# 05. 트랜잭션과 잠금



## Intro

트랜잭션 : 작업의 **완전성** 보장

- 논리적인 작업 셋을 모두 완벽하게 처리
- 원상태 복구, Partial update 방지

*잠금(Lock) vs 트랜잭션*

- 잠금 : **동시성**을 **제어**하기 위한 기능
- 트랜잭션 : 데이터의 **정합성**을 보장하기 위한 기능

✋ 여러 커넥션에서 *동시에 동일한 자원*을 *요청*할 경우 (예측 가능한 레코드를 위해)

- 잠금 : **순서대로 한 시점**에는 **하나의 커넥션**만 변경할 수 있게 함
- 트랜잭션(격리수준) : 하나 또는 그 이상 트랜잭션 내에서 **작업 내용**을 **어떻게 공유하고 차단할 것인지 결정하는 레벨**

# 5.1. 트랜잭션

`InnoDB` 스토리지엔진 : 트랜잭션 제공

`MEMORY / MyISAM` 스토리지엔진 : 제공 X

## 5.1.1. MySQL에서의 트랜잭션

트랜잭션은 꼭 여러 개의 변경 작업을 수행하는 쿼리가 조합됐을 때만 의미 있는건 아님

→ 논리적인 작업 셋에 **몇 개의 쿼리가 있든 다음을 보장**해주는 것 !!

- (COMMIT 실행 시) 작업 셋 자체를 100% 적용
- (ROLLBACK / 또는 ROLLBACK 시키는 오류 발생 시) 아무것도 적용 X

> 🖊️ *예제*

01 테스트용 테이블 생성 및 1건의 레코드 저장

```
mysql> CREATE TABLE tab_myisam ( fdpk INT NOT NULL, PRIMARY KEY (fdpk) ) ENGINE=MyISAM
mysql> INSERT INTO tab_myisam (fdpk) VALUES (3);

mysql> CREATE TABLE tab_innodb ( fdpk INT NOT NULL, PRIMARY KEY (fdpk) ) ENGINE=INNODB
mysql> INSERT INTO tab_innodb (fdpk) VALUES (3);
```

02 AUTO-COMMIT 모드에서 실행

```
-- // AUTO-COMMIT 활성화
mysql> SET autocommit=ON;

mysql> INSERT INTO tab_myisam (fdpk) VALUES (1),(2),(3);
mysql> INSERT INTO tab_innodb (fdpk) VALUES (1),(2),(3);
```

03 결과

- 두 INSERT 문장 모두 **PK 중복 오류**로 쿼리 실패

```
mysql> INSERT INTO tab_myisam (fdpk) VALUES (1),(2),(3);
ERROR 1062 (23000): Duplicate entry '3' for key 'PRIMARY'

mysql> INSERT INTO tab_innodb (fdpk) VALUES (1),(2),(3);
ERROR 1062 (23000): Duplicate entry '3' for key 'PRIMARY'
```

- `MyISAM` : **오류가 발생했음에도** ‘1’과 ‘2’는 INSERT 된 상태로 남아 있음
  - INSERT 문장 실행하면서 차례로 ‘1’, ‘2’ 저장

    → ‘3’을 저장하려는 순간 중복 키 오류 발생 (이미 ‘3’ 존재하기 때문)
  - **이미 INSERT된 ‘1’, ‘2’를 그대로 두고** *쿼리 실행 종료*해버림 (**=Partial update**)

    데이터의 정합성 안 맞춰짐..
- `InnoDB` : 쿼리 중 일부라도 오류가 발생하면 전체를 원상태로 만든다는 트랜잭션의 원칙대로 **INSERT 문장 실행 전의 상태로 그대로 복구**

```
mysql> SELECT * FROM tab_myisam;
+----+
|fdpk|
+----+
|   1|
|   2|
|   3|
+----+
mysql> SELECT * FROM tab_innodb;
+----+
|fdpk|
+----+
|   3|
+----+
```

- **Partial update** 발생시, 실패한 쿼리로 인해 남은 레코드를 다시 삭제하는 **재처리 작업** 필요

➡️ 트랜잭션은 애플리케이션 개발에서 고민해야 할 문제를 줄여주는 아주 필수적인 DBMS의 기능

```
-- // MyISAM : 복잡한 재처리 작업 😑
INSERT INTO tab_a ..;
IF(_is_insert1_succeed){
  INSERT INTO tab_b;
  IF(_is_insert2_succeed){
    //처리 완료
  }ELSE{
    DELETE FROM tab_a WHERE ...;
    IF(_is_delete_succeed){
      //처리 실패 및 tab_a, tab_b 모두 원상 복구 완료
    }ELSE{
      //해결 불가능한 심각한 상황 발생
      //이제 어떻게 해야 하나?
      //tab_b에 INSERT는 안되고, 하지만 tab_a에는 INSERT돼 버렸는데, 삭제는 안 되고..
    }
  }
}
```

```
-- // InnoDB : 깔끔! 😊
try{
  START TRANSACTION;
  INSERT INTO tab_a ...;
  INSERT INTO tab_b ...;
  COMMIT;
} catch(exception) {
  ROLLBACK;
}
```

### 5.1.2. 주의사항

**트랜잭션** 또한, DBMS 커넥션과 똑같이, **꼭 필요한 최소의 코드**에만 **적용**하는 것이 좋음

➡️ 프로그램 코드에서 트랜잭션의 범위를 최소화하자!

> 🖊️ *예시*

게시물 작성 → 저장 버튼 클릭

1) 처리 시작
⇒ DB 커넥션 생성
⇒ 트랜잭션 시작 ▶️
2) 사용자의 로그인 여부 확인
3) 사용자의 글쓰기 내용의 오류 여부 확인
4) 첨부로 업로드된 파일 확인 및 저장

5) 사용자의 입력 내용을 DBMS에 저장
6) 첨부 파일 정보를 DBMS에 저장

7) 저장된 내용 또는 기타 정보를 DBMS에서 조회
**8) 게시물 등록에 대한 알림 메일 발송**

9) 알림 메일 발송이력을 DBMS에 저장
<= 트랜잭션 종료(COMMIT) ⏹️
<= DB 커넥션 반납
10) 처리 완료

1) 처리 시작

2) 사용자의 로그인 여부 확인
3) 사용자의 글쓰기 내용의 오류 여부 확인
4) 첨부로 업로드된 파일 확인 및 저장
⇒ DB 커넥션 생성(도는 커넥션 풀에서 가져오기)
⇒ 트랜잭션 시작 ▶️
5) 사용자의 입력 내용을 DBMS에 저장
6) 첨부 파일 정보를 DBMS에 저장
<= 트랜잭션 종료(COMMIT) ⏹️
7) 저장된 내용 또는 기타 정보를 DBMS에서 조회
**8) 게시물 등록에 대한 알림 메일 발송**
⇒ 트랜잭션 시작 ▶️
9) 알림 메일 발송이력을 DBMS에 저장
<= 트랜잭션 종료(COMMIT) ⏹️
<= DB 커넥션 종료(또는 커넥션 풀에 반납)
10) 처리 완료

- 입력정보 저장은 하나의 트랜잭션으로 묶기
- 데이터 단순 조회는 트랜잭션에서 제외
- 네트워크 작업이 있는 경우 반드시 트랜잭션에서 배제 : 메일서버와 통신할 수 없는 상황 발생시 DBMS 서버가 높은 부하 상태로 빠지거나 위험 상태로 빠짐

## 5.2. MySQL 엔진의 잠금

잠금은 크게 *스토리지 엔진 레벨*과 *MySQL 엔진 레벨*로 나뉨

- **MySQL 엔진** 레벨의 잠금 : **모든 스토리지 엔진**에 영향
- 스토리지 엔진 레벨의 잠금 : 스토리지 엔진 간 상호 영향 X

** MySQL 엔진 : MySQL 서버에서 스토리지 엔진을 제외한 나머지 부분

### 5.2.1. 글로벌 락

- 획득 명령 : `FLUSH TABLES WITH READ LOCK`
  - 해당 명령 실행과 동시에 MySQL 서버에 존재하는 모든 테이블을 닫고 잠금
- MySQL에서 제공하는 잠금 중 **가장 범위가 큼 (MySQL 서버 전체)**

  → 즉, 작업 대상 테이블과 DB가 **달라도 동일하게 영향**을 미침.
- 한 세션에서 글로벌 락을 획득 후 다른 세션에서 SELECT를 제외한 대부분의 DDL, DML 문자을 실행할 경우, 글로벌 락 **해제시 까지 대기** 상태로 남음
- 여러 DB에 존재하는 MyISAM 또는 MEMORY 테이블에 대해 `mysqldump`로 **일관된 백업**을 받아야 할 때 사용

⚠️ 주의 : 장시간 실행되는 쿼리와 명령이 최악의 케이스로 실행되면 모든 테이블에 대한 쿼리가 아주 오랜 시간 동안 실행되지 않고 기다려야할 수 있음. →**웹 서비스용으로 사용되는 서버에서는 가급적 사용X**

### 5.2.2. 테이블 락

- 목적 : 테이블 **데이터 동기화**
- **개별 테이블 단위**로 설정되는 잠금
- 명시적/묵시적으로 특정 테이블의 락 획득 가능

#### 명시적

- 획득 명령 : `LOCK TABLES table_name [ READ | WRITE]`
- 반납 명령 : `UNLOCK TABLES`
- MyISAM, InnoDB 스토리지 엔진을 사용하는 테이블 모두에 적용 가능
- **온라인 작업**에 상당한 영향 → 애플리케이션에서 거의 사용X

#### 묵시적

- MyISAM, MEMORY 테이블에 데이터를 **변경하는 쿼리를 실행**하면 발생

→ 데이터가 변경되는 쿼리가 실행되는 동안 자동 획득, 쿼리가 완료된 후 자동 해제

- InnoDB의 경우 테이블 락이 설정되지만 스토리지 엔진 차원에서 레코드 기반 잠금 제공

→ 대부분의 DML에서는 무시, **DDL**의 경우에만 **영향** 줌

### 5.2.3. 네임드 락

- 테이블 / 레코드 / AUTO_INCREMENT 같은 데이터베이스 객체가 대상 X

  **사용자가 지정한 문자열**(String)에 대해 획득하고 반납하는 잠금
- `GET_LOCK()` 함수를 이용해 임의의 문자열에 대해 잠금 설정

> *사용 예시*

1. **여러 클라이언트**의 **상호 동기화** 처리
   - 데이터베이스 서버 1대에 5대의 웹서버가 접속하는 서비스
   - 5대의 웹서버가 어떤 정보를 동기화해야 하는 요건
2. **복잡한 요건**으로 **레코드를 변경**하는 트랜잭션:
   - 배치 프로그램 등 한꺼번에 많은 레코드를 변경하는 쿼리 → 자주 데드락의 원인이 됨
   - 이때, **동일 데이터를 변경**하거나 **참조하는 프로그램끼리 분류**하여 네임드 락을 걸고 쿼리를 실행하면 간단히 해결 가능!

** MySQL 8.0 버전부터 네임드 락을 중첩해서 사용 가능, 현재 세션에서 획득한 네임드 락을 한 번에 해제도 가능

```
SELECT GET_LOCK('mylock_1',10);
-- // mylock_1에 대한 작업 실행
SELECT GET_LOCK('mylock_2',10);
-- // mylock_1과 mylock_2에 대한 작업 실행

-- // 락 해제
SELECT RELEASE_LOCK('mylock_2');
SELECT RELEASE_LOCK('mylock_1');

-- // mylock_1과 mylock_2 동시에 모두 해제
SELECT RELEASE_ALL_LOCKS();
```

### 5.2.4. 메타데이터 락

- 데이터베이스 객체(테이블, 뷰 등)의 이름이나 **구조를 변경**하는 경우에 획득하는 잠금
- `RENAME TABLE tab_a TO tab_b` 같이 **테이블 이름을 변경**하는 경우 자동으로 획득 (명시적으로 획득 불가)
- `RENAME TABLE` 명령의 경우 **원본 이름과 변경될 이름 두 개 모두 한꺼번에 잠금** 설정

## 5.3. InnoDB 스토리지 엔진 잠금

### 5.3.1. InnoDB 스토리지 엔진의 잠금

MySQL에서 제공하는 잠금과 별개로 **레코드 기반 잠금 방식** 제공

*장점*

- MyISAM보다 뛰어난 **동시성 처리** 제공
- 잠금 정보가 작은 공간으로 관리되어 **락 에스컬케이션**(레코드락->페이지락->테이블락) 없음

*단점*

- **이원화된 잠금 처리** → InnoDB에서 사용되는 잠금에 대한 정보는 MySQL 명령을 이용해 접근하기 까다로움

> *레코드 락*

- **레코드 자체만**을 잠그는 것
- 다른 상용 DBMS의 레코드 락과 동일한 역할, 그러나 MySQL은 특이하게 레코드 자체가 아닌 **인덱스의 레코드**를 잠금
- 인덱스가 하나도 없는 테이블이더라도 내부적으로 자동 생성된 클러스터 인덱스를 이용해 잠금 설정

> *갭 락*

- 레코드 자체가 아닌 레코드와 **바로 인접한 레코드 사이의 간격만** 잠그는 것
- 레코드와 레코드 사이 간격에 새로운 레코드가 생성**(insert)되는 것 제어**

- 넥스트 키 락의 일부로 자주 사용

### *넥스트 키 락*

- **레코드 락**와 **갭 락**을 합쳐 놓은 형태의 잠금
- 갭락과 넥스트 키 락의 목적: 바이너리 로그에 기록되는 쿼리가 레플리카 서버에서 실행될 때 **소스 서버에서 만들어 낸 결과**와 동일한 결과를 만들어내도록 보장하는 것
- 문제점: 넥스트 키 락과 갭 락으로 인해 **데드락** 발생 / 다른 트랜잭션을 기다리게 만드는 일이 자주 발생

  → 해결: 바이너리 로그 포맷을 **ROW** 형태로 바꿔서 넥스트 키 락, 갭 락 줄이기

### *자동증가 락*

- AUTO_INCREMENT 칼럼이 사용된 테이블에 동시에 여러 레코드가 insert되는 경우, 저장되는 각 레코드는 **중복되지 않고** 저장된 **순서대로 증가하는 일련번호 값**을 갖게 만들기 위해
- InnoDB에서는 이를 위해 내부적으로 AUTO_INCREMENT 락이라는 테이블 수준 잠금 사용

- INSERT, REPLACE 처럼 **새로운 레코드를 저장하는 쿼리**에서만 필요함. (UPDATE, DELETE X)
- InnoDB의 다른 잠금(레코드 락 등)과 달리 **트랜잭션과 관계 없이** INSERT, REPLACE 문장에서 AUTOINCREMENT 값을 가져오는 순간만 락이 걸렸다가 즉시 해제됨.
- AUTO_INCREMENT 칼럼에 명시적으로 값을 설정해도 자동 증가 락을 걸게 됨.

- 명시적인 획득과 해제 불가.
- 아주 짧은 시간동안 걸렸다가 해제되는 잠금이라 대부분의 경우 문제 X
- MySQL 5.1 이상부터 `innodb_autoinc_lock_mode`라는 시스템 변수를 이용하여 자동 증가 락의 작동 방식 변경 가능 !!

### 5.3.2. 인덱스와 잠금

- **InnoDB**의 잠금 = **인덱스**를 잠그는 방식

  ➡️ 변경해야 할 레코드를 찾기 위해 인덱스의 레코드를 모두 락 걸어야 함

  ➡️ UPDATE를 위한 적절한 인덱스가 준비돼 있어야 함 ! (안 그러면 해당 조건의 레코드가 모두 잠김..)

### 5.3.3. 레코드 수준의 잠금 확인 및 해제

- 레코드 수준 잠금의 문제점: **레코드 각각에 잠금**이 걸리므로 그 레코드가 자주 사용되지 않는다면 오랜 시간 동안 잠겨진 상태로 있어도 잘 발견되지 X
- MySQL 5.1 부터  레코드 잠금과 잠금 대기에 대한 조회 가능 → 쿼리 하나만 실행하면 잠금과 잠금 대기를 바로 확인 가능

## 5.4. MySQL의 격리 수준

트랜잭션의 **격리수준(isolation level)** : 여러 트랜잭션이 동시에 처리될 때 **특정 트랜잭션**이 다른 트랜잭션에서 **변경되거나 조회하는 데이터를 볼 수 있게 허용할지** 말지를 결정하는 것

- 격리(고립) 정도 ⬆️, 동시 처리 성능 ⬇️
- 그렇다고 MySQL 서버 처리 성능 많이 떨어지진 X, (SERIALIZABLE 제외)

> *부정합의 문제*

- 격리 수준의 레벨에 따라 발생 여부가 달라짐
- REPEATABLE READ 격리 수준에서는 PHANTOM READ 발생 가능

InnoDB에서는 독특한 특성 때문에 REPEATABLE READ 격리 수준에서도 PHANTOM READ가 발생X

| | DIRTY READ | NON-REPEATABLE READ | PHANTOM READ |
|---|---|---|---|
| READ UNCOMMITTED | O | O | O |

| | DIRTY READ | NON-REPEATABLE READ | PHANTOM READ |
|---|---|---|---|
| READ COMMITTED | X | O | O |
| REPEATABLE READ | X | X | **O (InnoDB는 X)** |
| SERIALIZABLE | X | X | X |

- 일반적인 온라인 서비스 용도 : READ COMMITTED, REPEATABLE READ
- 오라클같은 DBMS : READ COMMITTED
- MySQL : REPEATABLE READ

*(* AUTOCOMMIT이 OFF인 상태에서만 테스트 가능 : SET autocommit=OFF)*

## 5.4.1. READ UNCOMMITTED (DIRTY READ)

- 각 트랜잭션에서의 **변경 내용**이 COMMIT/ROLLBACK 여부와 상관 없이 **다른 트랜잭션에서 보임 (=**Dirty Read)
- RDBMS 표준에서는 트랜잭션 격리 수준으로 인정하지 않을 정도로 **정합성에 문제가 많은** 격리 수준
  - 일반적인 DB에서 거의 사용 X
  - MySQL을 사용한다면, 최소 READ COMMITTED 이상의 격리 수준을 권장함.

## 5.4.2. READ COMMITTED

- **COMMIT이 완료된 데이터만** 다른 트랜잭션에서 **조회**할 수 있는 격리 수준 (→ Dirty Read 발생X)
- 그러나 **NON-REPEATABLE READ** 발생(부정합 발생)
  - 하나의 트랜잭션에서 동일 데이터를 여러 번 읽고 변경하는 작업이 금전적인 처리와 연결되면 문제될 수 있음
- 오라클 DBMS에서 기본으로 사용되고, 온라인 서비스에서 가장 많이 선택되는 격리수준

## 5.4.3. REPEATABLE READ

- InnoDB 스토리지 엔진에서 기본으로 사용되는 격리 수준

- 바이너리 로그를 가진 MySQL 서버에서는 최소 REPEATABLE READ 격리 수준 이상을 사용해야 함

- READ COMMITTED와 달리 NON-REPEATABLE READ 부정합이 발생X

** MVCC: InnoDB 스토리지 엔진에서 트랜잭션이 ROLLBACK될 가능성에 대비해 변경되기 전 레코드를 Undo 공간에 백업해두고 실제 레코드 값을 변경하는 기술

- REPEATABLE READ는 MVCC를 위해 Undo 영역에 백업된 이전 데이터를 이용해 **동일 트랜잭션 내에서는 동일한 결과**를 보여줄 수 있게 보장함.

- READ COMMITTED도 MVCC를 통해 COMMIT되기 전의 데이터를 보여줌.

그러나 차이는 “Undo 영역에 **백업된 레코드**의 여러 버전 가운데 **몇 번째 이전 버전**까지 보여주느냐'

- 언두 영역의 백업된 데이터는 불필요하다고 판단하는 시점에 주기적으로 삭제되지만, REAPEATABLE READ 격리 수준에서는 MVCC를 보장하기 위해 실행 중인 트랜잭션 가운데 가장 오래된 트랜잭션 번호보다 트랜잭션 번호가 앞선 언두 영역의 데이터를 삭제할 수 없음

- **PHANTOM READ 발생 :** 다른 트랜잭션에서 수행한 변경 작업에 의해 레코드가 보였다 안보였다 하는 현상

### 5.4.4. SERIALIZABLE

- 가장 **단순한** 격리 수준 && 가장 **엄격한** 격리 수준
- 그만큼 동시 처리 성능도 다른 트랜잭션 수준보다 떨어짐
- 동시성이 중요한 DB에서 거의 사용 X
- 읽기 작업도 공유 잠금(읽기 잠금)을 획득해야 하며, 동시에 다른 트랜잭션은 해당 레코드를 변경하지 못함

  ▶️ 한 트랜잭션에서 읽고 쓰는 레코드를 다른 트랜잭션에서 접근 불가
- PHANTOM READ가 발생X