구매하는게 가장 단순하다(수직 확장, 용량 확장) 많은 CPU, 메모리, 디스크를 하나의 운영체제로 함께 결합할 수 있다. 그래서 빠른 상호 연결로 모든 CPU 가 메모리나 디스크의 모든 부분에 접근 할 수 있다.
        - 공유 메모리 아키텍처에는 모든 구성 요소를 단일 장비처럼 다룰 수 있다.
        - 문제점은 비용이 선형적인 추세보다 훨씬 빠르게 증가한다.
          시스템 성능이 두 배를 내기 위해서는 비용이 두 배 이상이 소요된다.
          또한 병목 현상 때문에 두 배 크기의 장비가 반드시 두 배의 부하를 처리할 수 있는 것은 아니다.
        - 공유 메모리 아키텍처는 제한적인 내결함성을 제공한다. (장비를 중단 시키지 않고 스케일 업 할 수 있다) 하지만 완전히 하나의 지리적 위치로 제한된다.

- **공유 디스크 아키텍쳐**
  - 공유 메모리 아키텍쳐와는 다른 접근 방식이다. 독립적인 CPU와 RAM 을 탑재한 여러 장비를 사용하지만 데이터 저장은 장비 간 공유하는 디스크 배열을 한다.
  - 여러 장비는 고속 네트워크로 연결된다. 일부 데이터 웨어하우스 작업부하에 이 아키텍쳐를 사용하지만 잠금 경합과 오버헤드가 공유 디스크 접근 방식의 확장성을 제한한다.
- **비공유 아키텍쳐 (수평 확장, 규모 확장, 스케일 아웃)**
  - DB 소프트웨어를 수행하는 각 장비나 가상 장비를 **노드**라고 부른다.
    각 노드는 CPU, RAM, 디스크를 독립적으로 사용한다.
    노드 간 코디네이션은 일반적인 네트워크를 사용해 소프트웨어 수준에서 수행한다.
  - 비공유 시스템은 특별한 하드웨어를 필요하지 않아 가격 대비 성능이 가장 좋은 시스템을 사용할 수 있다. 잠재적으로 지리적인 영역에 걸쳐 데이터를 분산해 사용자 지연 시간을 줄이고 전체 데이터센터의 손실을 줄일 수 있다.
  - Part 2 에서는 비공유 아키텍쳐에 중점을 둔다.
    비공유 아키텍쳐를 사용시 애플리케이션 개발자가 반드시 주의해야 하는 점이 있기 때문,
    데이터를 여러 노드에 분산하려면 분산 시스템에서 발생하는 제약 조건과 트레이드오프를 알고 있어야 한다. DB 스스로 이런 점을 숨길 수 없다
  - 대개 장점이 많지만, 애플리케이션 복잡도를 야기하고 때로는 데이터 모델의 표현을 제한한다. 경우에 따라 간단한 단일 스레드 프로그램이 100개 이상의 CPU 코어를 사용하는 클러스터 보다 효율적일 수 있다. 하지만 비공유 시스템은 매우 강력하다.
- **복제 대 파티셔닝**
  - 여러 노드에 데이터를 분산하는 방법은 일반적으로 두 개다.
    - 복제
      - 같은 데이터 복사본을 잠재적으로 다른 위치에 있는 여러 노드에 유지한다.
      - 복제는 중복성을 제공한다. 일부 노드가 사용 불가능한 상태라면 해당 데이터는 남은 다른 노드를 통해 여전히 제공될 수 있다. 복제는 성능 향상에도 도움된다.
    - 파티셔닝
      - 큰 DB를 파티션이라는 작은 서브셋으로 나누고 파티션은 각기 다른 노드에 할당한다. (**샤딩**)
  - 복제와 파티셔닝은 다른 매커니즘이지만, 서로 관련있다.
    - 파티셔닝과 복제를 같이 사용해 분산 시스템에서 필요한 어려운 트레이드오프(트랜잭션, ACID)를 설명할 수 있다.
    - 트랜잭션을 이해하면 데이터 시스템에 발생하는 많은 문제를 설명하는 데 도움을 준다.
    - 이후 장에서 복잡한 애플리케이션의 요구사항을 만족하기 위해 어떻게 다양한 (분산된) 데이터 저장소를 가져와 대규모 시스템을 통할할 수 있는지 설명한다.

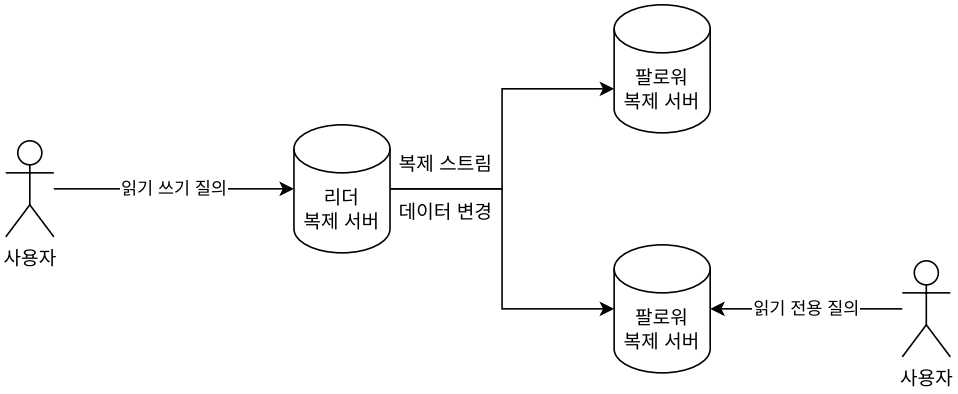
5. 복제
▼ 개요

- 복제
  - 복제란 네트워크로 연결된 여러 장비에 동일한 데이터의 복사본을 유지한다는 의미.

- 복제가 필요한 이유
  - 지리적으로 사용자와 가깝게 데이터를 유지해 지연 시간 감소
  - 시스템 일부 장애 발생하더라도 지속적 동작 가능 (HA)
  - 읽기 질의를 제공하는 장비의 수를 확장해 읽기 처리량 증가
- 복제 중인 데이터가 시간이 지나도 변경되지 않는다면 복제는 쉽다. 한번에 모든 노드에 데이터를 복사하며 된다. 복제의 어려움은 복제된 데이터의 변경 처리이다.
- 노드 간 변경을 복제하기 위한 세가지 복제 알고리즘
  - 단일 리더 (single-leader)
  - 다중 리더 (multi-leader)
  - 리더 없는 (learderless)
- 복제시 고려해야 할 많은 트레이드오프가 존재함.
  - 동기식 복제, 비동기식 복제
  - 잘못된 복제본의 처리
- 분산 DB 에 대한 내용
  - 최종적 일관성
  - 쓰기 읽기 보장
  - 단조 읽기 보장

▼ 리더와 팔로워

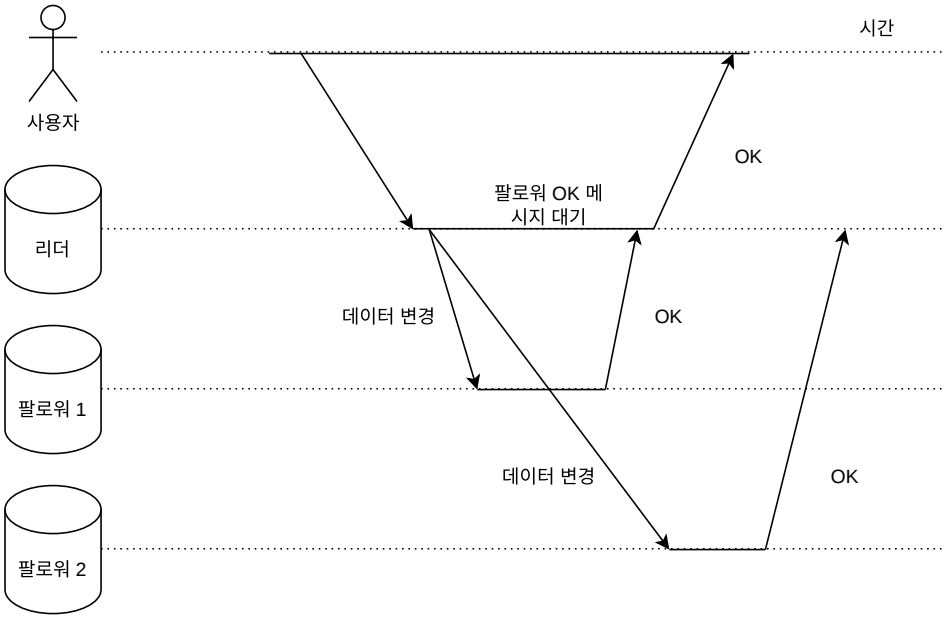
- **복제 서버(replica)** (DB 복사본)
  - 모든 복제 서버에 모든 데이터가 있다는 사실을 어떻게 보장할까?
    - DB의 모든 쓰기는 모든 복제 서버에서 처리되어야 한다. (그렇지 않으면 복제 서버는 더 이상 동일한 데이터를 유지할 수 없음)
    - 일반적인 해결책은 **리더 기반 복제 (leader-based replication) (능동/수동 복제) (마스터 슬레이브 복제)**
- **리더 기반 복제 (leader-based replication) (능동/수동 복제) (마스터 슬레이브 복제)**



  - 복제 서버 중 하나를 **리더(leader)(master, primary)** 로 지정.
  - 다른 복제 서버 **팔로워 (follower)(읽기 복제 서버)(슬레이브, secondary, hot standby)**

- 쓰기 (클라이언트의 쓰기는 반드시 리더만 허용된다.)
  - 클라이언트가 DB에 쓰기를 할 때 클라이언트는 요청을 리더에게 보내야 한다.
  - 리더는 먼저 로컬 저장소에 새로운 데이터를 기록한다.
  - 리더가 로컬 저장소에 새로운 데이터를 기록할 때마다 데이터 변경을 **복제 로그**, **변경 스트림**의 일부로 팔로워에게 전송한다.
  - 각 팔로워가 리더로부터 로그를 받으면 리더가 처리한 것과 동일한 순서로 모든 쓰기를 적용해 그에 맞게 DB 의 로컬 복사본을 갱신한다.
- 읽기
  - 클라이언트가 DB로 읽기를 할 때 리더 or 임의 팔로워에게 질의 가능하다.

- **동기식 대 비동기식 복제**



  - 동기식 (팔로워 1의 복제)
    - 리더는 팔로워 1이 쓰기를 수신했는지 확인해 줄 때까지 대기한다.
      확인이 끝나면 사용자에게 성공 응답을 해주고 다른 클라이언트에게 해당 쓰기를 보여준다.
    - 장점
      - 리더와 일관성 있게 최신 데이터 복사본을 가지는 것을 보장
      - 리더가 동작하지 않아도 데이터는 팔로워에게 계속 사용할 수 있음을 보장
    - 단점
      - 동기 팔로워가 응답하지 않으면 쓰기 처리 불가능
      - 리더는 모든 쓰기를 차단하고 동기 복제 서버가 다시 사용할 수 있을 때까지 대기됨.
      - 위 이유로 모든 팔로워가 동기식 방식은 비현실적이다.
        그래서
        반동기식을 사용한다. (하나의 팔로워는 동기식 나머지는 비동기식)
  - 비동기식 (팔로워 2의 복제)

        - 리더는 메시지를 전송하지만 팔로워의 응답을 기다리지 않음.
    - DB 복제 시간은 보장할 수 없다.
        - 장애를 극복 중
        - 시스템 최대 가용량 근처에서 동작
        - 노드 간 네트워크 문제 등등..
- **새로운 팔로워 설정**
    - 복제 서버 수를 늘리거나, 장애 노드 대체를 위함
      새로운 팔로워가 리더의 데이터 복제본을 정확히 가지고 있는지 어떻게 보장함?
        - 한 노드에서 다른 노드로 데이터 파일을 복사하는 것만으로는 대개 충분하지 않다.
        - 클라이언트는 지속적으로 DB에 기록하고 데이터는 항상 유동적이기 때문에 표준 파일 복사본은 다른 시점에 DB의 다른 부분을 보게 된다. 즉, 복사 결과가 유효하지 않을 수 있다.
        - DB를 lock 해서 디스크 파일을 일관성 있게 만들 수 있지만, 고가용성 목표에 부합하지 못한다. 다행히 팔로워 설정은 대개 중단시간 없이 수행할 수 있다.
    - 새로운 팔로워 추가 과정
        1. 가능하다면 전체 DB를 잠그지 않고 리더 DB의 스냅숏을 일정 시점에 가져온다. 대부분의 DB는 백업이 필요하기 때문에 이 기능이 있다.
        2. 스냅숏을 새로운 팔로워 노드에 복사한다.
        3. 팔로워는 리더에 연결해 스냅숏 이후 발생한 모든 데이터 변경을 요청한다.이것은 스냅숏이 리더의 복제 로그의 정확한 위치와 연관돼야 한다.
        4. 팔로워가 스냅숏 이후 데이터 변경의 미처리분을 모두 처리했을 때 따라잡았다고 한다.
- **노드 중단 처리**
    - 시스템의 모든 노드는 장애로 인해 중단될 수 있지만 계획된 유지보수로 인해 중단될 수도 있다.
    - 중단시간 없이 개별 노드를 재부팅할 수 있다는 점은 운영과 유지보수에 큰 장점이다.
    - 따라서 개별 노드의 장애에도 전체 시스템이 동작하게끔 유도하고 노드 중단의 영향을 최소화하는 것이 목표다.
    - **팔로워 장애: 따라잡기 복구**
        - 각 팔로워는 리더로부터 수신한 데이터 변경 로그를 로컬 디스크에 보관한다.
        - 팔로워가 죽어 재시작하거나 리더와 팔로워 사이의 네트워크가 일시적으로 중단된다면 팔로워는 매우 쉽게 복구할 수 있다.
        1. 먼저 보관된 로그에서 결함이 발생하기 전에 처리한 마지막 트랜잭션을 알아낸다.
        2. 그러면 팔로워는 리더에 연결해 팔로워 연결이 끊어진 동안 발생한 데이터 변경을 모두 요청할 수 있다.
        3. 이 변경이 다 적용되면 리더를 따라잡게 되고 이전과 같이 데이터 변경의 스트림을 계속 받을 수 있다.
    - **리더 장애: 장애 복구(failover)**
        - 리더의 장애를 처리하는 것은 까다롭다.
          팔로워 중 하나를 새로운 리더로 승격해야 하고 클라이언트는 새로운 리더로 쓰기를 전송하기 위해 재설정이 필요하며 다른 팔로워는 새로운 리더로 부터 데이터 변경을 소비하기 시작해야 한다. 이

과정을
장애 복구(failover) 라 한다.

- 장애 복구는 수동 이나 자동으로 진행된다.
- 자동 장애 복구 과정
  1. 리더가 장애인지 판단
     - 고장, 정전, 네트워크 문제 등 잠재적으로 여러 가지가 문제 일 수 있다.
     - 무엇이 잘못됐는지 발견할 수 있는 확실한 방법이 없기 때문에 대부분의 시스템은 단순히 타임아웃을 사용한다.
     - 노드들은 자주 서로 메시지를 주고 받으며 일정 시간 동안 노드를 응답하지 않으면(타임아웃) 죽은 것으로 간주한다.
  2. 새로운 리더 선택
     - 산출 과정(리더가 나머지 복제 서버의 대다수에 의해 선택) 통해 되거나 이전에 선출된 제어 노드에 의해 새로운 리더가 임명될 수 있다.
     - 가장 적합한 후보는 보통 이전 리더의 최신 데이터 변경사항을 가진 복제 서버다
  3. 새로운 리더 사용을 위한 시스템 재설정
     - 클라이언트는 이제 새로운 쓰기 요청을 새로운 리더에게 보내야 한다
     - 이전 리더가 돌아오면 여전히 자신이 리더라 믿을 수 있어야 하고 다른 복제 서버들이 자신을 리더에서 물러나게 한 것을 알지 못한다.
     - 시스템은 이전 리더가 팔로워가 되고 새로운 리더를 인식할 수 있게끔 해야한다.
- 복구 과정에서 잘못될 수 있는 과정 (이 문제들에 대한 쉬운 해결책은 없다)
  - 비동기식 복제를 사용한다면 새로운 리더는 이전 리더가 실패하기 전에 이전 쓰기 일부를 수신하지 못할 수 있다. 새로운 리더가 선출된 다음 이전 리더가 클러스터에 다시 추가된다면 이 쓰기를 어떻게 해야 할까? 그 동안 새로운 리더가 충돌하는 쓰기를 수신했을지도 모른다. 가장 일반적인 해결책은 이전 리더의 복제되지 않은 쓰기를 단순히 폐기하는 방법이다. (내구성을 기대할 수 없다)
    - 쓰기를 폐기하는 방법은 DB 외부의 다른 저장소 시스템이 DB 내용에 맞춰 조정돼야 한다면 특히 위험하다.
      ex)
      깃허브 유요하지 않은 마이 SQL 팔로워 승격된 사례
  - **스플릿 브레인(split brain)**
    - 특정 결함 시나리오에서 두 노드가 모두 자신이 리더라고 믿을 수 있다.
    - 매우 위험한 상황이고 두 리더가 쓰기를 받으면서 충돌을 해소하는 과정을 거치지 않으면 데이터가 유실되거나 오염된다. 일부 시스템에서는 안전 장치로 두 리더가 감지되면 한 노드를 종료하는 매커니즘이 있다. (잘 못하면 두 리더 모두 종료 될 수도 있다.)
  - 리더가 분명히 죽었다고 판단 가능한 적절한 타임아웃은 얼마일까?
    - 긴 타임아웃은 리더가 작동하지 않을 때부터 복구까지 오랜 시간이 소요된다는 뜻이다.
    - 하지만 타임아웃이 너무 짧으면 불필요한 장애 복구가 있을 수 있다.
      - ex) 일시적인 부하 급증으로 노드 응답 시간이 타임아웃보다 커지거나 네트워으 고장으로 패킷이 지연되는 경우

        - 노드 장애, 불안정한 네트워크, 복제 서버 일관성과 관련된 트레이드오프, 지속성, 가용성, 지연 시간 등의 문제는 사실 분산 시스템에서 발생하는 근본적인 문제다.

- **복제 로그 구현 (리더 기반 복제 내부적 동작)**

    1. **구문 기반 복제**

        - **리더는 모든 쓰기 요청(구문)을 기록하고 쓰기를 실행한다.**
          다음 구문 로그를 팔로워에게 전송한다.

          (관계형 DB는 모든 DML 구문을 팔로워에게 전달하고 각 팔로워는 클라이언트에게 직접 받은 것처럼 SQL 구문을 파싱하고 실행한다)
        - 복제가 깨질 수 있는 사례
            - 현재 날짜와 시간, 임의 숫자를 얻는 서버마다 다른 값을 생성할 수 있는 함수로 다른 값을 생성 할 수 있다. (now, rand)
            - 자동증가 컬럼을 사용하는 구문이나 DB에 있는 데이터에 의존한다면, 구문은 각 복제 서버에서 정확히 같은 순서로 실행되어야 한다. 이 방식은 동시에 여러 트랜잭션을 수행되는 것을 막는다
            - 부수 효과를 가진 구문 (트리거, SP, function) 은 부수 효과가 완벽하게 결정적이지 않으면 각 복제 서버에서 다른 부수 효과가 발생할 수 있다.
        - 해결책이 있긴하다. (ex) 리더는 구문을 기록할 때 모든 비결정적 함수 호출을 고정 값을 반환하게 할 수 있다.)
          하지만 여러 에러 케이스가 있어서 지금은 다른 복제 방법을 선호 한다.

    2. **쓰기 전 로그 배송**

        - 3 장 저장소 엔진이 디스크 상에서 데이터를 표현하고 모든 쓰기는 로그에 기록됨
            1. 3 장에서 나온 로그 구조화 저장소 엔진 (SS 테이블, LSM 트리) - 로그 세그먼트 작게 유지 백그라운드 GC,
            2. 개별 디스크 블록에 덮어쓰는 B 트리 (B 트리) - 모든 변경은 쓰기 전 로그 (WAL)에 쓰기 때문에 고장 이후 일관성 있는 상태로 색인 복원 가능
        - 위 두 경우 **모든 로그는 DB의 모든 쓰기를 포함하는 추가 전용(append-only) 바이트 열이다. 완전히 동일한 로그를 사용해 다른 노드에서 복제 서버 구축이 가능하다.**
          그러면, 리더는 디스크에 로그를 기록 + 팔로워에게 네트워크로 로그 전송을 해야한다.
        - 단점
            - 로그가 제일 저수준의 데이터를 기록한다.
            - WAL은 어떤 디스크 블록에서 어떤 바이트를 변경했는지와 같은 상세 정보를 포함한다. 이렇게 하면 복제가 저장소 엔진과 밀접하게 엮인다.
            - DB가 저장소 형식을 다른 버전으로 변경한다면 대개 리더와 팔로워의 DB 소프트웨어 버전을 다르게 가져갈 수 없다.
            - WAL 배송과 같이 복제 프로토콜이 버전의 불일치를 허용하지 않는다면 DB 업그레이드 시 중단시간이 필요하다.

    3. **논리적(로우 기반) 로그 복제**

        - **복제 로그를 저장소 엔진 내부와 분리하기 위한 대안 하나는 복제와 저장소 엔진을 위해 다른 로그 형식을 사용하는 것이다.** (이 같은 종류의 복제 로그를 저장소 엔진의 (**물리적**) 데이터 표현과 구별

하기 위해 **논리적 로그**라고 부른다.)

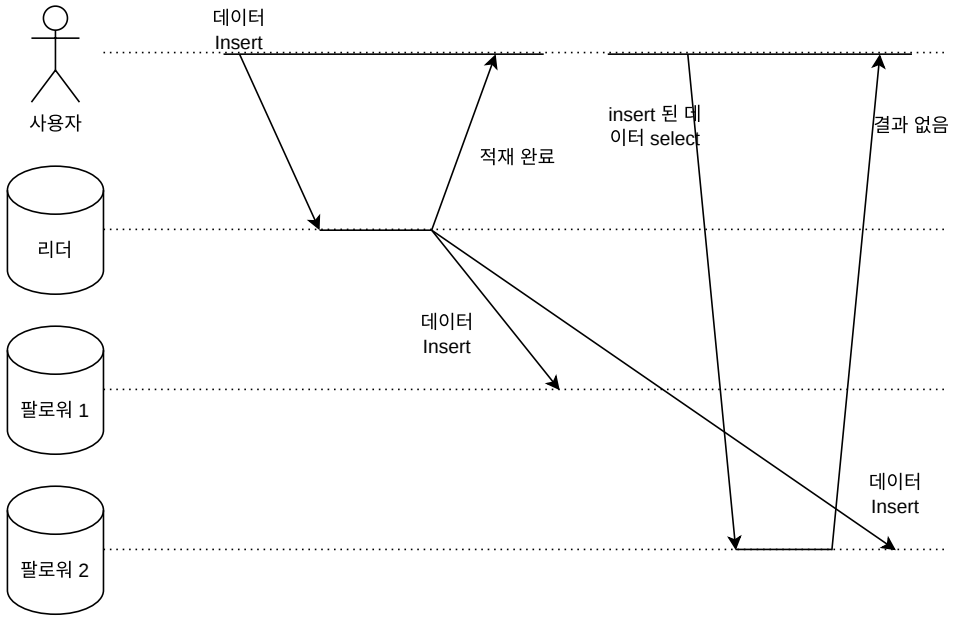
- 관계형 DB용 로그는 대개 로우 단위로 DB 테이블에 쓰기를 기술한 레코드 열이다.
  - 삽입된 로우의 로그는 모든 칼럼의 새로운 값을 포함한다.
  - 삭제된 로우의 로그는 로우를 고유하게 식별하는 데 필요한 정보를 포함한다.
    보통 이것은 기본키지만, 테이블에 기본키가 없다면 모든 칼럼의 예전 값을 로깅해야 한다.
  - 갱신된 로우의 로그는 로우를 고유하게 식별하는 데 필요한 정보와 모든 칼럼의 새로운 (적어도 변경된 모든 칼럼의 새로운 값)값을 포함한다.
- 여러 로우를 수정하는 트랜잭션은 여러 로그 레코드를 생성한 다음 트랜잭션이 커밋됐음을 레코드에 표시한다.
- 논리적 로그를 저장소 엔진 내부와 분리했기 때문에 하위 호환성을 더 쉽게 유지할 수 있고 리더와 팔로워에서 다른 버전의 DB SW나 심지어 다른 저장소 엔진을 실행할 수 있다.
- 또한 논리적 로그 형식은 외부 애플리케이션이 파싱하기 더 쉽다. 이런 측면은 오프라인 분석이나 사용자 정의 색인과 캐시 구축을 위해 데이터 웨어하우스 같은 외부 시스템에 DB의 내용을 전송하고자 할 때 유용하다. (이 기술을 **변경 데이터 캡처**라 한다.)

4. **트리거 기반 복제**
   - 앞 선 복제 접근 방식 3가지는 애플리케이션 코드의 사용 없이 DB 시스템에 의해 구현된다. 대부분 이 방식을 원하지만 조금 더 유연성이 필요한 상황이 있다.
     (ex. 데이터의 서브셋만 복제하거나 DB 를 다른 종류의 DB로 복제해야 하거나 충돌 해소 로직이 필요하다면 복제를 애플리케이션 단으로 옮겨야 한다)
   - 일반적인 것은 **트리거**나 **스토어드 프로시저(SP)**
   - 트리거는 사용자 정의 애플리케이션 코드를 등록할 수 있게 한다.
     - 이 애플리케이션 코드는 DB 시스템에서 데이터가 변경되면(쓰기 트랜잭션) 자동으로 된다.
     - 트리거는 데이터 변경을 분리된 테이블에 로깅할 수 있는 기회를 가진다.
     - 이 테이블로부터 데이터 변경을 외부 프로세스가 읽을 수 있다.
     - 그러면 외부 프로세스는 필요한 애플리케이션 로직을 적용해 다른 시스템으로 데이터 변경을 복제한다.
   - 일반적으로 트리거 기반 복제에는 다른 복제 방식보다 많은 오버헤드가 있다.
     DB에 내장된 복제보다 버그나 제한 사항이 더 많이 발생한다. 그래도 유연성 때문에 유용하다.

▼ 복제 지연 문제

- 정의
  - 복제는 노드 내결함성만 필요한 것은 아니다.
    확장성(단일 장비에서 감당하지 못하는 요청을 처리)과
    지연 시간(지리적으로 더 가까운 복제 서버를 위치 시킴)이 또 다른 이유이다.
  - 리더 기반 복제는 모든 쓰기가 단일 노드를 거쳐야 하지만 읽기 전용 질의는 어떤 복제에서도 가능하다.
    - 대부분이 읽기 요청이고 쓰기가 아주 작은 비율로 구성된 작업부하라면 많은 팔로워를 만들어 팔로워 간 읽기 요청을 분산하는 매력적인 옵션이 있다.
    - 이 방식을 사용하면 리더의 부하를 없애고 근처 복제 서버에서 읽기 요청을 처리할 수 있다.
  - 이런 **읽기 확장 아키텍쳐**에서는 간단히 팔로워를 더 추가함으로써 읽기 전용 요청을 처리하기 위한 용량을 늘릴 수 있다. 하지만 이 접근 방식은 실제로는 **비동기식 복제**에서만 동작한다.

        - 동기식으로 모든 팔로워에 복제를 시도한다면 단일 노드 장애나 네트워크 중단으로 전체의 쓰기가 불가능해진다.
        - 노드가 많아지면 다운될 가능성도 커져 완전한 동기식 설정은 매우 불안정하다.
    - 아쉽게도 애플리케이션이 **비동기 팔로워**에서 데이터를 읽을 때 팔로워가 뒤처진다면 지난 정보를 볼 수도 있다. (DB 불일치)
        - 이와 동시에 리더와 팔로워에 동일한 질의를 수행하면 모든 쓰기가 팔로워에 반영되지 않았기 때문에 서로 다른 결과를 얻을 수도 있다.
        - 하지만 , 이런 불일치는 일시적인 상태에 불과하다.
        - DB는 쓰기를 멈추고 잠시동안 기다리면 팔로워는 결국 따라잡게 되고 리더와 일치하게 된다. (최종적 일관성)
        - 정상적인 동작에서 리더에서 일어난 쓰기와 팔로워에서 반영 사이의 지연은 실제로는 아주 짧은 시간이다.
- 복제 지연이 있을 때 발생할 수 있는 3가지 사례
    - **자신이 쓴 내용 읽기**
        - 많은 애플리케이션은 사용자가 임의 데이터를 제출하고 해당 사용자에게 제출한 데이터를 볼 수 있게 한다. 새로운 데이터를 제출하면 리더에게 전송해야 하지만, 읽을 때는 팔로워에서 읽을 수 있다. 이것은 데이터를 자주 읽지만 가끔 쓰는 경우에 적합하다.
        - **비동기식 복제 문제**



            - 사용자가 쓰기를 수행후 직후 데이터를 본다면 새로운 데이터는 아직 복제 서버에 반영되지 않을 수도 있다. (사용자 경험 Down)
            - 이런 상황에서는 **쓰기 후 읽기 일관성(자신의 쓰기 읽기 일관성)**이 필요하다.
              정의) 사용자가 페이지를 재로딩했을 때 항상 자신이 제출한 모든 갱신을 볼 수 있음을 보장하며 다른 사용자에 대해서는 보장하지 않는다. (다른 사용자 갱신은 일정 시간 이후까지 보이지 않을 수 있다. 하지만 사용자 자신의 입력이 올바르게 저장됐음을 보장한다)
        - **리더 기반 복제 시스템에서 쓰기 후 읽기 일관성을 구현하는 법**

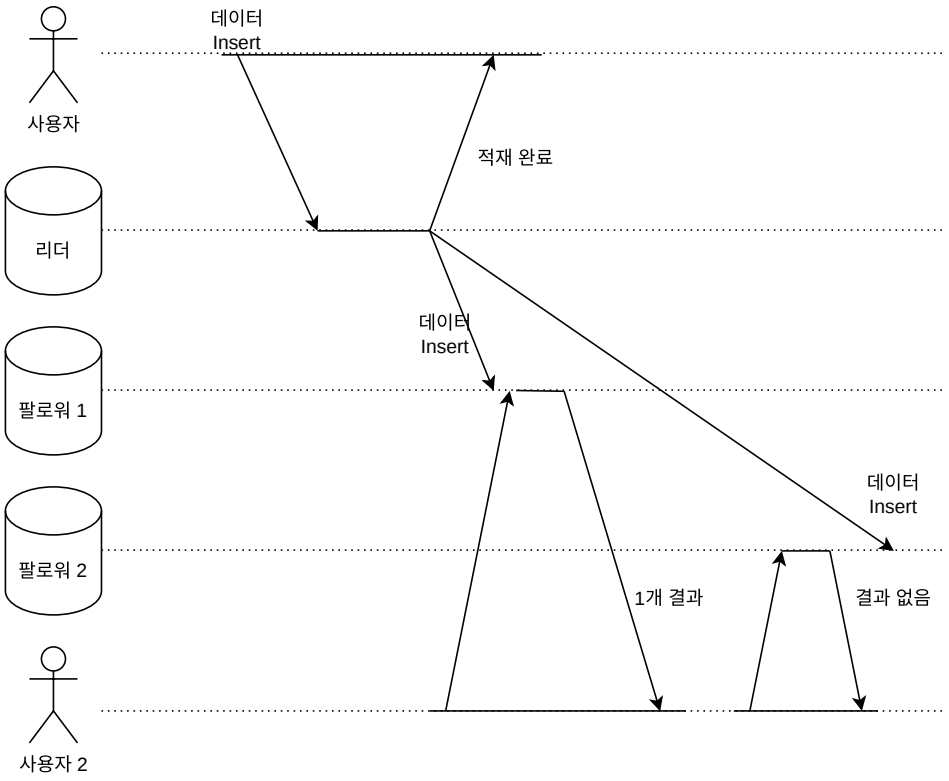
- 사용자가 수정한 내용을 읽을 때는 리더에서 읽는다. 그 밖에는 팔로워에서 읽는다.
  이를 위해선 실제로 질의하지 않고 무엇이 수정됐는지 알 수 있는 방법이 필요하다.
  ex) 사용자 프로필 정보는 보통 다른 사람이 아닌 프로필 소유자만 편집할 수 있음으로 사용자 소유 프로필은 리더에서 읽고 다른 사용자 프로필은 팔로워에서 읽는 규칙을 사용한다.
- 애플리케이션 내 대부분의 내용을 사용자가 편집할 가능성이 있다면 위 접근 방식은 대부분 리더에서 읽기 때문에 효율적이지 않다. 이런 경우에는 리더에서 읽을지 말지를 결정하기 위해 다른 기준을 사용해야 한다. ex) 마지막 갱신 시각을 찾아서 마지막 갱신 후 1분 동안은 리더에서 모든 읽기를 수행한다. 또한 팔로워에서 복제 지연을 모니터링해 리더보다 1분 이상 늦은 모든 팔로워에 대한 질의를 금할 수 있다.
- 클라이언트는 가장 최근 쓰기의 타임스탬프를 기억할 수 있다. 그러면 시스템은 사용자 읽기를 위한 복제 서버가 최소한 해당 타임스탬프까지 갱신을 할 수 있다. 복제 서버가 아직 최신 내용이 아닌 경우에는 다른 복제 서버가 읽기를 처리하거나 복제 서버가 따라잡을 때까지 질의를 대기시킬 수 있다. 타임스탬프는 **논리적 타임스탬프(로그 열 숫자처럼 쓰기 순서 지정)**거나 **실제 시스템 시간(실제 시스템 시간인 경우 동기화가 중요하다)**일 수 있다.
- 복제 서버가 여러 데이터센터에 분산(사용자에게 지리적인 근접성이나 가용성을 위해)됐다면 복잡도가 증가한다. 리더가 제공해야 하는 모든 요청은 리더가 포함된 데이터 센터로 라우팅돼야 한다.

▪ **동일한 사용자가 여러 디바이스로 서비스를 접근할 때에 문제**

- 이 경우에는 디바이스 간 쓰기 후 읽기 일관성이 제공돼야 한다.
  사용자가 한 디바이스에서 어떤 정보를 입력하면 다른 디바이스에서 볼 때는 방금 입력한 정보가 보여야 한다.
- 고려 항목
  - 사용자의 마지막 갱신 타임스탬프를 기억해야 하는 접근 방식은 더욱 어렵다.
    한 디바이스에서 수행 중인 코드는 다른 디바이스에서 발생한 갱신을 알 수 없기 때문이다.
    이 메타데이터는 중앙집중식으로 관리해야 한다.
  - 복제 서버가 여러 데이터센터 간에 분산돼 있다면 다른 디바이스의 연결이 동일한 데이터센터로 라우팅된다는 보장이 없다. 리더에서 읽어야 할 필요가 있는 접근법이라면 먼저 사용자 디바이스의 요청을 동일한 데이터센터로 라우팅해야 한다.

◦ **단조 읽기**

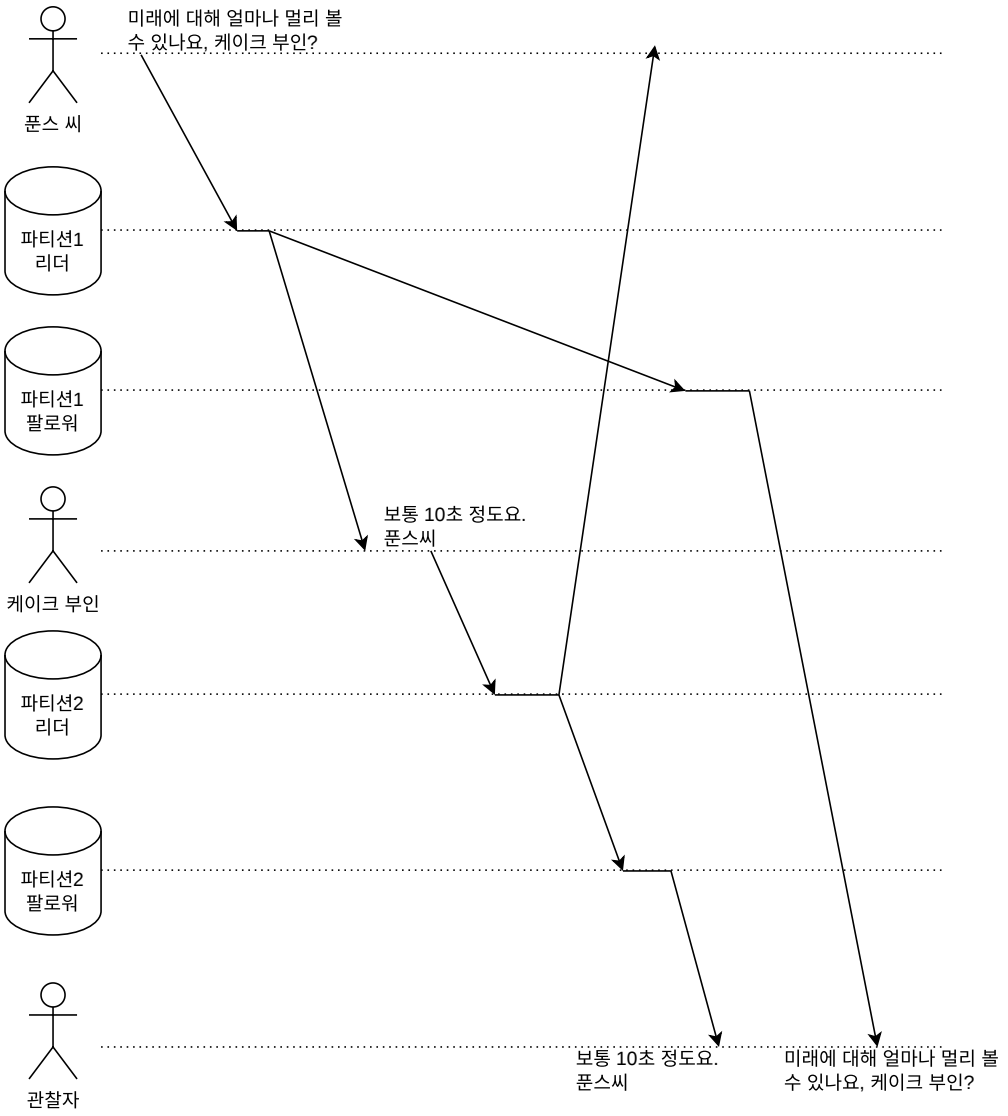
▪ **비동기식 팔로워에서 읽을 때 두 번째 문제 (시간이 거꾸로 흐르는 현상)**
(
사용자가 각기 다른 복제 서버에서 여러 읽기를 수행할 때 발생할 수 있다.)



- 쓰기시 첫 번째 팔로워는 지연이 거의 없고 두 번째 팔로워는 큰 지연이 있음
- 읽기는 첫 번째 질의는 결과를 출력하지만, 두 번째 질의는 결과를 출력하지 못한다.
  사용자 2 기준으로는 결과가 나왔다가 안 나오는 이상한 경험(UX)을 하게된다.
- **단조 읽기(monotonic read)**는 이런 종류의 이상 현상이 발생하지 않음을 보장한다.
  단조 읽기는 강한 일관성보다 덜한 보장이지만 최종적 일관성보다는 더 강한 보장이다.
  데이터를 읽을 때 이전 값을 볼 수 있다.
  한 사용자가 여러 번에 걸쳐 여러 번 읽어도 시간이 되돌아가는 현상을 보지 않는다는 의미다.
  즉, 이전에 새로운 데이터를 읽은 후에는 예전 데이터를 읽지 않는다.
- 단조 읽기를 달성하는 한 방법은 각 사용자의 읽기가 항상 동일한 복제 서버에서 수행되게끔 하는 것이다. (다른 사용자는 다른 복제 서버에서 읽을 수 있다.) ex) 임의 선택보다는 사용자 ID의 해시를 기반으로 복제 서버를 선택한다. 하지만 복제 서버가 고장 나면 사용자 질의를 다른 복제 서버로 재라우팅할 필요가 있다.

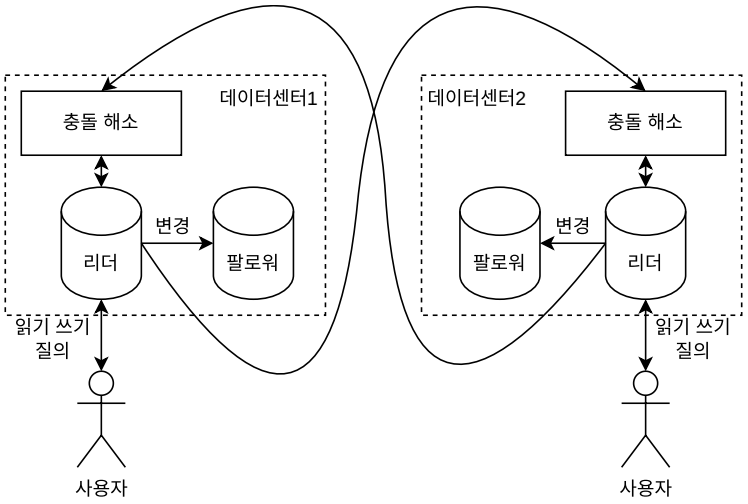
◦ **일관된 순서로 읽기**

▪ **세 번째 복제 지연 문제 (인과성의 위반 우려)**



- 위 문장 사이에는 인과성이 있다.
    - 푼스 씨 → 케이크 부인
      하지만, 관찰자 입장으로 케이크 부인 → 푼스 씨로 보일 것임으로 혼란 스러울 것이다.
- 이런 종류의 이상 현상을 방지하려면 **일관된 순서로 읽기(Consistent Prefix Read)** 같은 또 다른 유형의 보장이 필요하다. 일관된 순서로 읽기는 일련의 쓰기가 특정 순서로 발생한다면 이 쓰기를 읽는 모든 사용자는 같은 순서로 쓰여진 내용을 보게 됨을 보장한다.
- 파티셔닝된(샤딩된) DB에서 발생되는 특징적인 문제다.
  DB가 항상 같은 순서로 쓰기를 적용한다면 읽기는 항상 일관된 순서를 보기 때문에 이런 이상 현상은 일어나지 않는다. 하지만
  많은 분산 DB에서 서로 다른 파티션은 독립적으로 동작하므로 쓰기의 전역 순서는 없다.
  즉,
  사용자가 DB에서 읽을 때 예전 상태의 일부와 새로운 상태의 일부를 함께 볼 수 있다.
- 해결책
    - 서로 인과성이 있는 쓰기가 있는 동일한 파티션에 기록되게끔 하는 방법이다. 하지만 일부 애플리케이션에서는 효율적이지 않다.
    - 인과성을 명시적으로 유지하기 위한 알고리즘 (이후 이전 발생과 동시성)

- **복제 지연을 위한 해결책**
  - 최종적 일관성 시스템으로 작업할 때 복제 지연이 몇 분이나 몇 시간으로 증가한다면 애플리케이션이 어떻게 동작할지 생각해 볼 가치가 있다. 대답이 "문제 없음"이면 아주 좋지만 결과가 UX가 좋지 않다면 쓰기 후 읽기와 같은 강한 보장을 제공하게끔 시스템을 설계해야 한다. **사실 복제가 비동기식으로 동작하지만 동기식으로 동작하는 척 하는 것이 문제 해결 방안이다.**
  - 애플리케이션이 기본 DB보다 더 강력한 보장을 제공하는 방법이 있다. (ex. 특정 종류의 리더에서 읽기 수행) 하지만, 애플리케이션 코드에서 이 문제를 다루기에는 너무 복잡해서 잘못되기 쉽다.
  - 애플리케이션 개발자가 이런 미묘한 복제 문제를 걱정하지 않고 "올바른 작업 수행"을 위해 항상 DB를 신뢰할 수 있다면 훨씬 좋다. 이것이 **트랜잭션**이 있는 이유다. 트랜잭션은 애플리케이션이 더 단순해지기 위해 DB가 더 강력한 보장을 제공하는 방법이다.
  - 오랫동안 단일 노드 트랜잭션은 존재했다. 하지만 분산(복제되고 파티셔닝된) DB로 전환하는 과정에서 많은 시스템이 트랜잭션을 포기했다. 트랜잭션이 성능과 가용성 측면에서 너무 비싸고 확장 가능 시스템에서는 어쩔 수 없이 최종적 일관성을 사용해야 한다는 주장이 있다. 이 주장은 일부 사실이지만 지나치게 단순화됐다.
- **다중 리더 복제**
  - 정의
    - 리더 기반 복제에는 주요한 단점 하나가 있다. 리더가 하나만 존재하고 모든 쓰기는 해당 리더를 거쳐야 한다. 어떤 이유로 리더에 연결할 수 없다면 DB에 쓰기를 할 수 없다.
    - 리더 기반 복제 모델은 쓰기를 허용하는 노드를 하나 이상 두는 것으로 자연스럽게 확장한다. 복제는 여전히 같은 방식을 사용한다. 쓰기 처리를 하는 각 노드는 데이터 변경을 모든 다른 노드에 전달해야 한다. 이 방식을 **다중 리더** 설정(**마스터 마스터, 액티브/액티브**) 복제이라 부른다.
    - 다중 리더 복제는 많은 DB에 새로 추가된 기능임으로 미묘한 설정상의 실수나 다른 DB 기능과의 뜻밖에 상호작용이 있다. (ex. 자동 증가 키, 트리거, 무결성 제약 문제가 될 소지가 있다.) - 이런 이유로 다중 리더 복제는 가능하면 피해야 하는 위험한 영역이다.
  - 사용 사례
    - 다중 데이터센터 운영

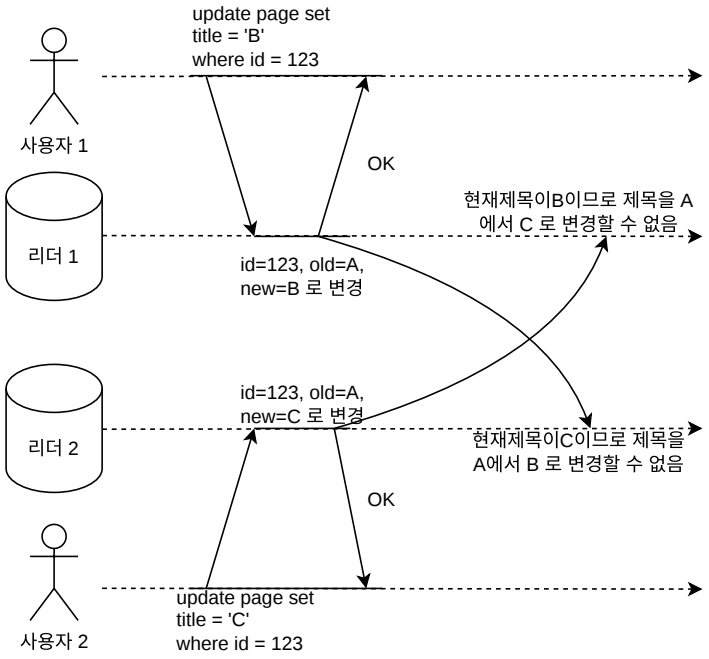


- 여러 다른 데이터센터에 DB 복제 서버가 있다면 (데이터센터 내결함성, 사용자 지리적 위치), 일반적인 리더 기반 복제 설정은 리더가 하나의 데이터 센터에 있고 모든 쓰기는 해당 데이터 센터를 거쳐야 한다.

- 다중 리더 설정에서는 각 데이터센터마다 리더가 있을 수 있다.
  각 데이터센터 내에는 보통의 리더의 팔로워 복제를 사용하고
  데이터 센터 간에는 각 데이터센터의 리더가 다른 데이터센터의 리더에게 변경 사항을 복제한다.
- 단일 리더 설정과 다중 리더 설정 배포 (다중 리더 장점)
  - 성능
    - 단일 리더 설정
      - 모든 쓰기는 인터넷을 통해 리더가 있는 데이터센터로 이동해야 한다. 이것은 쓰기에 지연 시간을 상당히 늘리는 원인이 된다. 그리고 처음에는 여러 데이터센터를 갖는 목적에도 위배될 수 있다.
    - 다중 리더 설정
      - 모든 쓰기는 로컬 데이터센터에서 처리한 다음 비동기 방식으로 다른 데이터센터에 복제한다. 따라서 데이터센터 간 네트워크 지연은 사용자에게 숨겨진다. 즉 사용자가 인지하는 성능이 더 좋다.
  - 데이터센터 중단 내성
    - 단일 리더 설정
      - 리더가 있는 데이터 센터가 고장 나면 장애 복구를 위해 다른 데이터센터에서 한 팔로워를 리더로 승진시킨다.
    - 다중 리더 설정
      - 각 데이터센터는 다른 데이터센터와 독립적으로 동작하고 고장 난 데이터센터가 온라인으로 돌아왔을 때 복제를 따라잡는다.
  - 네트워크 문제 내성
    - 데이터센터 간 트래픽은 보통 공개 인터넷을 통해 처리됨 (로컬 네트워크보다 안정성이 떨어짐)
    - 단일 리더 설정
      - 데이터센터 내 연결의 쓰기는 동기식임으로 데이터센터 내 연결 문제에 민감함.
    - 다중 리더 설정
      - 비동기 복제 사용함으로 네트워크 문제 내성이 단일 보다 좋다.
        일시적인 네트워크 중단에도 쓰기 처리는 진행되기 때문
- 다중 리더 단점
  - 동일한 데이터를 다른 두 개의 데이터센터에서 동시에 변경할 수 없다.
    이때 발생하는
    쓰기 충돌은 반드시 해소해야 한다.

- 오프라인 작업을 하는 클라이언트
  - 다중 리더 복제가 적절한 또 다른 상황은 인터넷 연결이 끊어진 동안 애플리케이션이 계속 동작해야 하는 경우
  - 오프라인 상태에서 데이터를 변경하면 디바이스가 다음에 온라인 상태가 됐을 때 서버와 다른 디바이스를 동기화해야 한다.

        - 이 경우 모든 디바이스에는 리더 처럼 동작하는 로컬 DB가 있다. (쓰기 요청을 받아야 함) 그리고 모든 디바이스 상에서 복제 서버 간 다중 리더 복제를 비동기 방식으로 수행하는 프로세스(동기화)가 있다. 복제 지연은 사용자가 인터넷이 접근이 가능해진 시점에 따라 몇 시간 ~ 며칠 이상 걸릴 수 있다.
        - 아키텍처 관점으로 보면 이 설정은 근본적으로 데이터센터 간 다중 리더 복제와 동일하다. (각 디바이스 = 데이터센터, 디바이스간 네트워크 연결은 신뢰 불가능)
    - 협업 편집
        - 동시에 여러 사람이 문서를 편집할 수 있는 애플리케이션을 **실시간 협업 편집** 애플리케이션이라 함.
        - 일반적으로 협업 편집을 DB 복제 문제라 생각하지 않는다. 그러나 앞에 언급한 오프라인 편집 사용 사례와 공통점이 많다. (한 사용자가 문서를 편집할 때 변경 내용을 즉시 로컬 복제 서버에 적용하고 나서 동일한 문서를 편집하는 다른 사용자와 서버에 비동기식으로 복제한다.)
        - 편집 충돌이 없음을 보장하려면 애플리케이션은 사용자가 편집하기 전에 문서의 잠금을 얻어야 한다. 다른 사용자가 같은 문서를 편집하려면 첫 번째 사용자의 변경이 커밋되고 잠금이 해제될 때까지 대기해야한다. (리더에서 트랜잭션을 사용하는 단일 리더 복제와 동일)
- **쓰기 충돌 다루기**
    - 정의
        - 다중 리더 복제에서 제일 큰 문제는 쓰기 충돌이 발생한 다는 점이다. (충돌 해소 必)

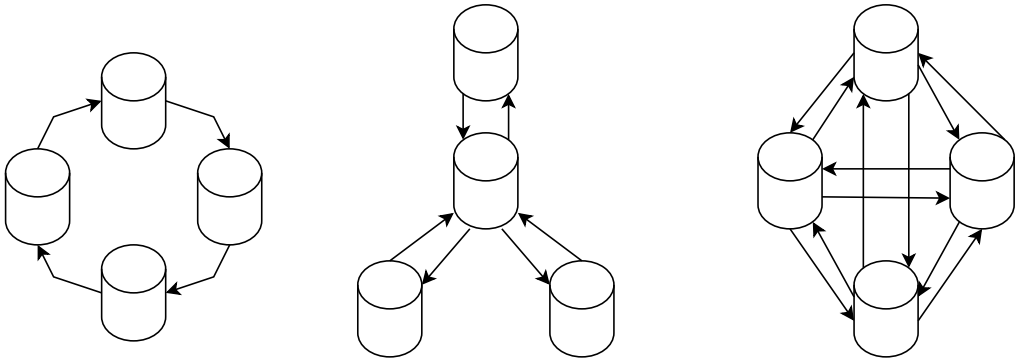


        - (다중 리더 복제일 때) 변경을 비동기로 복제할 때 충돌을 감지한다.
    - **동기 대 비동기 충돌 감지**
        - 단일 리더 DB 에서 첫 번째 쓰기가 완료될 때까지 두 번째 쓰기를 차단해 기다리게 하거나 두 번째 쓰기 트랜잭션을 중단해 사용자가 쓰기를 재시도하게 한다.
        - 반면 다중 리더 설정에서는 두 쓰기는 모두 성공하며 충돌은 이후 특정 시점에서 비동기로만 감지한다. (이때, 사용자에게 충돌을 해소하게끔 요청하면 너무 늦을 수도 있다.)

        - 이론식으로 충돌 감지는 동기식으로 만들 수 있다.
          즉, 쓰기가 성공한 사실을 사용자에게 말하기 전에 모든 복제 서버가 쓰기를 복제하기를 기다린다.
          하지만 이렇게 하면 다중 리더 복제의 주요 장점(각 복제 서버가 독립적 쓰기를 허용)을 잃는다. 동기식으로 충돌 감지를하려면 단일 리더 복제만 사용해야 할 수도 있다.
    - **충돌 회피**
        - 가장 간단한 전략 (사용자는 하나의 데이터센터에 쓰기, 읽기를 보장한다)
        - 특정 레코드의 모든 쓰기가 동일한 리더를 거치도록 애플리케이션이 보장한다면 충돌은 발생하지 않는다. 많은 다중 리더 복제 구현 사례에서 충돌은 잘 처리하지 못하기 때문에 충돌을 피하는 것이 자주 권장되는 방법이다.
        - ex) **사용자가 자신의 데이터를 편집할 수 있는 애플리케이션에서 특정 사용자의 요청을 동일한 데이터센터로 항상 라우팅하고 데이터센터 내 리더를 사용해 읽기와 쓰기를 하게끔 보장할 수 있다**. 다른 사용자는 서로 다른 데이터센터 (아마도 사용자와의 지리적 근접성을 기반으로 선택)를 가질 수 있지만. 한 사용자 관점으로 보면 구성은 기본적으로 단일 리더이다.
        - 하지만, 때때로 한 데이터센터가 고장 나서 트래픽을 다른 데이터센터로 다시 라우팅해야 하거나 다른 지역으로 이동해 현재는 다른 데이터센터가 가깝다면 레코드를 위해 지정된 리더를 변경하고 싶을 수도 있다. 이런 경우는 충돌 회피가 실패한다.
    - **일관된 상태 수렴**
        - 단일 리더 DB는 순차적인 순서로 쓰기를 적용한다.
          동일한 필드를 여러 번 갱신한다면 마지막 쓰기가 필드의 최종 값으로 결정된다.
        - 다중 리더 설정에서는 쓰기 순서가 정해지지 않아 최종 값이 무엇인지 명확하지 않다.
        - 단순하게 각 복제 서버가 쓰기를 본 순서대로 적용한다면 DB는 결국 일관성 없는 상태가 된다. (그림에서는 리더 1 최종 값이 C가되고 리더 2 최종 값은 B가 된다.) 이 상황은 용인되지 않는다. 모든 복제 계획은 모든 복제 서버가 최종적으로는 동일하다는 사실을 보장해야 한다.
        - 따라서 DB는 **수렴 방식**으로 충돌을 해소해야 한다. 이는 모든 변경이 복제돼 모든 복제서버에 동일한 최종 값이 전달되게 해야 한다는 의미이다.
        - 수렴 충돌 해소 방법
            - 각 쓰기에 고유 ID를 부여하고 가장 높은 ID를가진 쓰기를 고른다. 다른 쓰기는 버린다. 타임스탬프를 사용하는 경우 **최종 쓰기 승리(LWW)**라 한다. 이 접근 방식은 대중적이지만 데이터 유실 위험이 있다.
            - 각 복제 서버에 고유 ID를 부여하고 높은 숫자의 복제 서버에서 생긴 쓰기가 낮은 숫자의 복제 서버에서 생긴 쓰기보다 항상 우선적으로 적용되게 한다. 이 접근 방식 또한 데이터 유실 가능성이 있다.
            - 어떻게든 값을 병합한다. ex) 사전 순으로 정렬 후 연결 (위 그림으로는 B/C)
            - 명시적 데이터 구조에 충돌을 기록해 모든 정보를 보존한다. 나중에 충돌을 해소하는 애플리케이션 코드를 작성한다.
    - **사용자 정의 충돌 해소 로직**
        - 충돌을 해소하는 가장 적합한 방법은 애플리케이션에 따라 다르다. 따라서 대부분의 다중 리더 복제 도구는 애플리케이션 코드를 사용해 충돌 해소 로직을 작성한다. 해당 코드는 쓰기나 읽기 수행 중에 실행될 수 있다.
            - 쓰기 수행 중

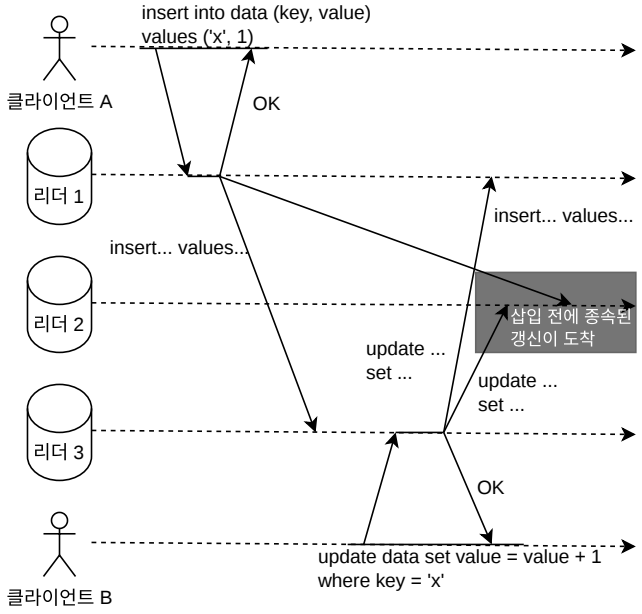
        - 복제된 변경 사항 로그에서 DB 시스템이 충돌을 감지하자마자 충돌 핸들러를 호출한다. 이 핸들러는 일반적으로 사용자에게 충돌 내용을 표시하지 않는다. 그리고 백그라운드 프로세스에서 빠르게 실행되어야 한다.
    - 읽기 수행 중
        - 충돌을 감지하면 모든 충돌 쓰기를 저장한다. 다음 번 데이터를 읽을 때 이런 여러 버전의 데이터가 애플리케이션에 반환된다. 애플리케이션은 사용자에게 충돌 내용을 보여주거나 자동으로 충돌을 해소할 수 있다. 충돌을 해소한 결과는 다시 DB에 기록한다.
  - 충돌 해소는 보통 전체 트랜잭션이 아니라 개별 로우나 문서 수준에서 적용된다.
    따라서 원자적으로 여러 다른 쓰기를 수행하는 트랜잭션이라면, 각 쓰기는 충돌 해소를 위해 여전히 별도로 간주된다.
- **자동 충돌 해소**
  - 충돌 해소 규칙은 빠르게 복잡해질 수 있고 맞춤형 코드는 오류가 발생할 수 있다.
  - 충돌 없는 데이터 타입
    - 셋, 맵, 정렬 목록, 카운터 등을 위한 데이터 구조의 집합으로 동시에 여러 사용자가 편집할 수 있고, 합리적인 방법으로 충돌을 자동 해소 한다.
  - 병합 가능한 영속 데이터 구조
    - 깃 버전 제어 시스템과 유사하고 명시적으로 히스트로리를 추적하고 삼중 병합 함수를 사용한다.
  - 운영 반환
    - 협업 편집 애플리케이션의 충돌 해소 알고리즘이다. 특히 텍스트 문서를 구성하는 문자 목록과 같은 정렬된 항목 목록의 동시 편집을 위해 설계됐다.
- **충돌은 무엇인가?**
  - 어떤 종류의 충돌은 명확하다.
    (위 그림 두 번째 쓰기는 동일한 레코드의 동일한 필드를 동시에 수정해 두 개의 다른 값으로 설정한다.) - 충돌 확정
  - 어떤 종류의 충돌은 감지하기 조금 더 어렵다.
    (ex. 회의실 예약 시스템, 애플리케이션은 각 회의실이 특정 시간대에 한 사람만 예약하게 끔 보장해야 한다. 이 때 같은 시간에 같은 회의실을 예약하는 두 개의 다른 예약이 생기면 충돌이 발생한다. 사용자 예약을 허용하기 전에 애플리케이션이 예약 가능한지 확인하더라도 두 예약이 각기 다른 리더에서 이뤄지면 충돌이 발생할 수 있다.)

- **다중 리더 복제 토폴로지**
  - **복제 토폴로지**는 한 노드에서 다른 노드로 전달하는 통신 경로를 뜻한다.
    리더가 둘 이상이라면 다양한 토폴로지가 가능하다.



원형 토폴로지 별 모양 토폴로지 전체 연결 토폴로지

- **원형 토폴로지**
  - 각 노드가 하나의 노드로부터 쓰기를 받고, 쓰기를 자신을 포함 + 다른 1개 노드에 전달한다.
- **별 모양 토폴로지**
  - 지정된 루트 노드 하나가 다른 모든 노드에 쓰기를 전달한다.
  - 트리로 일반화 할 수 있다.
- **전체 연결(all-to-all)**
  - 가장 일반적인 방식
  - 모든 리더가 각자의 쓰기를 다른 모든 리더에게 전송한다.
  - 메시지가 여러 경로를 따라 이동할 수 있으면 단일 장애점을 피할 수 있기 때문에 토폴로지 내결함성이 보다 좋음
  - 단점.
    일부 네트워크 연결이 다른 연결보다 빠르면 일부 복제 메시지가 다른 메시지를 추월 할 수 있다.

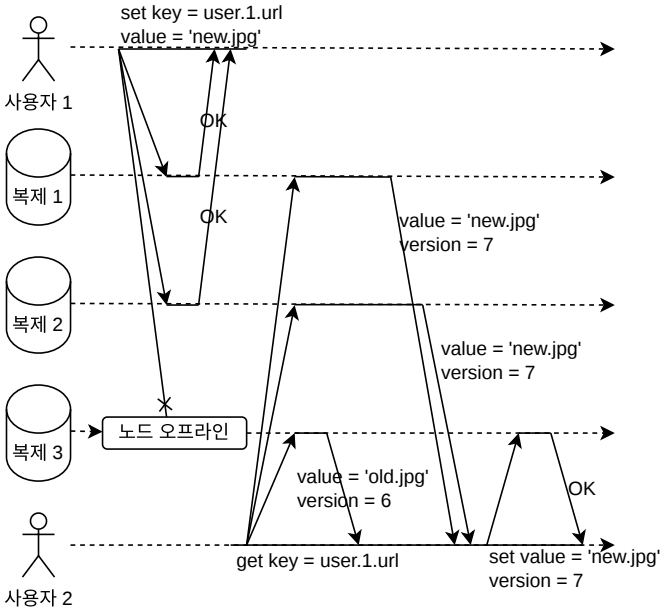


- 먼저 update을 받고 (리더 2 관점에서 DB에 없는 로우의 update) 나중에 해당 insert (update 전 insert) 를 받는다.

        - 인과성 문제, 갱신은 이전 삽입에 종속적이라 모든 노드에서 먼저 insert을 처리한 다음 update를 처리 해야한다. (모든 쓰기에 간단히 타임스탬프를 추가하는 방식으로는 충분하지 않다. 얼마나 대기 시간을 줘야 동기화됬다고 보장되는게 아님으로 (리더 2에서 이런 이벤트를 올바르게 정렬하기에 충분할 정도로 노드들의 시간이 동기화 됐다고 신뢰할 수 없기 때문))
        - 이런 이벤트를 올바르게 정렬하기 위해 **비전 벡터**라고 하는 기법을 사용해야 한다.
    - 원형, 별 모양 토폴로지 특징
        - 쓰기는 모든 복제 서버에 도달하기 전에 여러 노드를 거쳐야 한다.
          그러므로 노드들은 다른 노드로부터 받은 데이터 변경 사항을 전달해야 한다.
        - 무한 복제 루프를 방지하기 위해 각 노드에는 고유 식별자가 있고 복제 로그에서 각 쓰기를 거치는 모든 노드의 식별자가 태깅된다. (자신의 식별자가 태깅된 경우 이미 처리된 내용임으로 무시함)
        - 단점.
          하나의 노드에 장애가 발생하면 장애가 다른 노드 간 복제 메시지 흐름에 방해를 준다. 해당 노드가 복구될 때까지 통신할 수 없음.
          토폴로지는 장애 노드를 회피하게끔 재설정 가능하지만, 대부분이 수동으로 진행됨.

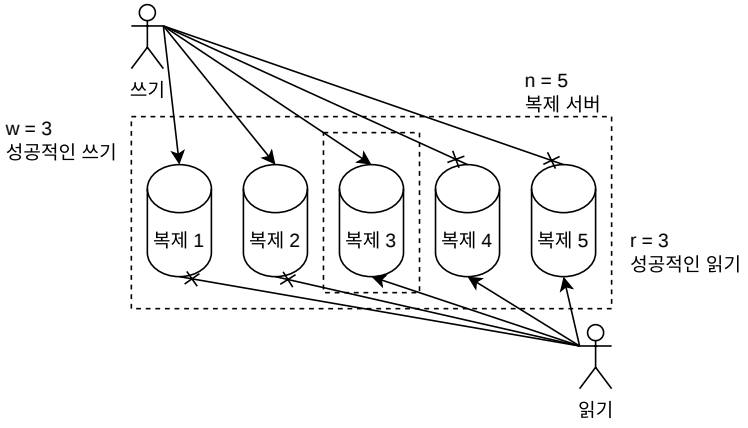
▼ 리더 없는 복제

- **정의**
    - 리더의 개념을 버리고 모든 복제 서버가 클라이언트로부터 쓰기를 직접 받을 수 있게 허용하는 접근 방식
    - **다이나모(Dynamo)** 시스템 (리더 없는 복제)
    - 일부 리더 없는 복제 구현에서는 클라이언트가 여러 복제 서버에 쓰기를 직접 전송하는 반면 코디네이터 노드(coordinator node)가 클라이언트를 대신해 이를 수행하기도 한다.
      하지만, 리더 DB와 달리 코디네이터 노드는 특정 순서로 쓰기를 수행하지 않는다.
- **노드가 다운됐을 때 DB에 쓰기**
    - 세 개의 복제 서버를 가진 DB가 있고 복제 서버 중 하나를 사용 할 수 없다고 가정한다면 리더 기반 설정에서쓰기 처리를 계속하려면 장애 복구를 해야 한다.
    - 반면, 리더 없는 설정에서는 장애 복구가 필요하지 않다.



- 클라이언트 (사용자 1)가 쓰기를 세 개의 모든 복제 서버에 병렬로 전송한다.
  세 개의 복제 서버 중 두 개의 서버가 쓰기를 확인하면 충분하다고 가정한다면, 쓰기가 성공한 것으로 간주된다.
- 사용할 수 없었던 노드가 다시 온라인이 되고 클라이언트가 이 노드에서 읽기를 시작한다면, 클라이언트는 **오래된 값(outdated)**을 받을 수 있다.
- 이 문제를 해결하기 위해서는 클라이언트가 DB에서 읽을 때 하나의 복제 서버로 요청을 보내지 않고 **읽기 요청을 병렬로 여러 노드에 전송**한다. (버전을 이용해 최신 내용을 결정한다.)

- **읽기 복구와 안티 엔트로피**
  - 복제 계획은 최종적으로 모든 데이터가 모든 복제 서버에 복사된 것을 보장해야 한다.
  - (오프라인 → 온라인 후 누락된 쓰기) 다이나모 시스템 데이터스토어에 두 가지 매커니즘
    - 읽기 복구
      - 클라이언트가 여러 노드에서 병렬로 읽기를 수행하면 오랜된 응답을 감지할 수 있다.
      - 값을 자주 읽는 상황에 적합
    - 안티 엔트로피 처리
      - 추가적으로 일부 데이터스토어는 백그라운드 프로세스를 두고 복제 서버 간 데이터 차이를 지속적으로 찾아 누락된 데이터를 하나의 복제 서버에서 다른 서버로 복사한다.
      - 리더 기반 복제에서의 복제 로그와 달리 이 **안티 엔트로피 처리**는 특성 순서로 쓰기를 복사하기 때문에 데이터가 복사되기까지 상당한 지연이 있을 수 있다.
- **읽기와 쓰기를 위한 정족수**
  - 위 그림 설명에 (읽기 또는 쓰기)를 성공 한 것으로 간주할 수 있는 수를 정족수라고 함. 위 예에서는 n = 3, w =2
  - n 개의 복제 서버가 있을 때
    모든 쓰기는 w 개의 노드에서 성공해야 쓰기가 확정되고
    모든 읽기는 최소한 r개의 노드에 질의해야 한다.
  - 다이나모 스타일 DB에서 n, w, r 은 설정 가능하다.
    일반적으로 n 은 홀수, w = r = (n + 1) / 2 반올림으로 설정한다.
  - 이러한 r, w 를 읽기와 쓰기를 **정족수 읽기, 정족수 쓰기**라 부른다.



  - 정족수 조건이 w + r > n 이라면

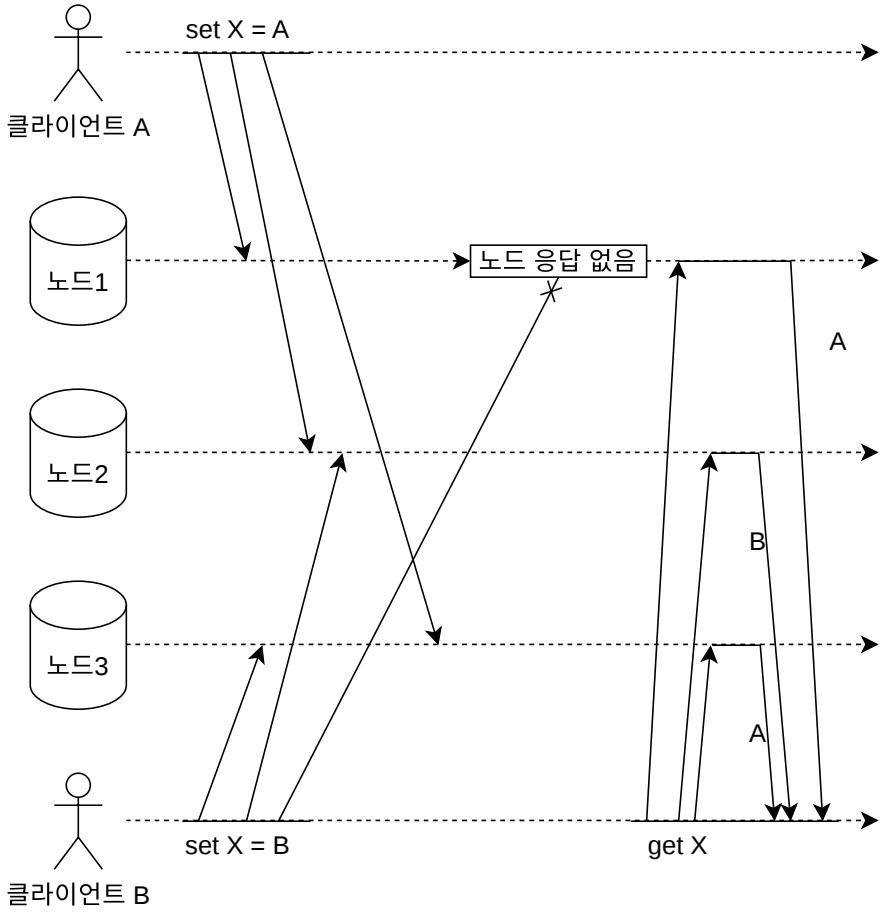
        - 읽기 시, 최신 값을 얻을 것으로 기대한다.
        - w < n 이면 노드 하나를 사용할 수 없어도 여전히 쓰기를 처리할 수 있다.
        - r < n 이면 노드 하나를 사용할 수 없어도 여전히 읽기를 처리할 수 있다.
        - n = 3, n = 2, r = 2 이면 사용 불가능한 노드 하나를 용인한다.
        - n = 5, w = 3, r = 3 이면 사용 불가능한 노드 둘을 용인한다.
        - 일반적으로 읽기와 쓰기는 항상 모든 n 개의 복제 서버에 병렬로 전송한다. w, r 은 얼마나 많은 노드를 기다릴지 결정한다.
          즉, 읽기나 쓰기가 성공했다고 간주하려면 n개의 노드 중에 몇 개의 노드에서 성공을 확인해야 하는지를 나타낸다.
        - 필요한 w나 r개 노드 보다 사용 가능한 노드가 적다면 쓰기나 읽기는 에러를 반환한다. (오류를 구분할 필요는 없음)
- **정족수 일관성의 한계**
    - 보통 r, w를 노드의 과반수 (n/2 초과)로 선택한다.
      이유는 n/2 노드 장애까지 허용해도 w + r > n 이 보장되기 때문이다.
      하지만, 정족수가 다수 일 필요는 없다.
      읽기와 쓰기 동작에서 사용되는 노드 셋 중 하나의 노드만 겹치면 된다.
    - w, r 이 작을수록 오래된 값을 읽을 확률이 높다.
      최신 값을 가진 노드가 읽을 노드에 포함되지 않을 가능성이 높기 때문이다.
    - 네트워크 중단으로 많은 복제 서버가 응답하지 않으면 읽기와 쓰기 처리 가 계속 진행될 가능성이 높다.
      응답할 수 있는 복제 서버의 수가 w 나 r 보다 아래로 떨어지면 DB 에서 읽기, 쓰기가 불가능하다.
    - 하지만 w + r > n 인 경우에도 오래된 값을 반환하는 에지 케이스가 있다.
        - 느슨한 정족수를 사용하면 w 개의 쓰기는 r 개의 읽기와 다른 노드에서 수행될 수 있음으로 r개의 노드와 w개의 노드가 겹치는 것을 보장하지 않는다.
        - 두 개의 쓰기가 발생하면 어떤 쓰기가 먼저 일어났는지 분명하지 않다. 이 경우 안전한 해결책은 동시 쓰기를 합치는 방법밖에없다. (승자가 타임스탬프를 기반으로 결정되면 (최종 쓰기 승리), clock skew로 인해 쓰기가 유실될 수 있음) (자세한 건 요기)
        - 쓰기가 읽기와 동시에 발생하면 쓰기는 일부 복제 서버에만 반영될 수 있다.
          이 경우 읽기가 예전 값 또는 최신 값을 반환하는지 여부가 분명하지 않다.
        - 쓰기가 일부 복제 서버에서는 성공했지만 다른 복제 서버에서 실패해 전체에서 성공한 서버가 w 복제 서버보다 적다면 성공한 복제 서버에서는 롤백하지 않는다. 이는 쓰기가 실패한 것으로 보고되면 이어지는 읽기에 해당 쓰기 값을 반환될 수도 아닐 수도 있다는 의미이다.
        - 새 값을 전달하는 노드가 고장나면 예전 값을 가진 다른 복제 서버에서 해당 데이터가 복원되고 새로운 값을 저장한 복제 서버 수가 w 보다 낮아져 정족수 조건이 깨진다.
        - 모든 과정이 올바르게 동작해도 시점 문제로 에지 케이스가 있을 수 있다.
    - 따라서 정족수가 읽기 시 최근에 쓴 값을 반환하게 끔 보장하지만 실제로는 어렵다.
    - 매개 변수 w, r로 오래된 값을 읽는 확률을 조정할 수있지만, 이는 절대적 보장이 아니다.
    - 최신성 모니터링
        - 운영 관점에서 볼 때 DB가 최신 결과를 반환하는지 여부를 모니터링하는 일은 중요하다. 애플리케이션이 오래된 값 읽기를 허용하더라도 복제 상태에 대해 알아야 한다. 복제가 명확히 뒤쳐진다면 원인을 조사할 수 있게 알려줘야 한다.

        - 리더 기반 복제에서 DB는 일반적으로 복제 지연에 대한 지표를 노출한다.
          이는, 쓰기가 리더에 적용되고 같은 순서로 팔로워에도 적용되고 각 노드가 복제 로그의 위치를 가지기 때문에 가능하다.
        - 리더 없는 복제에서는 쓰기가 적용된 순서를 고정할 수 없어, 모니터링이 조금 더 어렵다. 더욱이 DB가 읽기 복구만 사용한다면 자주 읽히지 않는 값이 얼마나 오래된 것인지에 대한 제한이 없어 오래된 복제 서버에서 반환된 값은 아주 오래된 값일 수 있다.
- **느슨한 정족수와 암시된 핸드오프**
    - 적절히 설정된 정족수가 있는 DB는 장애 복구 없이 개별 노드 장애를 용인한다.
      요청은 w나 r개 노드가 응답할 때 반환할 수 있어 모든 n개 노드가 응답할 때까지 기다릴 필요가 없기 때문에 개별 노드의 응답이 느려지는 것도 허용 가능하다.
      이런 특성 때문에 높은 가용성과 낮은 지연 시간이 필요하다. 가끔 오래된 값 읽기를 허용하는 사례에는 리더 없는 복제 기능을 가진 DB가 매력적이다.
    - 하지만 정족수는 내결함성이 없다.
      네트워크 중단으로 다수 DB 노드와 클라이언트가 쉽게 끊어질 수 있다.
      이 상황에서는 응답 가능한 노드가 w 나 r 보다 적을 가능성이 있으므로 클라이언트는 더 이상 정족수를 충족할 수 없다.
    - 노드가 n 개 이상인 대규모 클러스터에서 클라이언트는 네트워크 장애 상황에서 일부 DB 노드에 연결될 가능성이 있다. 이 경우 DB 설계자는 트레이드 오프에 직면된다.
        - w 나 r 노드 정족수를 만족하지 않는 모든 요청에 오류를 반환하는게 좋을까?
        - 아니면 일단 쓰기를 받아들이고 값이 보통 저장되는 n개 노드에 속하지 않지만 연결할 수 있는 노드에 기록할까? (aka. **느슨한 정족수)**
    - **느슨한 정족수**
        - 쓰기와 읽기는 여전히 w와 r의 성공 응답이 필요하지만 값을 위해 지정된 n 개의 “홈” 노드에 없는 노드가 포함될 수 있다.
        - 가용성을 높이는데 유용하다.
          모든 w개 노드를 사용할 수 있는 동안 DB는 쓰기를 받아드릴 수있다.
          하지만 이것은 w + r > n 인 경우에도 키의 최신 값을 읽는다고 보장하지 않는다.
        - 지속성에 대한 보장으로 데이터가 w 노드 어딘가에 저장된다는 의미.
          암시된 핸드오프가 완료될 때까지 r 노드의 읽기가 저장된 데이터를 본다는 보장이없다
    - **암시된 핸드오프**
        - 네트워크 장애 상황이 해제되면 한 노드가 다른 노드를 위해 일시적으로 수용한 모든 쓰기를 해당 “홈” 노드에 전송한다.
    - **다중 데이터센터 운영**
        - 리더 없는 복제도 동시 쓰기 충돌, 네트워크 중단, 지연 시간 급증을 허용하기 때문에 다중 데이터센터 운영이 적합하다.
        - 방법
            - n 개의 복제 서버 수에는 모든 데이터센터의 노드가 포함되고 설정에서 각 데이터센터마다 n 개의 복제 서버 중 몇 개를 보유할지를 지정할 수 있다. 클라이언트의 각 쓰기는 데이터센터 상관없이 모든 복제 서버에 전송되지만, 클라이언트는 보통 로컬 데이터센터 안에서 정족수 노드의 확인 응답을 기다리기 때문에 데이터센터 간 연결의 지연과 중단에 영향을 받지 않는다. 다

른 데이터센터에 대한 높은 지연 시간의 쓰기는 설정에 어느 정도 유연성이 있지만 대개 비동기로 발생하게끔 설정한다.

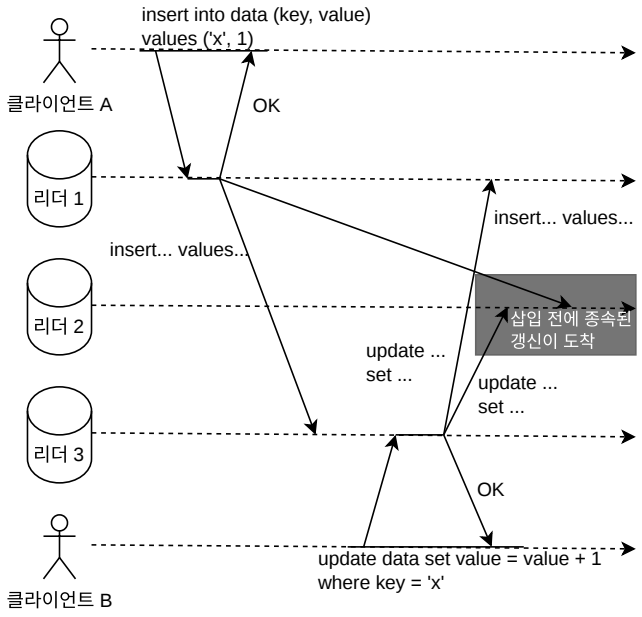
- 클라이언트와 DB간 모든 연결이 하나의 데이터센터의 로컬에서 이뤄지게 하기 때문에 n은 하나의 데이터 센터 안에 있는 복제 서버 수를 나타낸다. DB 클러스터들의 데이터센터 간 복제는 백그라운드에서 비동기로 일어나며 방식은 다중 리더 복제와 유사

- **동시 쓰기 감지**
  - 정의
    - 다이나모 스타일 DB는 여러 클라이언트가 동시에 같은 키에 쓰는 것을 허용하기 때문에 엄격한 정족수를 사용하더라도 충돌이 발생한다.
    - 리더 기반 복제와는 다르게, 다이나모 스타일 DB에서 충돌은 읽기 복구나 암시된 핸드오프 중에도 발생할 수 있다.
    - 문제는 다양한 네트워크 지연과 부분적인 장애 때문에 이벤트가 다른 노드에 다른 순서로 도착할 수 있다는 것이다.
      - ex) 클라이언트 A,B 가 키 X를 동시에 세 노드 데이터스토어에 기록하는 상황

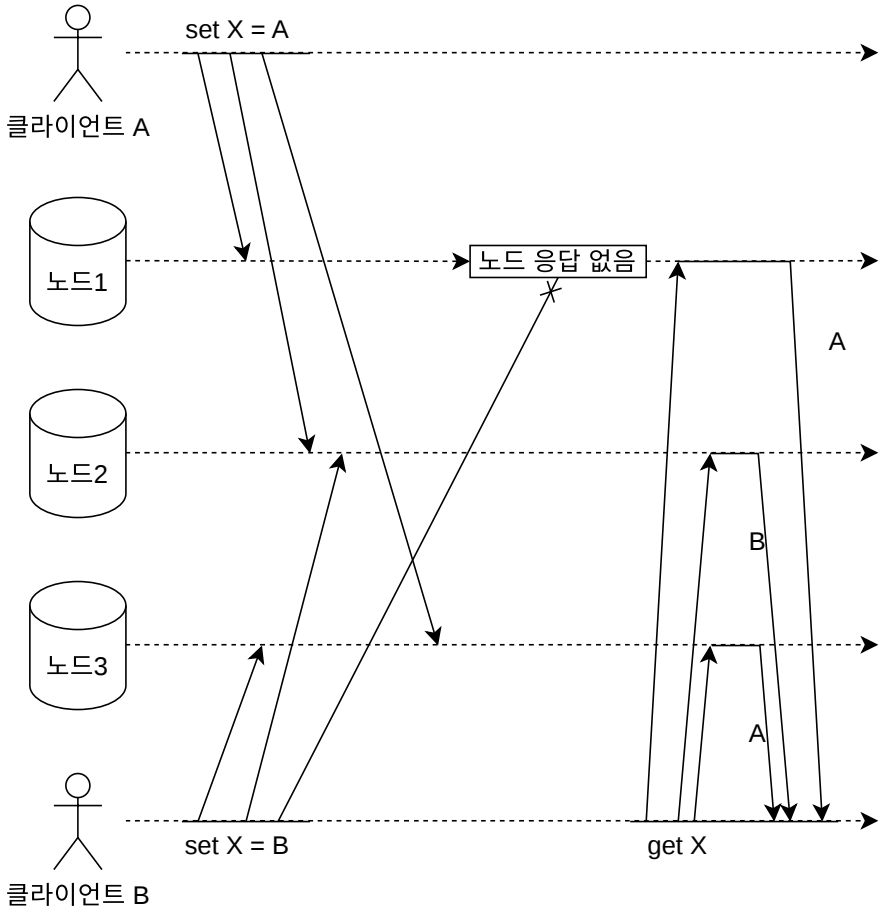


      - 노드1은 A로부터 쓰기를 받지만 순간적 장애로 B로부터 쓰기를 못 받음
      - 노드2는 A로부터 쓰기를 먼저 받고 그 다음 B로부터 쓰기를 받음
      - 노드3은 B로부터 쓰기를 먼저 받고 그 다음 A로부터 쓰기를 받음

        - 예시 처럼 쓰기 요청이 있을 때 마다 키의 값을 단순하게 덮어 쓴다면 노드들의 키를 영구적으로 일관성이 깨진다.
    - 최종적인 일관성을 달성하기 위해 복제본들은 동일한 값이 돼야 한다.
- **최종 쓰기 승리 (동기 쓰기 버리기)**
    - 최종적으로 값을 수렴하기 위한 접근 방식 하나는 각 복제본이 가진 "예전" 값을 버리고 가장 "최신" 값으로 덮어쓰는 방법이다.
    - 어떤 쓰기가 "최신"인지 명확하게 결정할 수 있는 한 모든 쓰기는 최종적으로 모든 복제 서버에 복사되므로 복제본은 최종적으로 동일한 값에 수렴한다.
    - 클라이언트가 쓰기 요청을 DB 노드에 전송할 때 다른 클라이언트에 대해서는 아는 것이 없기 때문에 어떤 이벤트가 먼저 발생했는지 확실하지 않다.
      이벤트 순서가 정해지지 않았기 때문에 그냥 동시 쓰기로 해야한다.
    - 비록 쓰기는 자연적인 순서가 없지만 임의로 순서를 정할 수 있다.
      ex) 쓰기에 타임스탬프를 붙여 가장 "최신"이라는 의미로 제일 큰 타임스탬프를 선택하고 예전 타임스탬프를 가진 쓰기는 무시한다.
      이를
      최종 쓰기 승리(LWW)라 부르는 충돌 해소 방법이다.
        - LWW는 최종적 수렴 달성이 목표지만 지속성을 희생한다.
        - 동일한 키에 여러 번의 동시 쓰기가 있다면 클라이언트에게 모두 성공으로 보고 될지라도 쓰기 중 하나만 남고 다른 쓰기는 조용히 무시된다.
        - 캐싱과 같이 손실된 쓰기를 허용하는 상황이 있다.
          손실 데이터를 허용하지 않는다면 LWW가 충돌해소에 적합하지 않는 방법이다.
        - LWW로 DB 를 안전하게 사용하는 유일한 방법은 키를 한 번만 쓰고 이후에는 불변 값으로 다루는 것이다.
- **"이전 발생 관계"와 동시성**
    - 두 개 작업이 동시에 수행됐는지 여부를 어떻게 결정할까?
      (아래 그림은 위에서 본 그림들)



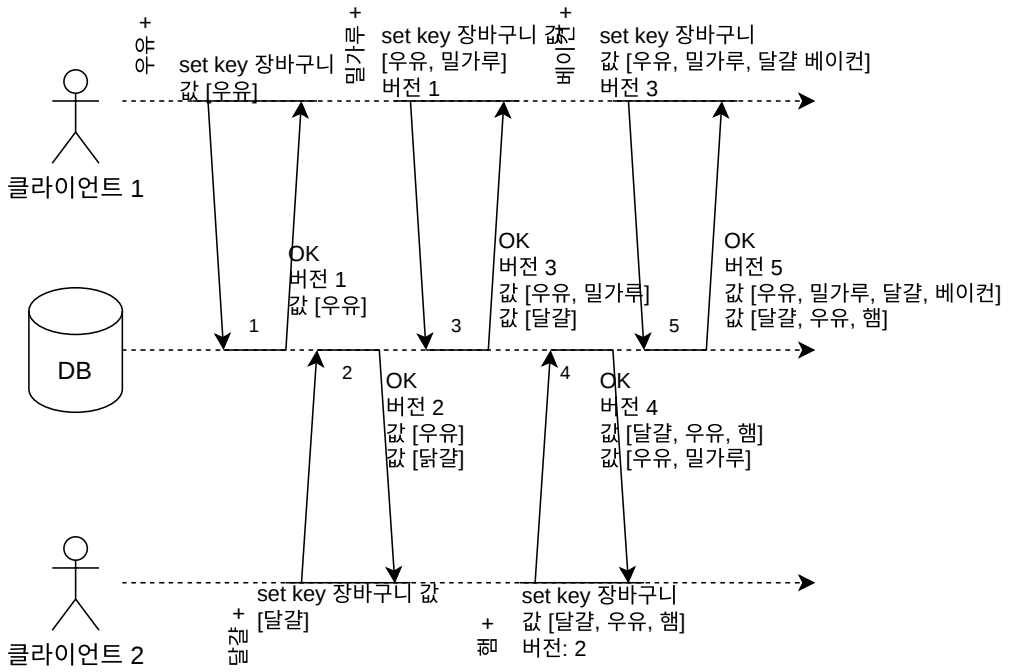
- 두 개의 쓰기는 동시에 수행되지 않았다. A 삽입이 B의 증가 이전에 발생했다. B가 증가시킨 값은 A가 삽입한 값이기 때문이다. 즉, B 작업은 A 작업 기반이기 때문에 B 작업은 나중에 발생해야 한다. 이를 B는 A에 인과성에 있다고 한다.



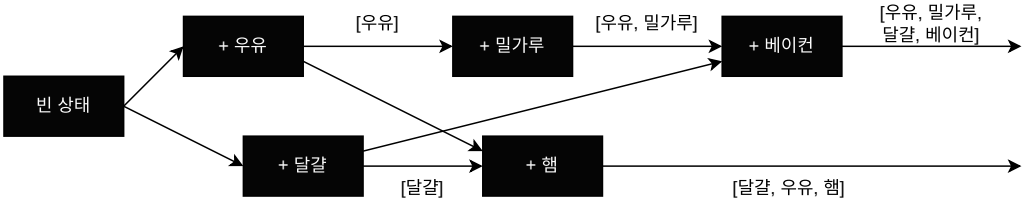
- 두 개 의 쓰기는 동시에 수행됐다. 각 클라이언트가 작업을 시작할 때 다른 클라이언트가 동일한 키에 대한 작업을 수행했는지 알지 못한다. 따라서 작업관 인과성이 없다.

  - 작업 B가 작업 A에 대해서 알거나 A에 의존적이거나 어떤 방식으로든 A를 기반으로 한다면 작업 A는 작업 B의 **이전 발생**이다. (한 작업이 다른 작업 이전에 발생했는지가 동시성의 의미를 정의하는 핵심이다.)
    작업이 다른 작업보다 먼저 발생하지 않으면 (어느 작업도 다른 작업을 모른다면)
    동시 작업이라 본다.
  - 따라서 A, B 작업이 있다면 3가지가 있다. (B 이전에 A 이거나, A 이전에 B 이거나, A, B 동시) 즉, 두 작업이 동시성인지 아닌지 알 수 있는 알고리즘이 필요하고 A, B가 동시에 발생하면 충돌을 해소해야 한다.

- **이전 발생 관계 파악하기**
  - 두 작업이 동시에 발생했는지 또는 하나가 이전에 발생했는지 여부를 결정하는 알고리즘



1. 클라이언트 1은 장바구니에 우유를 추가한다. 이것은 키에 첫 번째 쓰기이므로 서버는 저장 성공하고 버전1을 할당, 또 서버는 클라이언트에게 값을 다시 보여준다.
2. 클라이언트 2는 클라이언트 1이 현재 우유를 추가했다는 사실을 알지 못하는 상태에서 장바구니에 달걀을 추가한다. 서버는 버전 2를 할당하고 달걀과 우유를 개별 값으로 저장한다. 클라이언트에게 버전 2를 가진 두 개의 값을 반환한다
3. 클라이언트 2가 쓴 내용을 모르는 클라이언트 1은 밀가루를 장바구니에 추가한다. 현재 [우유, 밀가루] 가 있다고 판단한다. 이 값은 이전에 서버가 클라이언트 1에게 준 버전 1과 함께 서버에 전송된다. 서버는 이 버전 번호로 이전의 [우유]를 [우유, 밀가루] 쓰기로 대체하지만 [달걀]과도 동시라는 사실을 안다. 따라서 서버는 [우유, 밀가루] 에 버전 3에 할당하고 버전 1의 [우유] 값을 덮어쓴다.
4. 클라이언트2는 클라이언트 1이 밀가루를 추가했는지 모른채 햄을 장바구니에 추가하려 한다. 클라이언트 2는 지난 응답에서 서버로부터 [우유]와 [달걀]이라는 두 값을 받았기 때문에 클라이언트는 응답 값에 햄을 추가해서 새로운 값인 [달걀, 우유, 햄]으로 합친다. 이 값은 예전 버전인 2를 가지고 서버에 전송된다. 서버는 버전 2로 [달걀]을 덮어쓰지만 [우유, 밀가루]는 동시에 수행된 사실을 감지하기 때문에 [우유, 밀가루]라는 두 값은 버전3으로 남아 있고 [달걀, 우유, 햄]은 버전 4를 갖는다.
5. 마지막으로 클라이언트 1이 베이컨을 추가하려고 한다. 이전에 서버로부터 버전 3의 [우유, 밀가루]와 [달걀]을 받았으므로 여기에 베이컨을 추가해 최종 값인 [우유, 밀가루, 달걀, 베이컨]으로 합쳐서 버전 3으로 서버에 전송한다. 이 값은 [우유, 밀가루]를 덮어 쓰지만 [달걀, 우유, 햄]은 동시에 수행됐기 때문에 서버는 이 두개의 동시 수행된 값을 유지한다.



- 인과성 도표
  - 화살표는 어떤 작업이 다른 작업 이전에 발생했는지와 나중 작업이 이전에 수행된 작업을 알거나 의존했다는 사실을 나타낸다.
  - 위 예에서는 항상 다른 작업이 동시에 수행됐기 때문에 클라이언트는 서버 데이터와 동일한 최신 상태를 유지하지 못한다.
  - 그러나, 최종적으로 예전 버전의 값을 덮어쓰기 때문에 손실된 쓰기는 없다.
- 핵심
  - 서버는 버전 번호를 보고 두 작업이 동시에 수행됐는지 여부를 결정할 수 있으므로 값 자체를 해석할 필요는 없다. 따라서 값을 데이터 구조로 사용할 수 있다.
  - 알고리즘 순서
    - 서버가 모든 키에 대한 버전 번호를 유지하고 키를 기록할 때마다 버전 번호를 증가시킨다. 기록한 값은 새로운 버전 번호를 가지고 저장한다.
    - 클라이언트가 키를 읽을 때는 서버는 최신 버전뿐만 아니라 덮어쓰지 않은 모든 값을 반환한다. 클라이언트는 쓰기 전에 키를 읽어야 한다.
    - 클라이언트가 키를 기록할 때는 이전 읽기의 버전 번호를 포함해야 하고 이전 읽기에서 받은 모든 값을 함께 합쳐야 한다. (쓰기 요청의 응답은 읽기 요청과 같을 수 있다. 쓰기 요청이 현재 모든 값을 반환하기 때문)
    - 서버가 특정 버전 번호를 가진 쓰기를 받을 때 해당 버전 이하 모든 값을 (새로운 값으로 합친다는 사실을 알고 있으므로) 덮어 쓸 수 있다. 하지만 이보다 높은 버전 번호의 모든 값은 유지해야 한다. 이 값들은 유입된 쓰기와 동시에 발생되었기 때문

- **동시에 쓴 값 병합**
  - 위 알고리즘은 어떤 데이터도 자동으로 삭제되지 않음을 보장하지만 불행히도 클라이언트가 추가적으로 작업을 수행해야 한다. 여러 작업이 동시에 발생하면 클라이언트는 동시에 쓴 값을 합쳐 정리해야 한다. 이런 동시값을 **형제(sibling) 값**이라 부른다.
  - 형재 값의 병합은 다중 리더 복제에서 충돌을 해소하는 문제와 본질적으로 같다. 간단한 접근 방식으로 버전 번호나 타임스탬프 기반으로 하나의 값을 선택하는 방법(최종 쓰기 승리)가 있지만 데이터 손실이 생길 수 있다.
  - 형제를 병합하는 합리적인 접근 방식은 합집합을 취하는 것이다.
    ex) 위 예시에서는 [우유, 밀가루, 달걀, 베이컨, 햄] 으로 병합
  - 하지만, 장바구니 추가 외에 제거도 할 수 있으려면 형제의 합집합으로 올바른 결과를 얻을 수 없다. 해결법은 상품을 제거할 때 DB에 단순히 삭제가 아닌 해당 버전 번호에 표시를 남긴다. 이런 삭제 표시를 **툼스톤**이라 한다.
- **버전 벡터**

▼ 정리