



Transaction

⚙ 상태	진행 중
📁 분류	Database
📅 주차	2주차



목차

트랜잭션

트랜잭션이란?

트랜잭션의 예시

트랜잭션의 장점

트랜잭션의 특징

원자성

일관성

독립성

지속성

트랜잭션 상태

활성

작업 성공시

작업 실패시

Lock

Lock의 종류

공유 락

배타 락

블로킹(Blocking)

블로킹을 방지하기 위한 주의 사항

DeadLock

트랜잭션 격리 수준

READ UNCOMMITTED

READ COMMITED

READ COMMITED 예시1

REPEATABLE READ

REPEATABLE READ 예시1

REPEATABLE READ 예시2

REPEATABLE READ 예시3

SERIALIZABLE

MySQL의 트랜잭션

MySQL에서의 트랜잭션이란?

수동 트랜잭션

Rollback이 적용되지 않는 경우

트랜잭션

트랜잭션이란?

- 데이터베이스의 상태를 변화시키기 위해서 수행하는 **작업의 단위**
- 수행되어야 할 일련의 연산 모음
 - 데이터베이스의 상태를 변화 = 아래의 SQL을 이용해 데이터베이스에 접근함을 의미
 - SELECT
 - INSERT
 - DELETE
 - UPDATE

트랜잭션의 예시

- A는 매달 부모님한테 생활비를 송금받는다. 어느 날, 부모님이 A에게 생활비를 송금해 주기 위해 ATM을 이용했고 A의 계좌에 생활비를 송금했다.
- 그러나 모종의 이유로 인하여 부모님의 계좌에선 생활비가 차감되었는데, A의 계좌에는 생활비가 입금되지 않았다.
- 위와 같이 인출에는 성공했는데 입금에 실패하면 치명적인 결과가 발생한다
 - 따라서, 위 두 과정은 동시에 성공하던가 혹은 동시에 실패해야 한다
 - 위 과정을 동시에 묶는 방법이 바로 트랜잭션이다
 - 따라서 트랜잭션(작업단위)은 SQL 한 문장이 아니라, 많은 SQL 명령문들을 사람의 기준에 따라 묶는 것을 의미함

트랜잭션의 장점

- 데이터베이스와 어플리케이션 간 데이터 거래에서 안전성을 확보
 - 작업을 수행하는 도중 오류가 발생하면 결과를 반영하는 것이 아니라 모든 작업을 원상태로 복구한다
 - 모든 과정이 성공하였을 때에만 그 결과를 반영

트랜잭션의 특징

- 원자성 (Atomicity)
- 일관성 (Consistency)
- 독립성 (Isolation)

- 지속성 (Durability)

원자성

- 트랜잭션이 데이터베이스에 모두 반영되던가, 아예 반영되지 않아야 한다
 - 만약 트랜잭션 단위로 데이터가 처리되지 않으면, 시스템 이해가 어렵고 오류가 발생했을 때 원인을 찾기가 매우 힘들어진다

일관성

- 트랜잭션의 작업 처리 결과가 항상 일관성이 있어야 한다
 - 트랜잭션이 진행되는 동안 데이터베이스가 변경 되어도 처음 트랜잭션을 진행하기 위해 참조한 데이터베이스를 바탕으로 진행된다
 - 따라서 사용자는 일관성 있는 데이터를 볼 수 있다

독립성

- 어떤 트랜잭션도 다른 트랜잭션의 연산 도중에 끼어들 수 없다
 - 하나의 트랜잭션이 완료될 때까지, 다른 트랜잭션이 특정 트랜잭션의 결과를 참조할 수 없다
- 격리성은 동시성과 관련된 성능 이슈로 인해 트랜잭션 격리 수준을 선택할 수 있다
 - 여러 트랜잭션이 동시에 처리될 때, 특정 트랜잭션이 다른 트랜잭션에서 변경하거나 조회하는 데이터를 볼 수 있게 허용할지의 여부를 결정
 - 고립 수준이 낮은 순서대로 다음과 같다
 - READ UNCOMMITTED (커밋되지 않은 읽기)
 - READ COMMITTED (커밋된 읽기) - 일반적으로 많이 사용
 - REPEATABLE READ (반복 가능한 읽기)
 - SERIALIZABLE (직렬화 가능)

지속성

- 트랜잭션이 성공적으로 완료되었을 경우, 결과는 영구적으로 데이터베이스에 반영되어야 한다

트랜잭션 상태

- 트랜잭션의 연산과정 단계를 도식화하여 각각을 아래와 같이 정리할 수 있다



활성

- 트랜잭션이 정상적으로 실행중인 상태
 - 해당 트랜잭션의 상태는 활성(Active) 상태가 된다
 - 설계자가 설계한 대로 연산들이 정상적으로 실행중인 상태를 의미

작업 성공시

- 부분완료 (Partially Committed)
 - 트랜잭션이 마지막까지 실행되었지만, Commit 연산이 실행되기 직전의 상태
- 완료 (Committed)
 - 트랜잭션이 성공적으로 종료되어 Commit 연산을 실행한 이후의 상태
- 부분완료 상태가 존재하는 이유
 - 설계된 작업대로 작업이 성공하였다고 무조건 반영하는 것이 아닌, 설계자의 최종 승인이 있을 때 까지 실제 데이터베이스에 작업 내용을 반영하지 않고 기다리는 것
 - 설계자가 최종 반영을 승인(Commit)하면 트랜잭션이 종료

작업 실패시

- 실패 (Failed)
 - 트랜잭션 실행에 오류가 발생해 중단된 상태
- 철회 (Aborted)
 - 트랜잭션이 비정상적으로 종료되어 Rollback 연산을 수행한 상태



Commit, Rollback이란?

Commit

- 하나의 트랜잭션이 성공적으로 끝났고, 데이터베이스가 일관성있는 상태에 있을 때 하나의 트랜잭션이 끝났다는 것을 알려주기 위해 사용하는 연산
- 수행했던 트랜잭션이 로그에 저장된다

Rollback

- 하나의 트랜잭션 처리가 비정상적으로 종료되어 원자성이 깨진 경우, 트랜잭션을 처음부터 다시 시작하거나 트랜잭션의 부분적으로 연산된 결과를 다시 취소 시킨다

Lock

- 트랜잭션 처리의 순차성을 보장하기 위한 방법
- DBMS마다 Lock을 구현하는 방식과 세부적인 방법이 다르므로 DB를 이용하기 전 Lock에 대한 이해가 필요

Lock의 종류

- 공유 락(Shared Lock)과 배타 락(Exclusive Lock)이 존재
 - 공유 락은 Read Lock이라고도 불리며, 배타 락은 Write Lock이라고도 불린다

공유 락

- 데이터를 읽을 때 사용되는 Lock
- 공유 락이 걸린 데이터는 읽기 연산만 실행이 가능하고 쓰기 연산은 실행이 불가능하다
- 공유 락이 걸린 데이터에 대해서 다른 트랜잭션도 동일하게 공유 락을 획득할 수 있다
 - 그렇지만 배타 락은 획득할 수 없다
- MySQL에서는 SELECT FOR SHARE라는 Query를 이용해 공유락을 사용할 수 있다

배타 락

- 데이터를 변경할 때 사용되는 Lock
- 트랜잭션이 완료될 때까지 유지
- 배타 락이 걸린 데이터는 락이 해제될 때까지 다른 트랜잭션에서 해당 리소스에 접근할 수 없다
 - 배타 락을 획득한 트랜잭션은 해당 데이터에 대해 독점권을 읽는다

블로킹(Blocking)

- Lock간 경합이 발생하여 특정 트랜잭션이 작업을 진행하지 못하고 멈춰선 상태
- 블로킹을 해소하기 위해서는 이전 트랜잭션이 Commit 혹은 Rollback 되어야 함
- 따라서 Blocking을 최소화 할 필요가 있다



블로킹을 방지하기 위한 주의 사항

- 한 트랜잭션의 길이를 너무 길게 하지 않는다
- 같은 데이터를 갱신하는 트랜잭션이 동시에 수행되지 않도록 설계한다
- 트랜잭션 격리성 수준을 불필요하게 상향 조정하지 않는다
- Query를 오랜 시간 잡아두지 않도록 적절한 튜닝을 진행한다

DeadLock

- 두 트랜잭션이 각각 Lock을 설정하고 서로의 Lock이 걸린 데이터에 접근해 값을 얻어 오려고 하는 상황
- 양쪽 트랜잭션 모두 영원히 처리가 되지 않는다



트랜잭션 격리 수준

READ UNCOMMITTED

- 커밋되지 않은 데이터도 접근할 수 있다
- 다른 트랜잭션의 작업이 Commit 또는 Rollback되지 않아도 즉시 보이게 된다



- 위와 같이 트랜잭션의 작업이 완료되지 않았는데도 다른 트랜잭션에서 볼 수 있는 부정합 문제를 Dirty Read라고 한다
- 위의 경우에서 사용자 A가 Commit이 아닌 Rollback을 수행한다면?



- 사용자 B의 트랜잭션은 id = 51인 데이터를 처리하고 있었는데, 다시 데이터를 조회하니 결과가 존재하지 않는다
 - 따라서 이러한 Dirty Read 상황은 시스템에 상당한 버그를 초래
- READ UNCOMMITTED는 RDBMS 표준에서 인정하지 않을 정도로 정합성에 문제가 많은 격리 수준
 - 최소한 READ COMMITED 이상의 격리 수준을 사용해야

READ COMMITED

- 커밋된 데이터만 조회할 수 있음
- READ COMMITED에서는 Phantom Read에 더해 Non-Repeatable Read(반복 읽기 불가능) 문제까지 발생

READ COMMITED 예시1

- 사용자 A가 트랜잭션을 시작하여 어떤 데이터를 변경하고 아직 Commit은 하지 않은 상태
- 테이블은 먼저 갱신되고, UNDO 로그로 변경 전의 데이터가 백업



- 이 때 사용자 B가 데이터를 조회하려고 하면, READ COMMITED에서는 커밋된 데이터만 조회할 수 있으므로 UNDO 로그에서 변경 전의 데이터를 찾아서 반환하게 된다
 - 사용자 A가 트랜잭션을 커밋해야지만 다른 트랜잭션에서도 새롭게 변경된 값을 참조할 수 있음



- 그렇지만 READ COMMITTED는 Non-Repeatable Read 문제가 발생할 수 있음
- 만약 사용자 B가 트랜잭션을 시작하고 name = "Minkyu"인 레코드를 조회
 - 해당 조건을 만족하는 레코드는 아직 존재하지 않으므로 아무것도 반환되지 않음
- 그러다가 사용자 A가 UPDATE문을 이용해 해당 조건을 만족하는 레코드를 생성한 후 Commit까지 완료
 - 이 때 사용자 B가 다시 동일한 조건으로 레코드를 조회한다면?



- 위와 같이 다른 트랜잭션의 커밋 여부에 따라 조회 결과가 변경
- 이러한 데이터 부정합 문제를 Non-Repeatable Read(반복 읽기 불가능)라고 함

REPEATABLE READ

- 일반적인 RDBMS는 변경 전의 레코드를 UNDO 공간에 백업해둔다
 - 따라서 변경 전, 후 데이터가 모두 존재하므로 동일한 레코드에 대해 여러 버전의 데이터가 존재하게 됨
 - 이를 MVCC(다중 버전 동시성 제어)라고 부른다
 - MVCC를 이용하면 트랜잭션이 롤백된 경우 데이터를 복원할 수 있을 뿐 아니라 서로 다른 트랜잭션 간에 접근할 수 있는 데이터를 세밀하게 제어할 수 있음
 - 각각의 트랜잭션은 순차 증가하는 고유한 트랜잭션 번호가 존재하며, 백업 레코드에는 어느 트랜잭션에 의해 백업되었는지 트랜잭션 번호를 함께 저장
 - 해당 데이터가 불필요하다고 판단되는 시점에 삭제
- MVCC는 한 트랜잭션 내에서 동일한 결과를 보장하지만, 새로운 레코드가 추가되는 경우 데이터 부정합이 생길 수 있다

REPEATABLE READ 예시1

- 트랜잭션을 시작하고 사용자 B가 id = 50인 데이터를 조회하면 1건 조회되는 상황이라고 가정, 아직 트랜잭션은 종료되지 않았다
- 이 때 다른 사용자 A의 트랜잭션에서 id = 50인 데이터를 갱신
 - MVCC를 통해 기존 데이터는 변경되지만 백업된 데이터가 UNDO 로그에 남는다



- 이전에 사용자 B가 데이터를 조회했던 트랜잭션은 아직 종료되지 않은 상황에서, 사용자 B가 다시 한번 동일한 SELECT문을 실행한다면?
 - 사용자 B의 트랜잭션(10)은 사용자 A의 트랜잭션(12)이 시작되기 전에 이미 시작된 상태이다
 - REPEATABLE READ는 트랜잭션 번호를 참고하여 자신보다 먼저 실행된 트랜잭션의 데이터만을 참고한다
 - 따라서 사용자 B는 기존과 동일한 조회 결과를 얻게 된다



- REPEATABLE READ는 새로운 레코드의 추가를 막지 않기 때문에 Phantom Read가 발생할 수 있다
 - Phantom Read : SELECT문을 이용해 조회할 때 트랜잭션이 끝나기 전에 다른 트랜잭션에 의해 추가된 레코드가 발견되는 경우
- MVCC에서 자신보다 나중에 실행된 트랜잭션이 추가한 레코드는 무시되기 때문에 일반적으로 Phantom Read는 발생하지 않는다
- 그렇지만, Lock이 사용될 때 Phantom Read가 발생할 수 있음

REPEATABLE READ 예시2

- 사용자 B가 먼저 데이터를 조회하는데, 이번에는 SELECT FOR UPDATE(배타 락)을 이용해 쓰기 잠금을 걸었다
- 사용자 A가 새로운 데이터를 INSERT 하는 상황, 일반적인 DBMS에서는 갭락이 존재하지 않으므로 id = 50인 레코드만 잠금이 걸려 사용자 A의 요청은 즉시 실행된다



- 이 때 사용자가 동일한 SELECT FOR UPDATE 쿼리로 다시 한번 데이터를 조회하면 이번엔 2건의 데이터가 조회된다
 - 이렇게 다른 트랜잭션에서 수행한 작업에 의해 레코드가 보였다 안보였다 하는 현상을 Phantom Read라고 한다
 - 다른 트랜잭션에서 새로운 레코드를 추가하거나 삭제하는 경우 발생할 수 있음
- 아래 경우를 MVCC를 통해 해결할 수 있을 것 같지만 SELECT FOR UPDATE는 데이터 조회가 UNDO 로그가 아닌 테이블에서 수행되기 때문에 그럴 수 없다
 - UNDO 로그는 append only 형태이므로 잠금 장치가 없어서 UNDO 로그를 잠그는 것은 불가능하다



- 그렇지만 일반적인 RDBMS와 달리 MySQL에는 **갭 락**이 존재하기 때문에 위 상황이 발생하지 않는다
 - 사용자 B가 SELECT문 조건으로 $id \geq 50$ 을 걸었기 때문에, MySQL은 id가 50인 레코드에는 레코드 락, 50보다 큰 범위에는 넥스트 키 락을 건다



REPEATABLE READ 예시3

- MySQL의 REPEATABLE READ에서 Phantom Read가 발생하는 경우는 다음과 같다
- 사용자 B는 트랜잭션을 시작하고 잠금없는 SELECT문으로 데이터를 조회
- 사용자 A는 INSERT문을 이용해 데이터를 추가, 잠금이 없으므로 바로 COMMIT
- 이후 사용자 B가 SELECT FOR UPDATE문으로 데이터를 조회
 - 배타 락을 이용해 데이터를 조회하게 되면 UNDO 로그가 아닌 테이블을 조회하게 되므로 Phantom Read가 발생한다



SERIALIZABLE

- 가장 엄격한 격리 수준, 트랜잭션을 순차적으로 진행시킨다
- 여러 트랜잭션이 동일한 레코드에 동시 접근할 수 없으므로 어떠한 데이터 부정합 문제도 발생하지 않는다
- 그렇지만 동시 처리 성능이 매우 떨어진다
- 일반적인 SELECT 작업은 아무런 레코드 잠금 없이 실행된다
- 그렇지만 SERIALIZABLE 격리 수준에서는 순수한 SELECT 작업에서도 대상 레코드에 공유 락을 건다
 - 따라서 다른 트랜잭션에서 공유 락이 걸린 데이터에 대해서 추가, 삭제, 수정을 할 수 없음

MySQL의 트랜잭션

MySQL에서의 트랜잭션이란?

- MySQL에서 트랜잭션은 데이터베이스의 상태를 바꾸는 일종의 작업 단위
 - 우리가 MySQL에서 입력하는 모든 쿼리들은 하나의 트랜잭션이라고 할 수 있다
 - INSERT, DELETE, UPDATE 등의 SQL 명령어를 통해 데이터의 상태를 바꿀 때마다 내부에선 자동적으로 Commit을 실행해 변경된 내용을 데이터베이스에 반영함
- 즉 여태까지 입력한 명령어들은 MySQL에서 자동 Commit을 통해 쿼리 입력과 동시에 처리하여 데이터베이스에 반영하게 한 것임

수동 트랜잭션

- 기본적으로 MySQL은 자동 Commit이 설정되어 있다
 - 세미콜론을 한 번 찍을 때마다 자동으로 Commit이 된다
- 아래 명령어를 통해 기본 Commit 방식을 변경할 수 있다

```
-- AUTO COMMIT OFF
SET AUTOCOMMIT = 0;

-- AUTO COMMIT ON
SET AUTOCOMMIT = 1;
```

```
-- AUTO QUERY 설정 확인
SELECT @@autocommit;
```

- 아래 명령어를 통해 트랜잭션을 실행할 수 있다
 - 이 때 Commit 방식은 수동 Commit으로 바뀐다

```
START TRANSACTION;
```

- 수동 트랜잭션을 시작한 후 INSERT와 DELETE를 해보는 명령어

```
START TRANSACTION; -- 트랜잭션 시작

SELECT * FROM members; -- 초기 상태 보여주기
INSERT INTO members VALUES(1, "메롱", "쌔터디"); -- 데이터 수정
SELECT * FROM members; -- 수정 상태 보여주기

COMMIT -- 트랜잭션을 DB에 적용

SELECT * FROM members; -- 적용된 결과 조회
```

- Rollback 명령을 실행하면 트랜잭션이 시작되기 이전 상태로 되돌아간다
 - 즉, 마지막 Commit을 완료한 시점으로 돌아간다

```
START TRANSACTION; -- 트랜잭션 시작

SELECT * FROM members; -- 초기 상태 보여주기
INSERT INTO members VALUES(1, "메롱", "쌔터디"); -- 데이터 수정
SELECT * FROM members; -- 수정 상태 보여주기

ROLLBACK; -- 트랜잭션을 취소하고 START TRANSACTION 실행 전 상태로 돌아감

SELECT * FROM members; -- 적용된 결과 조회
```

Rollback이 적용되지 않는 경우

- DDL문은 rollback이 적용되지 않는다

- ex) CREATE, DROP, ALTER, RENAME, TRUNCATE