22.12.29

📌JPA - 기본 키 매핑

영속성 컨텍스트에 데이터를 저장하고 조회하는 모든 기준은 데이터베이스 기본 키(식별자 값)이다.

엔티티가 영속 상태가 되려면 식별자가 반드시 필요하다.

- 직접 할당 : 기본키를 애플리케이션에서 직접 할당한다.
- 자동 생성 : 대리 키 사용 방식
  - IDENTITY : 기본키 생성을 DB에 위임한다.
  - SEQUENCE : DB 시퀀스를 사용하여 기본키를 할당한다.
  - TABLE : 키 생성 테이블을 사용한다.

cf)

기본키: 실체에서 각 인스턴스를 **유일하게 식별**하는 데 가장 적합한 키. 테이블 컬럼중에 하나 선택

대리키: 식별자가 너무 길거나 여러 개의 속성으로 구성되어 있는 경우 인위적으로 추가하는 식별자

기본키 직접 할당

- @Id로 매핑한다.
- em.persist()로 엔티티를 저장하기 전에 애플리케이션에서 기본키를 직접 할당하는 방법

→ setId()

💡 em.persist()

- 객체를 데이터베이스에 저장
- persist() 메소드 호출시, JPA가 객체와 매핑정보를 보고 적절한 INSERT SQL를 생성해서 DB에 전달
- persist() 메소드는 엔티티 매니저를 사용해서 엔티티를 영속성 컨택스트에 저장한다.

  (=영속 상태로 만든다)

IDENTITY 전략

- 기본키 생성을 데이터베이스에 위임하는 전략
- em.persist()를 호출해서 엔티티를 저장한 직후에 할당된 식별자 값 출력
  - em.persist 즉시 해당 엔티티를 데이터베이스에 플러시 합니다
  - 데이터를 데이터베이스에 INSERT한 후에 기본키 값 조회 가능
  - 기본 키 값을 얻어오기 위해 데이터베이스를 추가로 조회한다.
  - 엔티티를 **데이터베이스에 저장한 후에 식별자를 조회**해서 엔티티의 식별자에 할당
- 트랜잭션을 지원하는 쓰기 지연이 동작하지 않는다.

  → **em.persist()로 객체를 영속화 시키는 시점에 곧바로 insert 쿼리가 DB로 전송**

  entityManager.persist()가 호출되자마자 INSERT SQL을 통해 DB에서 식별자를 조회하여 영속성 컨텍스트의 1차 캐시에 값을 넣는다.

  → 한꺼번에 쿼리를 날리지 않는다. 매번 생성마다 애플리케이션과 DB와의 네트워크가 매번 이루어진다.

- 사용처

MySQL, PostgreSQL, SQL Server, DB2

SEQUENCE 전략

em.persist()를 호출할 때 먼저 데이터베이스 시퀀스를 사용하여 **식별자 조회**하고 조회한 식별자를 **엔티티에 할당한 후에** 엔티티를 영속성 컨텍스트에 **저장**한다. 이후 트랜잭션을 커밋해서 플러시가 일어나면 **엔티티를 데이터베이스에 저장**한다.

- JPA는 **시퀀스에 접근하는 횟수를 줄이기 위해** allocationSize를 사용
  - allocationSize: 한 번의 시퀀스 접근을 통해 사용할 수 있는 PK값의 개수
  - 네트워크 접근 횟수를 비약적으로 줄일 수 있게 된다.
- 최초에 persist()시 엔티티의 식별자를 구하기 위해 DB의 시퀀스를 두 번 호출

  ex) **DB의 시퀀스 증가값이 50, allocationSize 가 기본값인 50인 경우**

  두 번 호출시 1과 51이 각각 리턴되는데, 1을 JPA가 메모리에서 관리할 시작값, 51을 끝(MAX)값 으로 지정. 엔티티에 51의 식별자가 할당되는 때까지는 DB에 시퀀스를 호출하지 않고 JPA가 직접 가상의 시퀀그 값을 할당
- 사용처

오라클, PostgreSQL, DB2, H2 데이터베이스에 사용

참고)

identity는 테이블에 종속적이지만, sequence는 독립적이라고 할수 있겠네요,,
여러 테이블에 하나의 sequence로 유일값을 줄 수 있죠,,
sequence만의 기능이라면 몇가지 더 있는데, 버퍼 크기를 지정할 수 있으며, 값을 회전시킬 수 있습니다,,
예를들어 1~100 까지 반복이 가능하죠,,

identity의 버퍼는 1000이 기본값으로 알고 있습니다,,

📌Real MySQL

MySQL 서버는 요청된 SQL 문장을 분석하고 최적화하는 MySQL엔진과 레코드단위로 실제 데이터를 읽고 저장하는 스토리지엔진을 합친 것을 말한다. 두 엔진의 요청과 응답은 핸들러에 의해 행해진다.

쿼리 실행구조

1. 쿼리파서

   쿼리문장을 토큰으로 분리하여 트리형태 구조로 만든다.

문법 오류를 확인한다.

2. 전처리기

   토큰을 테이블 이름, 칼럼 이름으로 매핑한다.

   존재 여부, 객체 접근 권한 확인한다.

3. 옵티마이저

   최적화와 실행계획을 생성한다.

4. 쿼리 실행기 (실행엔진): 요청하기, 요청 받아서 입력으로 연결하기
5. 핸들러 : 요청처리

글로벌 메모리, 로컬메모리 영역

- 글로벌 메모리 영역
  - 서버 시작시 OS로부터 할당받음
  - 하나의 메모리 공간
  - 모든 스레드에 듸해 공유된다.
- 로컬 메모리 영역(=세션 메모리 영역)
  - 요청당 스레드 하나씩 만들어진다. 독립적이고 공유되지 않는다.

InnoDB 스토리지 특징

1. **primary key에 의한 클러스터링** (pk값의 순서대로 디스크에 저장된다.)

2. **외래키 지원**
잠금이 여러 테이블로 전파 → 데드락 요인

3. **Multi Version Concurrency Control**

- 하나의 레코드에 대해 여러 버전이 동시에 관리된다.
- 잠금을 사용하지 않는 일관된 읽기 제공 → 언두 로그로 구현
  읽기 작업 lock 기다리지 않고 읽기 작업이 가능 (Read_Uncommitted,

Read_Committed, Repeatable_Read)

- Update 문장 실행시 커밋 실행 여부 상관 없이
  버퍼풀 : 새로운 값 → Read_Uncommitted
  언두로그 : 변경 전 값 → Read_Committed, Repeatable_Read, Serialisable
- Rollback 실행시 언두 영역의 데이터를 버퍼풀로 다시 복구하고 언두영역 삭제
  커밋시 언두 로그 영역 데이터는 트랜잭션이 더는 없을 때야 삭제

4. **자동 데드락 감지**

잠금 대기 목록을 그래프 형태로 관리 (wait-for list) → 데드락 감지 스레드가 이용
언두 로그 레크드를 적게 가질 수록 종료 우선순위 높음

- 데드락 감지 스레드 : 잠금 대기 목록(잠금 테이블)에 잠금을 걸고 검사 → CPU 소모 높음
- 일정 시간 지나면 자동으로 요청 실패하도록

5. **자동 장애 복구**

InnoDB 엔진 아키텍처

- 버퍼풀 : 디스크의 데이터 파일이나 인덱스 정보를 메모리에 캐시에 두는 공간
- change buffer : 변경해야 할 인덱스 페이지의 임시 메모리 공간
- Adaptive hash index : 사용자가 자주 요청하는 데이터에 대해 자동으로 생성하는 인덱스
- Doublewrite Buffer: 변경된 페이지 기록
- Undo Logs : 데이터 변경 전 내용 기록하는 곳
- Redo Log :데이터 변경한 내용 기록하는 곳.

스레드풀

Foreground Thread는 사용자의 요청마다 만들어진다. Foreground Thread는 사용자가 작업을 마치고 세션이 종료되면 해당 스레드는 Thread Pool(캐시)로 돌아간다. 이때 이미 스레드 캐시에 일정 개수 이상의 대기중인 스레드가 있으면 스레드 캐시에 넣지 않고 스레드를 종료시켜 일정 개수의 스레드만 캐시에 존재하게 한다.

Background Thread 에서는 스레드풀에 대한 언급은 없음