[DB] Transaction 심화

1. 트랜잭션 격리 수준(Read Uncommitted, Read Committed, Repeatable Read, Serializable)의 차이점과 각 수준에서 발생할 수 있는 문제점을 설명해주세요.

- Read Uncommitted: 다른 트랜잭션이 커밋하지 않은 결과값을 읽을 수 있는 상태 -> 더티리드 발생 가능
- Read Committed: 다른 트랜잭션이 커밋한 결과만 읽을 수 있는 상태 -> 반복 불가능한 읽기, 팬텀 리드 발생 가능
- Repeatable Read: 동일 트랜잭션 내에서 같은 데이터 조회 시 항상 같은 결과 반환 -> 팬텀 리드 발생 가능
- Serializable: 트랜직션을 직렬화하여 실행하는 가장 엄격한 격리 수준 -> 성능 저하

특정 격리 수준을 선택할 때 고려해야 할 성능과 데이터 일관성 간의 트레이드오프는 무엇인가요?

트랜잭션 격리 수준은 여러 트랜잭션이 동시에 작동할 때 각 트랜잭션이 서로에게 미치는 영향을 조정하는 설정입니다. 동시성과 정합성 사이의 균형을 조정하는 중요한 요소입니다. 격리 수준이 높을 수록 데이터 정합성은 보장되지만 트레이드 오프로 성능 저하가 발생할 수 있습니다.

이러한 격리 수준이 실제 비즈니스 로직에 어떤 영향을 미치는지 설명할 수 있나요?

성능이 중요한 경우는 검색 서비스나 로그 시스템을 예시로 들 수 있는데, 이 경우 Read Committed를 적용해야 한다. 반대로 데이터 일관성이 중요한 경우는 금융 거래와 주문 시스템을 들 수 있는데, 이 경우 Repeatable Read나 Serializable을 격리 수준으로 설정해야 한다.

2. 낙관적 락(Optimistic Lock)과 비관적 락(Pessimistic Lock)의 차이점과 각각 어떤 상황에서 사용하는 것이 적합한지 설명해주세요.

- 이 두 락 방식의 구현 방법 차이에 대해 설명해주세요.
- 동시성이 높은 환경에서 각 락 방식의 장단점은 무엇인가요?

> 락(Lock)은 여러 트랜잭션이 동일한 데이터에 동시에 접근할 때 충돌을 방지하기 위한 방법이다.

비관적 락(Pessimistic Lock)

> 비관적 락은 트랜잭션이 데이터에 접근할 때 즉시 잠금처리를 하여 다른 트랜잭션이 해당 데이터를 수정하지 못하도록 한다. 은행 시스템과 같은 **동시 수정 가능성이 높은 환경** 사용하는 것이 좋다. 안전하게 데이터를 보호할 수 있지만, 락으로 인하여 성능이 저하되고 교착 상태(Deadlock)이 발생할 수 있다.

비관적 락의 구현 방법

- 데이터베이스 레벨: `SELECT ... FOR UPDATE` 를 사용하여 행 단위의 락을 설정한다.
- 애플리케이션 레벨: `synchronized`, `ReetrantLock` 활용

낙관적 락(Optimistic Lock)

> 낙관적 락은 데이터 수정 시, 기존 값과 비교하여 변경 가능 여부를 판단한다. 쇼핑몰 상품 조회와 같이 **읽기 작업이 많고 충돌 가능성이 낮은 환경**에서 사용하는 것이 좋다. 낙관적 락은 동시성 높은 환경에서 성능이 우수하지만, 충돌 발생 시 재시도가 필요하여 응답 속도가 저하될 수 있다.

> => 따라서 비즈니스 로직과 데이터 동시성 요구사항에 따라 적절한 락을 선택해야 한다!

낙관적 락의 구현 방법

- 버전 번호 사용: 데이터 변경 시, 기존 버전과 비교 후 업데이트한다.
- 타임스탬프 사용: 최근 업데이트 시간과 비교하여 데이터 정합성을 유지한다.

3. 분산 트랜잭션의 개념과 구현 방법에 대해 설명해주세요.

- 2단계 커밋(Two-Phase Commit) 프로토콜은 어떻게 작동하나요?
- 분산 트랜잭션에서 발생할 수 있는 문제점은 무엇이며 이를 어떻게 해결할 수 있나요?

4. 교착상태(Deadlock)이 발생하는 원인과 이를 감지하고 해결하는 방법은 무엇인가요?

> 데드락은 두 개 이상의 트랜잭션이 서로의 락을 기다리면서 무한 대기가 발생하는 현상이다.

해결방법

1. 트랜잭션의 순서 정하기: 특정 순서로 실행하여 충돌을 방지한다.
2. 락 타임아웃 설정하기: 일정 시간이 지나면 롤백하도록 한다.
3. 락 획득 순서 일관성 을 유지하기: 테이블 A -> B 순서로 접근하도록 코드를 설계한다.

5. InnoDB 스토리지 엔진에서 레코드 락(Record Lock), 갭 락(Gap Lock), 넥스트 키 락(Next-Key Lock)의 차이점은 무엇인가요?

Record Lock

> 특정 행에 대한 락

Gap Lock

> 특정 범위에 대한 락 (팬텀 리드를 방지한다.)

Next-Key Lock

> Record Lock + Gap Lock

6. MySQL에서 사용하는 다양한 락의 종류(글로벌 락, 테이블 락, 네임드 락, 메타데이터 락 등)에 대해 설명해주세요.

글로벌 락 (Global Lock)

> 전체 데이터베이스 락

테이블 락 (Table Lock)

> 특정 테이블 단위 락

네임드 락 (Named Lock)

> 애플리케이션에서 정의하는 사용자 지정 락

메타데이터 락 (Metadata Lock)

> 스키마 변경 시 발생하는 락

7. 트랜잭션 격리 수준을 애플리케이션 코드 레벨에서 어떻게 제어하나요?

- Spring Framework에서 @Transactional 어노테이션 사용 시 격리 수준 설정 방법과 주의점은 무엇인가요?
- 애플리케이션 레벨에서 트랜잭션 격리 수준을 잘못 설정했을 때 발생할 수 있는 문제는 무엇인가요?

트랜잭션 격리 수준은 동시에 실행되는 트랜잭션 간의 영향을 미치는 수준을 설정함에 있어 트랜잭션 간의 데이터 정합성을 유지, 최적화하기 위한 중요한 설정이다. 애플리케이션 코드 레벨에서 격리 수준을 제어하는 방법은 데이터베이스 설정, SQL 직접 설정, 프레임 워크 설정 등이 있다.

특히 Spring Framework에서 `@Transaction` 어노테이션을 사용하여 트랜잭션 격리를 실행할 수 있는데, 주의할 점은 `@Transaction` 어노테이션을 사용하더라도 데이터베이스 자체 설정이 우선 적용될 수 있으며, 프록시 기반 트랜잭션으로 private 메서드에서는 동작하지 않는 다는 점이다. 즉, `@Transaction` 어노테이션이 적용된 메서드는 **Spring 컨테이너에서 관리되는 빈**에서 호출 되어야 한다.

전체적으로 트랜잭션 격리 수준을 잘못 설정하게 되면,

- 격리 수준이 너무 높아 성능 저하가 발생하거나,
- 격리 수준이 너무 낮아 데이터 정합성 문제가 발생하거나,
- 불필요한 트랜잭션을 유지하여 성능이 저하되거나,
- 트랜잭션 전파 속성을 잘못 설정하여 예상과 다른 결과를 얻을 수 있다.

8. MSA(Microservice Architecture) 환경에서 트랜잭션 관리 방법과 패턴에 대해 설명해주세요.

- SAGA 패턴이란 무엇이며 어떻게 구현하나요?
- 보상 트랜잭션(Compensating Transaction)이란 무엇이며 어떻게 동작하나요?

MSA에서는 주로 SAGA 패턴 또는 2PC 패턴을 사용하여 트랜잭션을 관리한다.

SAGA 패턴

> 트랜잭션을 여러 개의 로컬 트랜잭션으로 나누어 실행하고, 실패 시 **보상 트랜잭션**을 실행하는 방식이다. 동기 방식이 아니라 **비동기 이벤트 기반**으로 트랜잭션을 처리하여 MSA환경에서 트랜잭션 일관성을 유지한다.

SAGA 패턴의 동작 방식

1. 각 서비스가 독립적으로 로컬 트랜잭션을 실행
2. 한 서비스에서 트랜잭션이 실패하면 이전 서비스에서 보상 트랜잭션을 수행하여 데이터 일관성을 유지한다.

SAGA 패턴의 종류

둘의 차이는 트랜잭션을 **어디서** 처리하냐에 있다.

- 오케스트레이션 기반 SAGA
  - **중앙 컨트롤러가** 전체 트랜잭션의 흐름을 관리 => 흐름 추적이 쉽지만, 중앙 컨트롤러가 병목이 될 가능성이 있음
  - 한 서비스가 트랜잭션을 실행하고, 완료되면 다음 서비스가 실행됨 (트랜잭션 실행순서는 정해져 있지만, 서비스는 독립적으로 실행됨)
  - 예) 주문 처리 시스템: 주문 -> 결제 -> 배송 순차 관리, 트랜잭션의 흐름이 명확한 시스템에서 유리 (ex. 금융 거래, 예약 시스템)
- 코레오그래피 SAGA
  - **각 마이크로 서비스가** 이벤트를 주고 받으며 독립적으로 트랜잭션을 처리
  - 별도의 중앙 컨트롤러 없이, 이벤트 브로커(Kafka 등)을 통해 서비스 간 통신
  - 예) 결제 서비스 성공 -> "결제 완료" 이벤트 발생 -> 주문 서비스가 이를 받아 상태 업데이트, 전자상거래, 마이크로서비스 기반 플랫폼에 유리

SAGA 패턴의 장점

- 서비스 간 느슨한 결합 구조 유지 가능
- 장애 발생 시 부분 롤백이 가능해 시스템 안정성이 높음
- 확장성이 뛰어나고, 비동기 이벤트 기반으로 높은 성능 유지 가능

SAGA 패턴의 단점

- 일관성 보장이 어려움
- 설계 복잡성 증가

2PC(Two-Phase Commit 패턴)

> 트랜잭션을 2단계 (준비 단계 -> 커밋 단계)로 나누어 실행하는 방식으로 분산 트랜잭션을 **동기적**으로 관리할 수 있다.

2PC 패턴의 장점

- 데이터 일관성을 강하게 유지할 수 있음

2PC 패턴의 단점

- 동기 방식으로 성능이 떨어져 MSA 환경에서는 권장되지 않는다.

보상 트랜잭션

> 트랜잭션 도중 일부 작업이 실패했을 때, 이전에 완료된 작업을 롤백하여 데이터 일관성을 유지하려는 트랜잭션 패턴이다.

보상 트랜잭션은 MSA 환경에서 분산 트랜잭션을 관리하는 대표적인 방식이며, SAGA 패턴 에서 자주 사용된다.

기존 RDBMS의 ACID 기반 트랜잭션에서는 하나의 트랜잭션이 실패하면 `Rollback()` 호출하여 모든 작업을 원래 상태로 되돌릴 수 있었지만, MSA 환경에서는 각 서비스가 독립적인 데이터베이스를 사용하므로, 단일 트랜잭션으로 모든 서비스를 한꺼번에 롤백할 수 없다. 그렇기 때문에 보상 트랜잭션을 실행하여, 실패한 서비스 이전의 작업들을 취소하는 방식으로 트랜잭션을 관리한다.

보상 트랜잭션 예시

1. 주문 - 주문 서비스
2. 결제 승인 실패 - 결제 서비스
3. 주문 취소 (보상 트랜잭션) - 주문 서비스

9. 데이터베이스 성능 최적화 관점에서 트랜잭션을 어떻게 설계하고 관리해야 하나요?

> 트랜잭션의 범위를 최소화하여 실행 시간을 줄이고, 인덱스를 활용하여 성능을 최적화해야합니다. 트랜잭션이 불필요하게 길어지는 것을 방지하기 위해 배치 처리를 활용하고, 커밋 빈도를 적절히 조정해야합니다. 읽기 전용 트랜잭션(`READ ONLY`)을 적용하면 불필요한 락을 방지할 수 있으며, 데이터 일관성을 유지하면서도 성능을 높일 수 있습니다. 또한 트랜잭션 수준을 비즈니스 요구사항에 맞게 설정하여 성능과 데이터 정합성 간의 균형을 유지해야합니다.

- 여기서 배치처리란, 대량의 데이처를 한 번에 처리하여 트랜잭션을 효율적으로 관리하는 기법으로 트랜잭션이 불필요하게 길어지는 것을 방지하고 성능을 최적화하는데 사용된다.

장기 실행 트랜잭션(Long-running Transaction)의 문제점과 이를 해결하기 위한 방법은 무엇인가요?

> 장기 실행 트랙잭션은 데이터 락을 장기간 유지하여 동시성을 저하시킬 수 있으며, 데드락 발생 가능성을 높입니다. 트랜잭션이 길어질 수록 로그 크기가 증가하여 I/O 부하가 발생하고, 시스템 성능이 저하될 수 있습니다. 해결방법으로는 트랜잭션을 작은 단위로 분리하여 비동기적으로 처리하거나 읽기 작업을 캐싱하여 DB 부하를 줄이는 방법도 효과적입니다.

트랜잭션 로그(Transaction Log)의 역할과 관리 방법은 무엇인가요?

> 트랜잭션 로그는 데이터 변경 사항을 기록하여 장애 발생 시 복구할 수 있도록 지원하는 역할을 합니다. 로그는 `Write-Ahead Logging (WAL)` 방식으로 동작하며, 변경 내용을 먼저 로그에 기록한 후 실제 데이터를 반영하는 방식으로 데이터 무결성을 보장합니다.

트랜잭션 로그가 과도하게 쌓이면 성능 저하를 유발할 수 있으므로 **Checkpoint**를 설정하여 불필요한 로그를 정리하고 주기적으로 백업해야합니다. `Redo Log` 와 `Undo Log` 를 활용하여 장애 발생 시 트랜잭션을 복구하고, 데이터 정합성을 유지하는 것이 중요합니다. 로그 파일을 **별도의 디스크에 저장**하는 것도 성능 향상에 도움이 됩니다.

- `Redo Log` : 트랜잭션이 성공적으로 커밋된 이후에도 장애가 발생하면 다시 데이터를 복구하기 위한 로그
- `Undo Log` : 트랜잭션이 롤백될 때 이전상태로 데이터를 되돌리기 위한 로그

10. 읽기 전용 트랜잭션(Read-only Transaction)의 최적화 방법은 무엇인가요?

> 읽기 전용 트랜잭션은 데이터를 변경하지 않고 SELECT 작업만 수행하는 트랜잭션으로, 데이터 정합성을 유지하면서도 성능을 최적화할 수 있도록 설계됩니다.

- 트랜잭션 격리수준 최적화: 읽기 전용 트랜잭션에서는 높은 격리 수준이 필요지 않으므로 Read Committed 또는 Repeatable Read로 설정하여 성능을 향상시킬 수 있음.
- 인덱스 최적화: where절에 자주 사용되는 털럼에 인덱스를 적용하여 검색 속도를 개선할 수 있음.
- 캐싱 적용: 자주 조회하는 데이터에 캐싱을 적용하여 DB 부하를 줄여 성능을 개선할 수 있음.

11. 트랜잭션 관련 디자인 패턴(Unit of Work, Repository 등)에 대해 설명하고 실제 코드에서 어떻게 구현하는지 설명해주세요.

Unit of Work

> Unit of Work 패턴은 여러 개의 데이터 변경 작업을 하나의 트랜잭션으로 묶어 관리하는 패턴으로, 이를 통해 중복된 트랜잭션 처리를 방지하고, 커밋(Commit)과 롤백(Rollback)을 일괄적으로 처리할 수 있다.

예를 들어, JPA의 EntityManager를 사용하면 여러 엔티티 변경 사항을 한 번에 관리할 수 있다.

Repository

> Repository 패턴은 데이터베이스 접근 로직을 추상화해서 트랜잭션을 일관되게 관리하는 방식이다. 이를 활용하면, DAO(Data Access Object)와 비슷하게 특정 엔티티에 대한 데이터를 조회, 저장, 삭제하는 메서드를 한 곳에서 관리할 수 있다.

트랜잭션 전파(Propagation) 속성이란 무엇이며 어떻게 활용하나요?

> 트랜잭션 전파(Propagation) 속성은 현재 실행 중인 트랜잭션을 어떻게 처리할지 결정하는 설정이다. 쉽게 말하면, 하나의 트랜잭션이 실행될 때 이미 진행 중인 트랜잭션이 있다면 이를 유지할지, 새로운 트랜잭션을 만들지, 아니면 기존 트랜잭션을 무시할지를 결정하는 방식이다.

Spring에서는 `@Transactional(propagation = Propagation.XXX)` 형태로 설정할 수 있는데, 가장 많이 사용하는 속성들은 다음과 같다.

- `REQUIRED` (기본값): 기존 트랜잭션이 있으면 참여하고, 없으면 새로 생성
- `REQUIRES_NEW`: 항상 새로운 트랜잭션을 생성 (기존 트랜잭션이 있어도 무시)
- `SUPPORTS`: 트랜잭션이 있으면 참여하지만, 없어도 실행
- `MANDATORY`: 반드시 기존 트랜잭션이 있어야 실행 (없으면 예외 발생)
- `NEVER`: 트랜잭션이 있으면 예외 발생, 트랜잭션 없이 실행
- `NESTED`: 기존 트랜잭션 내에서 별도의 중첩 트랜잭션 실행

중첩 트랜잭션(Nested Transaction)이란 무엇이며 언제 사용하는 것이 적합한가요?

> 중첩 트랜잭션(Nested Transaction)이란, 하나의 트랜잭션 내부에서 또 다른 트랜잭션을 실행하는 방식을 의미한다. 즉, 부모 트랜잭션이 진행 중인 상태에서 하위 트랜잭션(자식 트랜잭션)이 수행되는 구조이며, 하위 트랜잭션에서 문제가 발생하면 부분 롤백(Partial Rollback)이 가능한 것이 특징이다.

중첩 트랜잭션을 사용하는 것이 적합한 상황

- 부분 롤백이 필요한 경우: 예를 들어, 사용자 주문 처리 과정에서 배송 정보 저장은 취소해야 하지만, 결제 내역은 유지해야 하는 경우.
- 한 개의 트랜잭션 내에서 논리적으로 구분된 여러 작업을 실행해야 할 때: 예를 들어, 대량의 데이터를 삽입하는 동안 특정 레코드에서 오류가 발생하더라도, 전체 트랜잭션을 롤백하지 않고 일부 데이터만 롤백 가능.
- 서브 트랜잭션이 독립적인 실행 단위로 관리될 필요가 있을 때: 특정 모듈(예: 로그 기록)이 트랜잭션이 롤백되더라도 로그는 남아 있어야 할 경우.

데이터 정합성

> 데이터 정합성(Data Integrity)이란 데이터의 정확성, 일관성, 신뢰성을 유지하는 개념을 의미. 즉, 데이터가 손상되지 않고, 올바른 상태를 유지하며, 서로 다른 시스템 간에서도 일관된 상태를 유지하는 것.