Transaction

Transaction이란?

Transaction은 데이터베이스의 상태를 변환시키는 하나의 논리적 기능을 수행하기 위한 작업의 단위 또는 한꺼번에 모두 수행되어야 할 일련의 연산들을 의미한다.

Transaction의 특징

- 트랜잭션은 데이터베이스 시스템에서 병행 제어 및 회복 작업 시 처리되는 작업의 논리적 단위이다.
- 사용자가 시스템에 대한 서비스 요구 시 시스템이 응답하기 위한 상태 변환 과정의 작업 단위이다.
- 하나의 트랜잭션은 Commit되거나 Rollback된다.

트랜잭션의 성질

Atomicity(원자성)

- 트랜잭션의 연산은 데이터베이스에 모두 반영되든지 아니면 전혀 반영되지 않아야 한다.
- 트랜잭션 내의 모든 명령은 반드시 수행되어야 하며, 모두가 완벽히 수행되지 않고 어느 하나라도 오류가 발생하면 트랜잭션 전부가 취소되어야 한다.

Consistency(일관성)

- 트랜잭션이 그 실행을 성공적으로 완료하면 언제나 일관성 있는 데이터베이스 상태로 변환한다.
- 시스템이 가지고 있는 고정요소는 트랜잭션 수행 전과 트랜잭션 수행 완료 후의 상태가 같아야 한다.

Isolation(독립성, 격리성)

- 둘 이상의 트랜잭션이 동시에 병행 실행되는 경우 어느 하나의 트랜잭션 실행중에 다른 트랜잭션의 연산이 끼어들 수 없다.
- 수행중인 트랜잭션은 완전히 완료될 때까지 다른 트랜잭션에서 수행 결과를 참조할 수 없다.

Durability(영속성, 지속성)

- 성공적으로 완료된 트랜잭션의 결과는 시스템이 고장나더라도 영구적으로 반영되어야 한다.

트랜잭션 연산 및 상태

Commit 연산

1. Commit 연산은 한 개의 논리적 단위에 대한 작업이 성공적으로 끝났고 데이터베이스가 다시 일관된 상태에 있을 때, 이 트랜잭션이 행한 갱신 연산이 완료된 것을 트랜잭션 관리자에게 알려주는 연산이다.

Rollback : 트랜잭션 취소

Rollback은 진행중이던 트랜잭션 작업의 결과를 데이터베이스에 반영하지 않고 트랜잭션이 시작되기 전으로 작업을 취소하는 것을 말한다.

```
SELECT * FROM history;
START TRANSACTION;
DELETE * FROM history;
SELECT * FROM history;
ROLLBACK;
SELECT * FROM history;
```

위와 같은 쿼리를 구성하면 트랜잭션이 명시적으로 시작된 이후에 연산되는 DELETE와 SELECT가 ROLLBACK으로 인해 데이터베이스에 반영되지 않는다. 일반 쿼리문의 집합이었다면 DELETE가 되었기 때문에 이후 연산을 진행할 수 없었겠지만, 롤백으로 인해 원래 테이블로 돌아갔기 때문에 마지막 쿼리를 적용할 수가 있게 된다.

1. Rollback 연산은 하나의 트랜잭션 처리가 비정상적으로 종료되어 데이터베이스의 일관성을 깨뜨렸을 때, 이 트랜잭션의 일부가 정상적으로 처리되었더라도 트랜잭션의 원자성을 구현하기 위해 이 트랜잭션이 행한 모든 연산을 취소(Undo)하는 연산이다.
2. Rollback시에는 해당 트랜잭션을 재시작하거나 폐기한다.

Read-Only Transactions

어떤 트랜잭션이 데이터베이스에 쓰는 작업은 허용하지 않고 읽기 작업만 허용하는 것을 Read-Only Transaction으로 지정할 수 있다. 읽기만 하는 경우에는 데이터를 바꿔버리는 위험성이 없기 때문에 일반 Read & Write 트랜잭션보다 병렬성을 늘리는데 도움이 된다.

```
SET TRANSACTION READ ONLY;
START TRANSACTION;
...
COMMIT; or ROLLBACK;
```

Isolation Level

데이터 베이스에게 많은 트랜잭션이 들어왔을 경우에는 어떻게 이 작업들을 관리해주어야 할까? 만약 여러 트랜잭션이 동시다발적으로 수행된다면 데이터의 무결성이 깨질 가능성이 있다. 하지만 그렇다고 트랜잭션을 한 번에 하나씩만 돌아가면서 수행한다면 쌓여있는 트랜잭션을 모두 처리하기에는 응답속도가 크게 떨어질 것입니다. 따라서 트랜잭션들 사이에 발생하는 연산의 충돌을 방지하기 위해서 트랜잭션을 격리하는 전략을 사용합니다. 그리고 트랜잭션들을 격리하기 위한 격리 단계 수준을 설정할 수 있다.

```
SET TRANSACTION ISOLATION LEVEL <LEVEL>;
START TRANSACTION;
...
COMMIT; or ROLLBACK;
```

Issues in Isolation Level

트랜잭션은 Isolation level에 따라 몇 가지 이슈들이 발생할 수 있다.

- **Phantom Read** : 어떤 트랜잭션 T1이 SELECT를 통해 결과를 얻고 바로 이후에 T2 트랜잭션이 INSERT 쿼리를 통해 데이터베이스를 업데이트하고 커밋했다면, T1이 다시 한번 데이터베이스를 검색했을 때 T2에 의해서 첫 조회에는 없었던 새로운 데이터가 추가된 채로 조회된다.
- **Nonrepeatable Read** : 어떤 트랜잭션 T1이 SELECT를 통해 반복적으로 데이터를 조회하고 있을 때, 새로운 트랜잭션 T2가 데이터를 UPDATE하고 커밋을 한다면, T1은 실제 데이터베이스에 있는 값과 다른 값을 가지게 된다.
- **Dirty Read** : 어떤 트랜잭션 T1이 데이터를 변경하고 아직 커밋하지 않았을 때, 다른 트랜잭션이 해당 데이터를 조회하게 되면 아직 커밋되지 않은 데이터를 가져가게 되는데, 이 데이터는 최종적으로 커밋된다는 보장이 없다.

Set Isolation Level

Level 3 : SERIALIZABLE

- Serializable 레벨은 가장 엄격하게 트랜잭션들을 서로 격리시키는 방침이다.
- 병렬적으로 처리되는 트랜잭션을 허용하지 않기 때문에 데이터 무결성은 완벽하게 보장된다.
- 하지만 한 번에 하나의 트랜잭션만 처리하는 방침이기 때문에 성능이 좋지 않다는 단점이 있다.
- 이 방침에서는 **Phantom Read, Nonrepeatable Read, Dirty Read**가 모두 발생하지 않는다.

Level 2 : REPEATABLE READ

- 한 트랜잭션 안에서 한 번 조회한 내용은 여러 번 조회해도 계속 같은 값을 유지하는 것을 보장한다.
- 이 방침은 대상 트랜잭션이 시작되기 전에 이미 커밋이 완료된 데이터만을 조회하기 때문에 다른 트랜잭션의 UPDATE 연산의 영향을 받지 않는다.
- 따라서 **Dirty Read, Nonrepeatable Read** 이슈가 발생하지 않는다.
- 하지만 INSERT 연산을 통해 다른 트랜잭션이 테이블에 데이터를 추가하게 되면 테이블을 구성하는 집합이 달라지기 때문에 **Phantom Read** 이슈는 발생할 수 있다.

Level 1 : READ COMMITTED

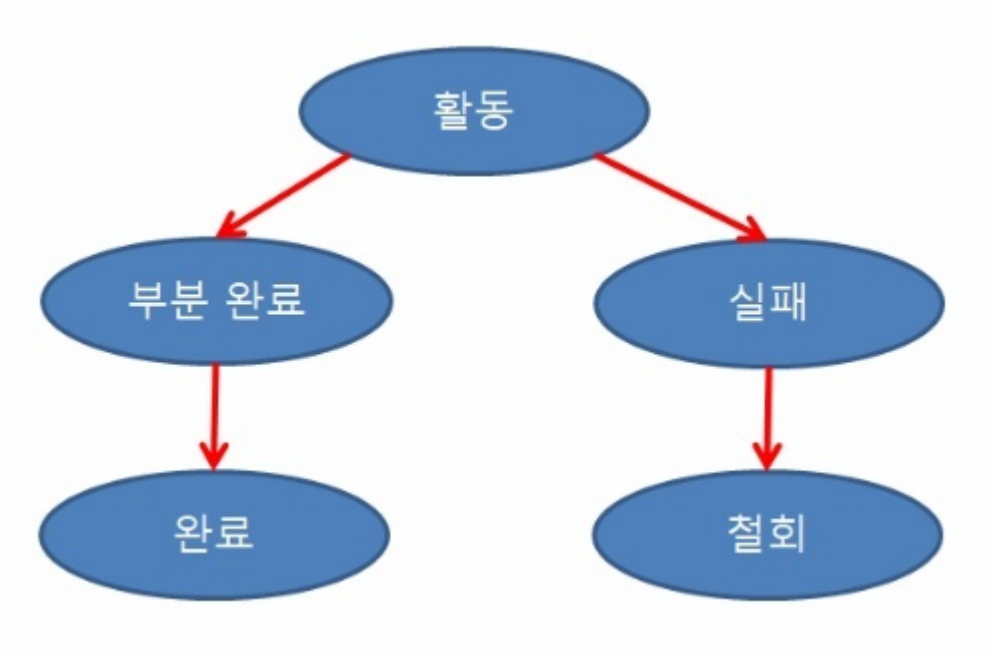
일반적으로 데이터베이스들이 가지고 있는 방침이다. 이 방침은 커밋이 완료된 데이터만 읽도록 한다. 따라서 Dirty Read 문제는 발생하지 않지만 **Nonrepeatable Read**나

Phantom Read는 발생할 여지가 있다.

Level 0 : READ UNCOMMITTED

모든 트랜잭션이 제한없이 데이터를 접근하는 방침이다. 커밋하지 않은 데이터도 다른 트랜잭션이 읽을 수 있다. 병렬적으로 트랜잭션이 수행되기 때문에 가장 성능이 좋지만 위에 설명한 세 문제들이 모두 발생할 여지가 있다.

트랜잭션의 상태



활동(Active) : 트랜잭션이 실행 중인 상태

실패(Failed) : 트랜잭션 실행에 오류가 발생하여 중단된 상태

철회(Aborted) : 트랜잭션이 비정상적으로 종료되어 Rollback 연산을 수행한 상태

부분 완료(Partially Committed) : 트랜잭션의 마지막 연산까지 실행했지만, Commit 연산이 실행되기 직전의 상태

완료(Committed) : 트랜잭션이 성공적으로 종료되어 Commit 연산을 실행한 후의 상태

Transaction 관리를 위한 DBMS의 전략

이해를 위한 2가지 개념 : DBMS의 구조 / Buffer 관리 정책

1. DBMS의 구조

- 크게 2가지 : Query Processor (질의 처리기), Storage System (저장 시스템)
- 입출력 단위 : 고정 길이의 page 단위로 disk에 읽거나 쓴다.
- 저장 공간 : 비휘발성 저장 장치인 disk에 저장, 일부분을 Main Memory에 저장

2. Page Buffer Manager or Buffer Manager

- DBMS의 Storage System에 속하는 모듈 중 하나로, Main Memory에 유지하는 페이지를 관리하는 모듈
    - Buffer 관리 정책에 따라, UNDO 복구와 REDO 복구가 요구되거나 그렇지 않게 되므로, transaction 관리에 매우 중요한 결정을 가져온다.

3. UNDO

필요한 이유 : 수정된 Page들이 Buffer 교체 알고리즘에 따라서 디스크에 출력될 수 있음. Buffer 교체는 transaction과는 무관하게 buffer의 상태에 따라서 결정됨. 이로 인해, 정상적으로 종료되지 않은 transaction이 변경한 page들은 원상 복구 되어야 하는데, 이 복구를 UNDO라고 한다.

- 2개의 정책(수정된 페이지를 디스크에 쓰는 시점으로 분류)
    - steal : 수정된 페이지를 언제든지 디스크에 쓸 수 있는 정
        - 대부분의 DBMS가 채택하는 Buffer 관리 정책
        - UNDO logging과 복구를 필요로 함.

- ¬steal : 수정도니 페이지들은 EOT(End Of Transaction)까지는 버퍼에 유지하는 정책
  - UNDO 작업이 필요하지 않지만, 매우 큰 메모리 버퍼가 필요함

4. REDO

이미 commit한 transaction의 수정을 재반영하는 복구 작업

Buffer 관리 정책에 영향을 받음

- Transaction이 종료되는 시점에 해당 transaction이 수정한 page를 디스크에 쓸 것이가 아닌가로 기준.
  - FORCE : 수정했던 모든 페이지를 Transaction commit 시점에 disk에 반영
    - transaction이 commit 되었을 때 수정된 페이지들이 disk 상에 반영되므로 REDO가 필요없다.
  - ¬FORCE : commit 시점에 반영하지 않는 정책
    - transaction이 disk 상의 db에 반영되지 않을 수 있기에 REDO 복구가 필요 (대부분의 DBMS 정책)

트랜잭션 스케줄

데이터베이스의 일관적인 상태를 유지하기 위해서 동시에 실행되는 **트랜잭션(병행 수행)들의 연산 순서를 정하는 것**을 의미한다. 연산 순서에 따라서 결과가 달랐기 때문에 병행 수행을 하기 위해서는 스케줄이 중요하다. 병행 수행에서 트랜잭션들은 차례로 번갈아 가면서 수행되는 인터리빙 방식으로 진행된다.

	T1	T2
S1	read(1000) write(1050)	read(1050) write(950)
S2	read(1000) write(1050)	read(1000) write(900)

[표1]

> final이 각각 950, 900으로 일차적이지 않다.

트랜잭션들의 연산을 실행하는 순서인 트랜잭션 스케줄은 세 가지 유형이 존재한다.

스케줄	의미
직렬 스케줄(Serial schedule)	인터리빙 방식을 이용하지 않고 각 트랜잭션별로 연산들을 순차적으로 실행한다.
비직렬 스케줄(nonserial schedule)	인터리빙 방식을 이용하여 트랜잭션들을 병행해서 수행시키는 것
직렬 가능 스케줄(serializable schedule)	직렬 스케줄과 같이 정확한 결과를 생성하는 비직렬 스케줄

> 정확한 결과를 만드는 스케줄은 직렬 스케줄이고 격리성이 가장 높다. 항상 consistent한 결과를 생성하지만 순차적으로 실행하기 때문에 처리량의 관점에서는 효율적이지 못하다. 스케줄이 직렬 실행은 아니지만 직렬 스케줄과 같은 결과를 내는 스케줄을 serializable이라 한다. 즉, 결과가 equivalent하다. serializable을 계산하기 위해서는 conflict serializability와 view serializability 두 가지 개념을 알아야 한다.

직렬 스케줄(Serial Schedule)

직렬 스케줄은 인터리빙 방식을 이용하지 않고 각 트랜잭션 별로 연산들을 순차적으로 실행시키는 것이다. 모든 트랜잭션이 완료될 때까지 다른 트랜잭션의 방해를 받지 않고 독립적으로 수행된다. 즉, 항상 정확한 결과를 만들며 격리성이 높다. 위에 직접 그린 그림에서 S1이 직렬 스케줄의 예시이다. 만약 T2의 모든 연산이 먼저 실행되어도 결과는 950으로 같다.

참고로 T1은 50을 더하고, T2는 100을 감소시킨다.

위에서 잠깐 언급한 것처럼, 인터리빙을 사용하지 않고, 독립적으로 수행하기 때문에 병행수행이라 할 수 없다. 즉, 데이터베이스의 처리량의 관점에서 효율적인 방식은 아니기 때문에 잘 사용하지 않는다.

비직렬 스케줄(Nonserial Schedule)

비직렬 스케줄은 인터리빙 방식을 이용해서 트랜잭션을 병행해서 수행시키는 것이다. 비직렬 스케줄에 따라 여러 트랜잭션을 수행하면 갱신 분실, 모순성, 연쇄 복귀 등의 문제가 발생할 수 있다. 이러한 문제는 다음에 따로 달도록 하겠다. 여기서는 그냥 최종 수행 결과의 정확성을 보장할 수 없다는 것만 기억하자. 예시는 위의 [표1]에서 S2를 확인하면 된다. 굳이 어떤 문제인지 분석해보자면 1번 트랜잭션의 갱신을 분실했고, 모순성의 가능성이 있다. 또한 가장 중요한 트랜잭션 일관성 또한 무너지게 된다.

직렬 가능 스케줄(Serializable schedule)

직렬 가능 스케줄은 직렬 스케줄에 따라 수행한 것과 같이 정확한 결과를 생성하는 비직렬 스케줄이다. 이렇게 정확한 결과를 보장하는 것은 serial schedule과 equivalent하다고 하다고 한다. 쉽게 설명하자면 모든 트랜잭션을 순차적으로 인터리빙 없이 실행한 결과를 내는 비직렬 스케줄이다. 즉, 제일 성능도 좋고 일관성을 유지할 수 있다. 다만 달리 생각해보면 그만큼 많은 개념이 존재할 것이다. 여러 개념을 파악해보자. 직렬 가능한 스케줄은 다음과 같이 두 개가 있고 포함관계이다.

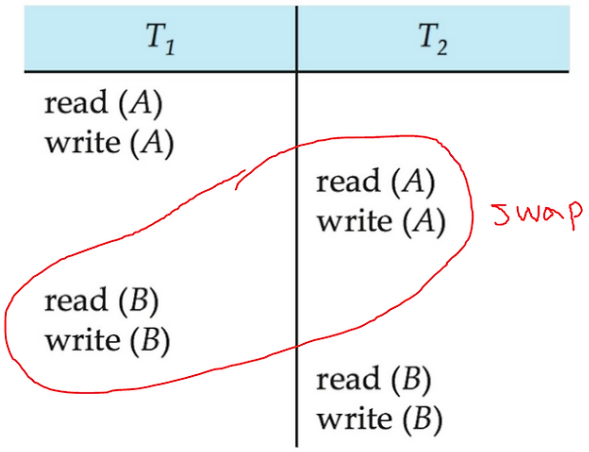
충돌 직렬 가능한 스케줄

동일한 데이터에 대해서 두 트랜잭션이 있다고 가정했을 때, 적어도 하나의 연산이 write인 경우 충돌(conflicting)이라 할 수 있다.

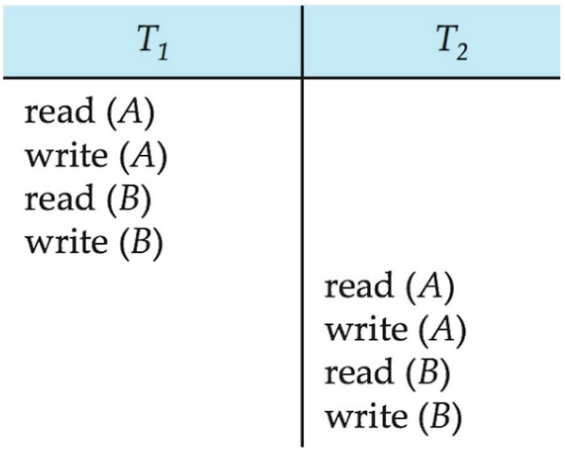
- I_i = **read**(Q), I_j = **read**(Q) → I_i and I_j don't conflict
- I_i = **read**(Q), I_j = **write**(Q) → conflict
- I_i = **write**(Q), I_j = **read**(Q) → conflict
- I_i = **write**(Q), I_j = **write**(Q) → conflict

충돌이 나지 않는 연산(non-conflicting instruction)들의 순서를 변경하여 새로운 스케줄을 만들 수 있다면 기존 스케줄과 새로운 스케줄은 충돌 동등하다. 그중에서 기존 스케줄이 직렬 스케줄과 충돌 동등하다면 기존 스케줄은 충돌 직렬 가능하다고 한다. 쉬운 이해를 위해서 정리해보겠다.

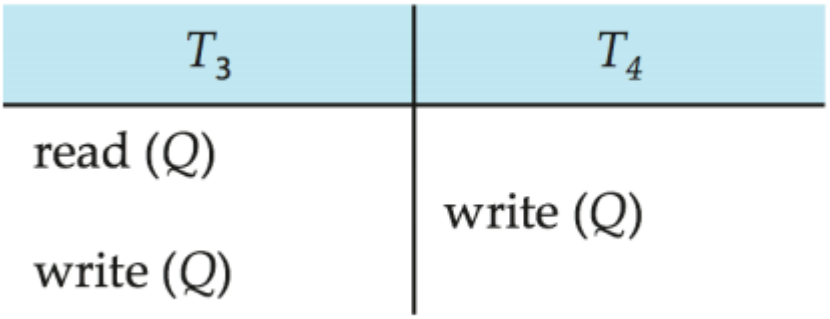
예시를 통해 이해해보자. 어느 한 스케줄의 충돌이 없는 연산의 순서를 조작해서 직렬 스케줄을 만들 수 있다면 그 스케줄은 직렬 가능 스케줄이며, 엄밀히는 충돌 가능한 스케줄이다. 아래 Schedule 3는 충돌 직렬 가능한 스케줄이다. 왜냐하면 T1과 T2의 충동하지 않는 연산을 스왑하여 직렬 스케줄을 변환할 수 있고, 서로 결과가 동등하기 때문이다.



Schedule 3



Schedule 6



위와 같은 경우에는 충돌하지 않는 연산을 스왑할 수 없기 때문에 충돌 직렬이 불가능하며, 충돌 직렬 가능하지 않은 스케줄이라 한다.

https://coding-factory.tistory.com/226

https://gyoogle.dev/blog/computer-science/data-base/Transaction.html

https://jeonyeohun.tistory.com/205

https://le2ksy.tistory.com/6