트랜잭션

> 우리는 향상 **트랜잭션 없이** 코딩하는 것보다 **트랜잭션을 과용**해서 병목지점이 생기는 성능 문제를 애플리케이션 프로그래머가 처리하게 하는 게 낫다고 생각했다.
>
> by **제임스 코벳**

데이터 시스템의 문제에는?

- DB S/W & H/W Fail
- Application Fail
- Network Fail : Application to DB & DB to DB
- Concurrent Write
- Partial Write
- Client Race Condition
  - EX) Redis O(N) 수행
  - 연못 같은 디비 풀

트랜잭션이란?

애플리케이션의 몇 개의 읽기와 쓰기를 하나의 논리적 단위로 묶는 방법.

트랜잭션은 전체가 성공하거나 실패함. 실패시 안전하게 재시도할 수 있으며 부분 실패가 발생하지 않음.

트랜잭션의 사용을 통해 애플리케이션에서 어느정도의 잠재적인 오류 시나리오와 동시성 문제를 무시할 수 있음

ACID의 의미

- Actomicity : 원자성
- Consistency : 일관성

- Isolation : 격리성
- Durabillity : 지속성

깊게 들어가면 어떤 것을 기대할 수 있는지 분명하지 않음.

ACID 표준을 따르지 않는 시스템을 BASE라고 부름.

- Basically Availlable : 기본적으로 가용성 제공
- Soft state: 유연한 상태
- Eventual Consistency: 최종적 일관성

원자성

원자적 == 더 이상 작은 부분으로 쪼갤 수 없는 것

시스템은 연산을 실행하기 전이나 실행 후의 상태에만 있을 수 있으며 그 중간 상태에는 머물 수 없음.

동시성과는 관련이 없는 요소

여러 쓰기 작업을 하나로 묶어 커밋이 될 수 없으면 어보트시킴.

정규화에서의 원자성도 있음

일관성

여러 의미로 쓰임

- 복제 일관성
- 일관성 해싱
- CAP 정리에서 일관성은 선형성
    - CAP 정리에서의 선형성은? 분산 데이터베이스에 여러 쓰기가 수행 되었을 때, 그 이후에 어떤 노드를 통해 읽기를 하더라도 동일한 결과가 보장되는 것.
- ACID 에서는 좋은 상태에 있어야 함을 의미

ACID에서 일관성의 아이디어는 항상 참이어야 하는 데이터에 관한 어떤 선언(불변식, invariant)가 있이 있다는 것. 트랜잭션이 이런 불변식이 유효한 데이터베이스에서 시작하고

트랜잭션에서 실행된 모든 쓰기가 유효성을 보존한다면 불변식이 항상 만족된다고 확신할 수 있음.

그런데 이 일관성은 애플리케이션 레벨에서의 책임이며, 데이터베이스가 보장해줄 수 있음 불변식은 한정되어 있음.

ex) 입출금통장이란 테이블이 있을 때 이 테이블은 0원 미만이 될 수 없단 불변식이 있어도 이를 보장 해주는건 애플리케이션

격리성

동일한 데이터베이스 레코드에 접근하면 동시성 문제(경쟁 조건)에 직면하게 됨. 격리성은 동시에 실행되는 트랜잭션을 서로 격리하는 것을 의미.

지속성

트랜잭션이 성공적으로 커밋됐다면 하드웨어 결함이 발생하거나 데이터베이스가 죽더라도 트랜잭션에서 기록한 모든 데이터는 손실되지 않음.

완벽한 지속성은 없다 !!

이를 보장해주는 도구는? WAL, 복제

단일 객체 연산과 다중 객체 연산

단일 객체 쓰기

단일 객체 수준은 원자성과 격리성을 제공하는 것을 목표로 함.

부분적인 쓰기 상황에서 원자성을 달성하기 위해 WAL을 사용하고, 격리성은 객체에 잠금으로 접근가능한 트랜잭션의 수를 한정함.

다중 객체 트랜잭셔의 필요성

..

오류와 어보트 처리

기본적으로 트랜잭션 수행 중에 성공하면 저장하고, 도중에 실패하면 트랜잭션을 폐기한다.

하지만. 리더 없는 복제에서는 **최선을 다하는 원칙**을 기반으로 훨씬 더 많은 일을 하려고 하며 이 경우에는 이미 한 일은 취소하지 않고, 이에 대한 책임을 애플리케이션에게 넘긴다.

어보트된 트랜잭션을 재시도하는 것은 간단하고 효과적인 오류 처리 메커니즘이지만 완벽하지는 않다.

- 트랜잭션 커밋 성공을 알리는 도중 네트워크가죽으면 실패한 것으로 판단하게 된다. 이때 재시도하면 트랜잭션이 두 번 실행되게 된다.
- 오류가 과부화 때문이면 재시도는 문제를 악화시킬 수 있다.
- 일시적인 오류(교착 상태, 격리성 위반, 일시적인 네트워크 단절) 시에만 재시도가 의미가 있다.
- 여러 개의 시스템들이 반드시 함께 커밋되거나 어보트 되게 만들고 싶다면 2단계 커밋이 도움이 될 수 있다.
- 클라이언트 프로세스가 재시도 중에 죽어버리면 그 클라이언트가 데이터베이스에 쓰려고 했던 데이터가 모두 손실된다.

격리

완화된 격리 수준

동시성 버그는 타이밍에 운이 없을 때만 촉발되기 때문에 테스트로는 발견과 재현되기가 어렵다. 트랜잭션 격리를 제공함으로써 개발자에게 동시성 문제를 감추려고 했다. 격리성은 동시성이 없는 것 처럼 행동할 수 있으므로 개발자들의 부담을 줄여줘야 한다.

격리 수준을 몇 가지 살펴보고 발생할 수 있는 경쟁 조건과 발생할 수 없는 경쟁 조건을 자세히 설명한다. 이를 보고 애플리케이션의 특성에 따라 적합한 격리 수준을 선택할 수 있다.

커밋 후 읽기 Read Committed

- 커밋되지 않은 데이터를 읽거나 쓰게 되는 것을 더티 리드, 더티 쓰기라고 함.
- 데이터베이스에서 읽을 때 커밋된 데이터만 읽음(더티 리드가 없음)
- 데이터베이스에 쓸 때 커밋된 데이터만 덮어쓰게 됨(더티 쓰기가 없음)

구현 방법(1) - Row Lock

읽거나 쓰는 중인 row 를 잠금으로써 구현. 동시성이 떨어짐.

구현방법(2) - Undo Segment 를 통한 MVCC

트랜잭션 내에서 변경이 발생하면 기존 데이터는 Undo Segment에 저장하고 이를 읽음.

비반복 읽기 or 읽기 스큐 Non Repeatble Read or Read Skew

Row Lock과 Undo Segment를 통한 MVCC를 사용해도 커밋이 된 시점부터는 변경된 데이터를 읽을 수 있는 문제가 있음.

(+) Skew v.왜곡하다

스냅숏 격리 OR 반복 읽기

트랜잭션 내에서 조회되는 데이터가 동일한 것을 보장해주는 격리레벨이다. 스냅숏 격리와 반복 읽기와 비슷한 의미를 가지는데 갱신 손실 자동 감지가 되어야만 스냅숏 격리라고 할 수 있어 엄밀히는 의미상의 차이가 있다고 볼 수 있다.

커밋된 데이터를 읽게 되는 비반복 읽기와 읽기 스큐를 방지하기 위한 방법이다. 읽는 쪽에서 쓰는 쪽을 차단하지 않고, 쓰는 쪽에서 읽는 쪽은 차단하지 않는다.

서로 다른 시점에서 데이터베이스 상태를 봐야할 수도 있는데 데이터베이스가 객체의 여러 버전을 함께 유지한 다는 특성때문에 다중 버전 동시성 제어(Multi Version Concurrency Control, MVCC)이라고 한다.

스냅숏 격리시 볼 수 있는 객체는 아래 두 가지가 참이여야 한다.

- 읽기를 실행하는 트랜잭션이 시작한 시점에 읽기 대상 객체를 생성한 트랜잭션이 이미 커밋된 상태였다.

- 읽기 대상 객체가 삭제된 것으로 표시되지 않았다. 또는 삭제된 것으로 표시됐지만 읽기를 실행한 트랜잭션이 시작한 시점에 삭제 요청 트랜잭션이 커밋되지 않았다.

갱신 손실 방지, Avoiding Lost Updates

read-modify-write 패턴으로 갱신을 하는 두 개의 트랜잭션이 있을 때 갱신 손실이 발생할 수 있다. 갱신 손실과 관련하여 예로 많이 드는 것에는 카운트 증가가 있다.

최초에 카운트가 42인 상황에서 두 개의 트랜잭션이 read-modify-write 패턴으로 카운트를 1 증가시킬 경우 44가 되어야 되지만, 하나가 덮어 써지면서 43이 되는 것이다. 이러한 현상은 Read Committed와 Repeatable Read(Snapshot isolation) 모두 발생할 수 있다. 이러한 현상은 ORM 프레임워크를 쓰는 경우 많이 발생하는데, 이를 방지하려면 어떻게 해야 할까?

총 네 가지 정도의 방법이 있다.

원자적 쓰기 연산

카운터를 1 증가시키는 요청을 데이터베이스에 보낼 때 아래와 같은 SQL을 보내는 것이다.

```
UPDATE counters SET value = value + 1 WHERE key = 'foo'
```

하지만, 원자적으로 표현가능한 연산은 한정되어 있으며 비즈니스 로직에서 필요로 하는 수준의 처리는 거의 불가능하다고 볼 수 있다. 또한 원자적 연산은 배타적인 락을 통해서 구현이 된다.

명시적인 잠금

연산을 수행하기 전에 명시적으로 잠금을 하고, 완료한 이후에 반납하는 것이다. 확실하게 갱신 손실 방지가 될 수는 있으나 취득한 잠금을 반환하는 것을 잊어버리는 휴먼 에러가 발생할 수 있다.

갱신 손실 자동 감지

원자적 연산과 잠금은 read-modify-write 주기가 순차적으로 실행도디도록 강제함으로써 갱신 손실을 방지하는 방법이다. 대안으로 이들의 병렬 실행을 허용하고 트랜잭션 관리자가 갱신 손실을 발견하면 트랜잭션을 어보트 시키고 read-modify-write 주기를 재시도하도록 강제하는 방법이 있다.

이 방법의 이점은 데이터베이스가 이 확인을 스냅숏 격리와 결합해 효과적으로 수행할 수 있다는 것이다. 실제로 PostgreSQL의 Repeatable Read, Oracle의 Serializable, SQL Server의 Snapshot Isolation이 갱신 손실이 발생하면 자동으로 발견해서 문제가 되는 트랜잭션을 어보트 시킨다. 그러나 MySQL은 갱신 손실을 감지하지 못하는데 스냅숏 격리를 제공한다는 것은 이에 대한 감지 또한 가능한다는 것을 의미하기 때문에 엄밀히 말해서 MySQL은 스냅숏 격리를 제공하지 않는다고 볼 수 있다.

Compare-and-set

트랜잭션을 제공하지 않는 데이터베이스 중에는 원자적 compare-and-set 연산을 제공하는 것도 있다. 이 연산은 마지막으로 읽었을 때 값이 변하지 않은 경우에만 갱신을 허용함으로써 갱신 손실을 회피하는 것이다. 현재 값과 이전 값과 일치하지 않으면 갱신이 반영되지 않고 read-modify-write를 재시도해야 한다,

(+) JPA에서는 갱신 전에 read 시점의 엔티티 version과 write 시점에 엔티티 version이 동일한지 확인해봄으로써 compare-and-set을 구현했다.

주의해야 할 점은 데이터베이스가 오래된 스냅샷으로부터 읽는 것을 허용한다면 갱신 손실을 막지못할 수 있다,

(+) 갱신 손실 우려가 있는 조회 로직에서는 for share를 통한 낙관적인 락이 필수적일 것으로 보인다,

충돌 해소와 복제

잠금과 compare-and-set 연산은 데이터의 최신 복사본이 하나만 있다고 가정합니다.(리더 하나만 있음!)

그러나 다중 리더 또는 리더 없는 복제를 사용하는 데이터베이스는 일반적으로 여러 쓰기가 동시에 수행되고 비동기식으로 복제되는 것을 허용하므로 데이터의 최신 복사본이 하나만

있으리라고 보장할 수 없습니다,

그래서 이렇게 복제가 적용된 경우에 흔히 쓰는 방법은 쓰기가 동시에 실행될 때 한 값에 대해 여러 개의 충돌된 버전을 생성하는 것을 허용하고 사후에 애플리케이션 코드나 특별한 데이터 구조를 통해 충돌을 해소합니다.

(+) 하지만 상상이 잘 안됩니다.

많은 복제 데이터베이스는 충돌 해소 방법으로 최종 쓰기 승리(Last Write win, LWW)을 사용하고 갱신 손실이 발생하기 쉽습니다.

쓰기 스큐(Write Skew)

쓰기 작업에서 왜곡이 있는 현상을 말하며, 정확히는 무언가 읽고, 읽은 값을 기반으로 어떤 결정을 한 다음 그 결정을 데이터베이스에 쓰는 비즈니스 로직이 있을 때 쓰기를 실행할 때는 결정의 전제가 참이 아닌 현상을 말한다. 가장 쉬운 방지법으로는 직렬성 격리가 있다.

> 불변식(invariant): 어떤 객체의 유효한 상태를 규정하는 조건 혹은 그 조건을 점검하는 코드를 일컫는 전문 용어

쓰기 스큐 현상은 회의실 예약 시스템, 멀티플레이어 게임, 사용자명 회득, 이중 사용 방지 등에서 발생할 수 있다.

발생 이유

어떤 트랜잭션에서 수행한 쓰기가 다른 트랜잭션의 검색 질의 결과를 바꾸는 효과를 팬텀이라고 한다. 쓰기 스큐는 이러한 팬텀으로 인해 발생하는 데 주로 아래 패턴으로 발생한다.

1. 수행 해야하는 쓰기 작업이 있을 때, 쓰기 작업이 수행 가능한지 SELECT 절로 확인한다.
2. SELECT 결과에 따라 수행할 쓰기 작업을 판단한다.
3. 쓰기 작업 이후에 커밋한다.

3번을 수행하는 시점에 다른 트랜잭션이1번의 SELECT 절 결과에 영향을 쓰기를 할 경우 올바르지 못한 쓰기, 즉 쓰기 스큐가 발생하게 된다. 이를 방지하기 위해 FOR UPDATE 절을 사용하더라도 1번에서 확인하는 연산이 존재 여부를 기반으로 한다면 막지 못할 수 있다,

이를 어떻게 방지하면 될까? 책에서는 위 1번 단계에서 확인하는 것이 존재 여부를 기반으로 하고 있고 존재하지 않는 것이 불변식을 깨지게 하는 주요 원인이라면 충돌 구체화 (Materializing Conflict) 활용하라고 한다. 예를 들어, 회의실 예약 테이블에 회의실 예약 시간 구간 별로 예약되지 않은 row를 미리 추가하는 것이다.

충돌 구체화의 단점은 두 가지가 있다.

1. 동시성 제어 메커니즘이 데이터베이스 레이어를 넘어 애플리케이션에 영향을 줌.
2. 구체화 하기 위한 방법을 생각해내기가 어려움.

결론적으로 충돌 구체화는 최후의 수단으로 생각하기를 권장한다.

Serializable

- 위의 완화된 격리 수준은 이해하기 어렵고 데이터베이스마다 구현의 일관성이 없다
- 경쟁 조건을 감지하는 데 도움이 되는 좋은 도구가 없다
- 여러 트랜잭션이 병렬로 실행되더라도 최종 결과는 동시성 없이 한 번에 하나씩 직렬로 실행될 때와 같도록 보장되며, 모든 경쟁 조건을 막아준다. 가장 강력한 격리 수준이라고 여겨진다.

실제적인 직렬 실행

2007년 경 한 번에 하나의 트랜잭션이 직렬로 싱글 쓰레드에서 수행되는 것이 가능하다고 결론이 내려졌다. 이러한 결론의 밑바탕에는 다음과 같은 변화와 발견이 있다.

- 최근 메모리 가격의 저렴해지면서 디스크 상에 있는 모든 데이터를 메모리에 올리는 것이 가능해졌다.

- OLTP 트랜잭션은 짧고, 실행되는 읽기와 쓰기 갯수가 적음. 오해 실행되는 분석 질의는 전형적으로 읽기 전용이여서 실행루프 밖에서 실행 가능.

(+) 개인적으로 첫번째 이유가 가장 커 보입니다!

성능적으로 뛰어난 스토어드 프로시저가 관리 및 테스트가 어렵다는 점 때문에 잘 사용되지 않아왔었는데 최근에는 많이 사용되는 프로그래밍 언어(자바, 그루비, 루아)등으로 구현될 수 있게 되면서 다시 사용되어 지고 있음. 트랜잭션 코드가 스토어드 프로시저에 모두 있으면, IO 대기 없이 단일 스레드로 여러 명령어를 한 번에 수행할 수 있으면서 동시에 좋은 처리량도 얻을 수 있음

쓰기 처리량이 높은 경우 여러 CPU 코어 활용 및 노드 확장을 위해 파티셔닝을 해야 하는데, 여러 파티션에 걸쳐 조회하는 상황이 발생하면 성능이 오히려 떨어지는 상황이 발생할 수 있어 이점에서는 주의가 필요합니다.

2단계 잠금 (2PL, 2 Phase Lock)

특정 객체에 대해 LOCK을 설정하는 1단계(Expending Phase)와 LOCK을 해제 하는 2단계(Shrinking Phase)로 나뉘는 잠금이여서 2PL이라고 불린다.

2PC(2 Phase Commit)와는 완전히 다르다.

스냅숏 격리는 **읽는 쪽은 쓰는 쪽을 막지 않으며 쓰는 쪽도 읽는 쪽을 막지 않지만,** 2PL은 **쓰기 트랜잭션은 다른 쓰기 트랜잭션 뿐 아니라 읽기 트랜잭션도 진행하지 못하게 막으며, 읽기 트랜잭션은 다른 읽기 트랜잭션은 허용하지만 쓰기 트랜잭션은 진행하지 못하게 막는다!**

MySQL의 Serializable은 2PL로 구현된다. MySQL은 SERIALIZABLE 구현은 REPEATABLE-READ에 모든 SELECT 문을 SELECT FOR SHARE 문으로 인식한다. SELECT 시에는 무조건 S Lock이 걸린다.

UPDATE/DELETE 시에는 X Lock을 획득해야 한다. X Lock과 S Lock은 상호 호환되지 않으므로, 읽기는 다른 읽기를 막진 않지만 쓰기를 막고, 쓰기는 다른 쓰기도 막고 읽기도 막게 된다

Deadlock **교착 상태가 아주 많이 발생할 수 있으며 애플리케이션은 재시도해야 한다!**

2PL - 서술 잠금

데이터베이스에 아직 존재하지 않지만, 미래에 추가될 수 있는 객체에 lock을 걸기 위해 서술 잠금(predicate locking) 획득/해제 하여 쓰기 스튜를 방지할 수 있음. (회의실 예약 문제에서 시간으로 Lock Range 잡기?)

(+) MySQL에서는 Spatial Index란 공간 인덱스가 서술 잠금을 지원하지만 성능이 좋지는 않음.

2PL - 색인 범위 잠금

서술 잠금은 너무 구체적이기 때문에, 오버헤드를 줄이기 위해 index-range locking, next-key locking으로 구현한다. WHERE 조건 보다 더 넓은 범위의 인덱스를 lock 잡는 것으로 구현. Write Phantom 과 Write Skew 보호. 그러나 적절한 인덱스가 없으면 테이블 락을 잡는다.

직렬성 스냅숏 격리(SSI)

스냅숏 격리 + 직렬성 충돌 감지 시 어보트시킬 트랜잭션을 결정하는 알고리즘. **직렬성 스냅숏 격리는 완전한 직렬성을 제공하지만 스냅숏 격리에 비해 약간의 성능 손해만 존재한다!**

2단계 잠금은 비관적 동시성 제어 메커니즘이다. 상호배제(mutex)와 비슷하다.

직렬성 스냅숏 격리는 낙관적 동시성 제어 기법. 위험한 상황이 발생할 수 있을 때, 일단 진행한다. 마지막에 커밋할 때 격리 위반을 확인하고 어보트한다.

쓰는 쪽은 읽는 쪽을 막지 않고, 반대도 마찬가지이며, 읽기 전용 질의는 어떤 잠금도 없이 일관된 스냅숏 위에서 실행된다.

어보트 비율이 성능 영향을 준다. 트랜잭션 동작을 상세하게 추적하면 정확해지지만 기록 오버헤드가 심해지며, 오랜 시간동안 데이터를 읽고 쓰는 트랜잭션은 충돌 후 어보트의 가능성이 높아짐. 읽기+쓰기 트랜잭션이 짧기를 요구한다.

쓰기 스큐처럼 질의 결과(전제 조건)와 쓰기 작업 사이 인과적 의존성이 있을 때, 전제 조건이 최신 결과가 아니면, 이를 감지해서 트랜잭션을 어보트시켜서 직렬성 격리를 제공한다.

오래된(stale) MVCC 객체 버전을 읽었는지 감지하기 (읽기 전에 커밋되지 않은 쓰기가 발생했음) → 일관된 스냅숏에서 읽을 때에는 무시됐던 쓰기가 지금은 영향이 있고 트랜잭션 43의 전제가 더 이상 참이 아님.

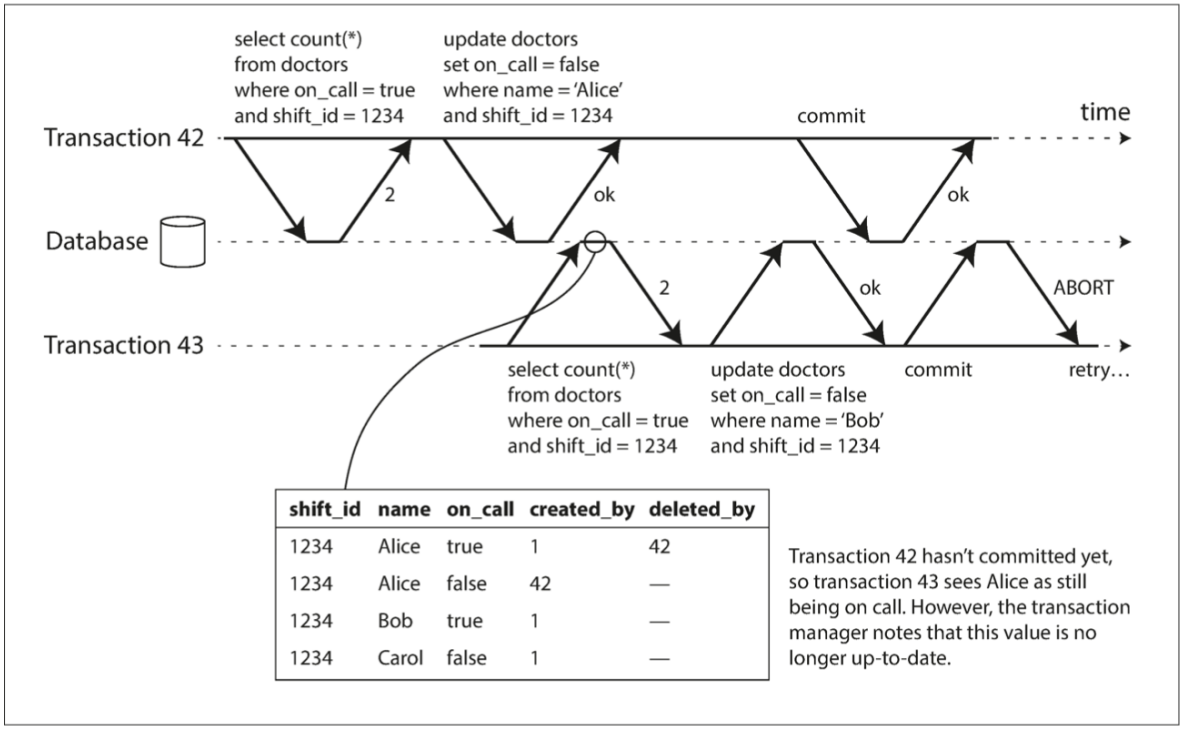

Figure 7-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

과거의 읽기에 영향을 미치는 쓰기 감지하기 (읽은 뒤에 쓰기가 실행됨) → 다른 트랜잭션을 차단하지 않음. 트랜잭션이 데이터베이스에 쓸 때 영향받는 데이터를 최근에 읽은 트랜잭션이 있는지 확인한다. 쓰기 잠금을 거는 것과 비슷하지만 커밋될 때까지 차단하지 않는다.

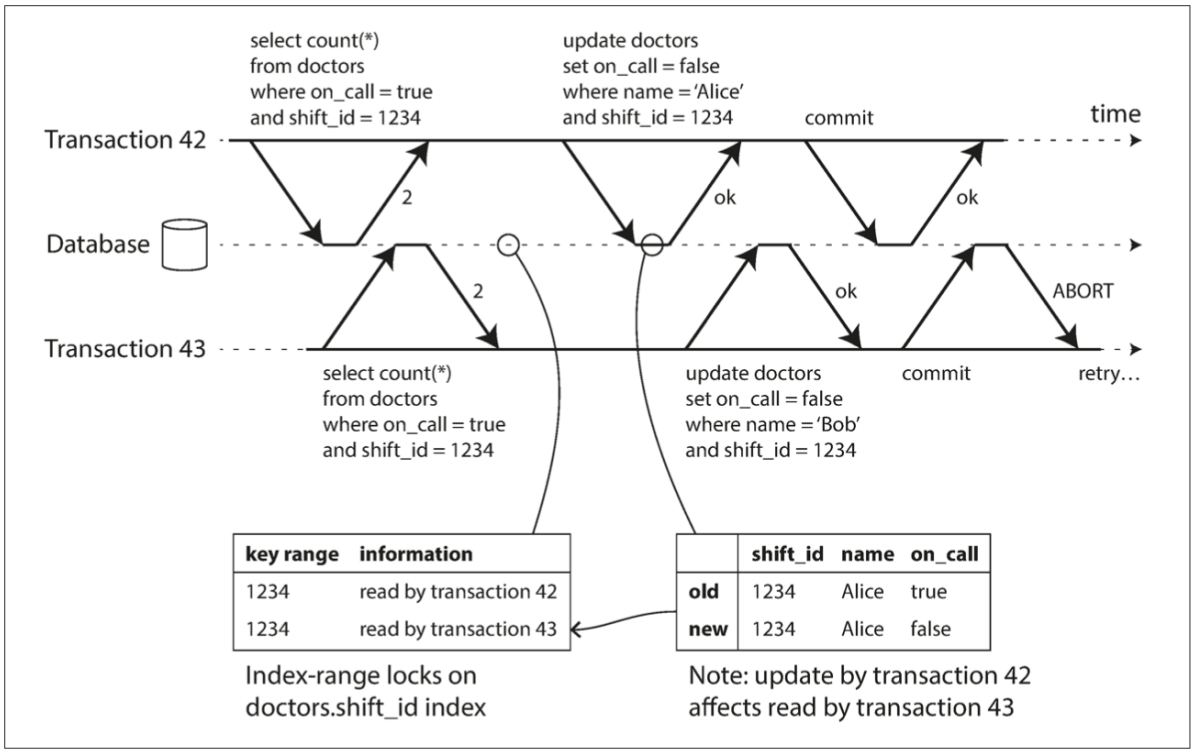

Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

단일 CPU 코어의 처리량에 제한되지 않음.

어보트 비율은 SSI의 전체적인 성능에 큰 영향을 미친다. 이를테면 오랜 시간 동안 데이터를 읽고 쓰는 트랜잭션은 충돌이 나고 어보트되기 쉬워서, SSI는 읽기 쓰기 트랜잭션이 상당히 짧기를 요구한다 - 오래 실행되는 읽기 전용 트랜잭션은 괜찮다. 2PL이나 싱글쓰레드 실행보다는 덜 민감하다.