성능 최적화

JPA로 애플리케이션을 개발할 때 발생하는 다양한 성능 문제와 해결 방안을 알아보자.

N+1 문제

JPA로 애플리케이션을 개발할 때 성능상 가장 주의해야 하는 것이 N+1 문제다. (N+1 문제는 웹 애플리케이션 심화편의 글로벌 페치 전략에 즉시 로딩 사용 시 단점에서 이미 설명했지만 중요한 내용이라 약간 다른 관점에서 다시 한번 설명하겠다.)

다음 예제를 통해 N+1 문제에 대해 알아보자.

====== 회원 ======

```
@Entity
public class Member {

    @Id @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "member", fetch = FetchType.EAGER) //**
    private List<Order> orders = new ArrayList<Order>();
    ...
}
```

====== 회원의 주문정보 ======

```
@Entity
@Table(name = "ORDERS")
public class Order {

    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    private Member member;
    ...
}
```

회원과 주문 정보는 1:N, N:1 양방향 연관관계다. 그리고 회원이 참조하는 주문 정보인 `Member.orders`를 즉시 로딩으로 설정했다.

- 즉시 로딩과 N+1

특정 회원 하나를 `em.find()` 메서드로 조회하면 즉시 로딩으로 설정한 주문 정보도 함께 조회한다.

```
em.find(Member.class, id);
```

====== 실행된 SQL ======

```
SELECT M.*, O.*
FROM
    MEMBER M
OUTER JOIN ORDERS O ON M.ID=O.MEMBER_ID
```

여기서 함께 조회하는 방법이 중요한데 SQL을 두 번 실행하는 것이 아니라 조인을 사용해서 한 번의 SQL로 회원과 주문정보를 함께 조회한다. 여기까지만 보면 즉시 로딩이 상당히 좋아보인다. 문제는 JPQL을 사용할 때 발생한다.

즉시 로딩은 주로 JPQL을 사용할 때 문제가 발생한다.

```
List<Member> members = em.createQuery("select m from Member m", Member.class)
                         .getResultList();
```

JPQL을 실행하면 이것을 분석해서 SQL을 생성한다. 이때는 즉시 로딩과 지연 로딩에 대해서 전혀 신경쓰지 않고 JPQL만 사용해서 SQL을 생성한다. 따라서 다음과 같은 SQL이 실행된다.

```
SELECT * FROM MEMBER
```

SQL의 실행 결과로 먼저 회원 엔티티를 애플리케이션에 로딩한다. 그런데 회원 엔티티와 연관된 주문 컬렉션이 즉시 로딩으로 설정되어 있으므로 JPA는 주문 컬렉션을 즉시 로딩하려고 다음 SQL을 추가로 실행한다.

```
SELECT * FROM ORDERS WHERE MEMBER_ID=?
```

조회된 회원이 하나면 이렇게 총 2번의 SQL을 실행하지만 조회된 회원이 5명이면 어떻게 될까?

```
SELECT * FROM MEMBER //1번 실행으로 회원 5명 조회
SELECT * FROM ORDERS WHERE MEMBER_ID=1 //회원과 연관된 주문
SELECT * FROM ORDERS WHERE MEMBER_ID=2 //회원과 연관된 주문
SELECT * FROM ORDERS WHERE MEMBER_ID=3 //회원과 연관된 주문
SELECT * FROM ORDERS WHERE MEMBER_ID=4 //회원과 연관된 주문
SELECT * FROM ORDERS WHERE MEMBER_ID=5 //회원과 연관된 주문
```

먼저 회원 조회 SQL로 5명의 회원 엔티티를 조회했다. 그리고 조회한 각각의 회원 엔티티와 연관된 주문 컬렉션을 즉시 조회하려고 총 5번의 SQL을 추가로 실행했다. 이처럼 처음 실행한 SQL의 결과 수만큼 추가로 SQL을 실행하는 것을 N+1 문제라 한다.

즉시 로딩은 JPQL을 실행할 때 N+1 문제가 발생할 수 있다.

- 지연 로딩과 N+1

회원과 주문을 지연 로딩으로 설정하면 어떻게 될까? 방금 살펴본 즉시 로딩 시나리오를 지연 로딩으로 변경해도 N+1 문제에서 자유로울 수는 없다. 회원이 참조하는 주문 정보를 `FetchType.LAZY`로 지정해서 지연 로딩으로 설정해보자.

```
@Entity
public class Member {

    @Id @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "member", fetch = FetchType.LAZY) //**
    private List<Order> orders = new ArrayList<Order>();
    ...
}
```

지연 로딩으로 설정하면 JPQL에서는 N+1 문제가 발생하지 않는다.

```
List<Member> members = em.createQuery("select m from Member m", Member.class)
                         .getResultList();
```

지연 로딩이므로 데이터베이스에서 회원만 조회 된다. 따라서 다음 SQL만 실행되고 연관된 주문 컬렉션은 프록시로 조회된다.

```
SELECT * FROM MEMBER
```

이후 비즈니스 로직에서 주문 컬렉션을 실제 사용할 때 지연 로딩이 발생한다.

```
firstMember = members.get(0);
```

```
firstMember.getOrders().size(); //지연 로딩 발생
```

`members.get(0)`로 회원 하나만 조회해서 사용했기 때문에 실행되는 SQL은 다음과 같다.

```
SELECT * FROM ORDERS WHERE MEMBER_ID=?
```

문제는 다음처럼 모든 회원에 대해 연관된 주문 컬렉션을 사용할 때 발생한다.

```
for (Member member : members) {
    System.out.println("member = " + member.getOrders().size());
}
```

주문 컬렉션을 초기화하는 수만큼 다음 SQL이 실행될 수 있다. 회원이 5명이면 회원에 따른 주문도 5번 조회된다.

```
SELECT * FROM ORDERS WHERE MEMBER_ID=1 //회원과 연관된 주문
SELECT * FROM ORDERS WHERE MEMBER_ID=2 //회원과 연관된 주문
SELECT * FROM ORDERS WHERE MEMBER_ID=3 //회원과 연관된 주문
SELECT * FROM ORDERS WHERE MEMBER_ID=4 //회원과 연관된 주문
SELECT * FROM ORDERS WHERE MEMBER_ID=5 //회원과 연관된 주문
```

이것도 결국 N+1 문제다.

지금까지 살펴본 것처럼 N+1 문제는 즉시 로딩과 지연 로딩일 때 모두 발생할 수 있다.

지금부터 N+1 문제를 피할 수 있는 다양한 방법을 알아보자.

- 페치 조인 사용

이 문제를 해결하는 가장 일반적인 방법은 페치 조인을 사용하는 것이다. 페치 조인은 SQL 조인을 사용해서 연관된 엔티티를 함께 조회하므로 N+1 문제가 발생하지 않는다.

====== JPQL 페치 조인 ======

```
select m from Member m join fetch m.orders
```

====== 실행된 SQL ======

```
SELECT M.*, O.* FROM MEMBER M
INNER JOIN ORDERS O ON M.ID=O.MEMBER_ID
```

참고로 이 예제는 일대다 조인을 했으므로 결과가 늘어나서 중복된 결과가 나타날 수 있다. 따라서 JPQL의 `DISTINCT`를 사용해서 중복을 제거하는 것이 좋다.

- 하이버네이트 `@BatchSize`

하이버네이트가 제공하는 `org.hibernate.annotations.BatchSize` 어노테이션을 사용하면 연관된 엔티티를 조회할 때 지정한 `size`만큼 SQL의 IN 절을 사용해서 조회한다. 만약 조회한 회원이 10명인데 `size=5`로 지정하면 2번의 SQL만 추가로 실행한다.

====== BatchSize 적용 ======

```
@Entity
public class Member {
    ...
    @org.hibernate.annotations.BatchSize(size = 5)
    @OneToMany(mappedBy = "member", fetch = FetchType.EAGER)
    private List<Order> orders = new ArrayList<Order>();
    ...
}
```

즉시 로딩으로 설정하면 조회 시점에, 지연 로딩으로 설정하면 지연 로딩된 엔티티를 사용하는 시점에 다음 SQL이 실행된다.

====== 실행된 SQL ======

```
SELECT * FROM ORDERS
WHERE MEMBER_ID IN (
        ?, ?, ?, ?, ?
)
```

- 하이버네이트 @Fetch(FetchMode.SUBSELECT)

하이버네이트가 제공하는 `org.hibernate.annotations.Fetch` 어노테이션을 사용하면 연관된 데이터를 조회할 때 서브 쿼리를 사용해서 N+1 문제를 해결한다.

====== @Fetch 적용 ======

```
@Entity
public class Member {
    ...
    @org.hibernate.annotations.Fetch(FetchMode.SUBSELECT)
    @OneToMany(mappedBy = "member", fetch = FetchType.EAGER)
    private List<Order> orders = new ArrayList<Order>();
    ...
}
```

회원 식별자 값이 10을 초과하는 회원을 모두 조회해 보자.

====== JPQL ======

```
select m from Member m where m.id > 10
```

즉시 로딩으로 설정하면 조회 시점에, 지연 로딩으로 설정하면 지연 로딩된 엔티티를 사용하는 시점에 다음 SQL이 실행된다.

====== 실행된 SQL ======

```
SELECT O FROM ORDERS O
  WHERE O.MEMBER_ID IN (
        SELECT
            M.ID
        FROM
            MEMBER M
        WHERE M.ID > 10
    )
```

정리

필자가 추천하는 방법은 즉시 로딩은 사용하지 말고 지연 로딩만 사용하는 것이다. 즉시 로딩 전략은 그럴듯해 보이지만 N+1 문제는 물론이고 비즈니스 로직에 따라 필요하지 않은 엔티티를 로딩해야 하는 상황이 자주 발생한다. 그리고 즉시 로딩의 가장 큰 문제는 성능 최적화가 어렵다는 점이다. 엔티티를 조회

하다보면 즉시 로딩이 연속 발생해서 전혀 예상하지 못한 SQL이 실행될 수 있다. 따라서 모두 지연 로딩으로 설정하고 성능 최적화가 꼭 필요한 곳에는 JPQL 페치 조인을 사용하자.

JPA의 글로벌 페치 전략 기본값은 다음과 같다.

- `@OneToOne`, `@ManyToOne` : 기본 페치 전략은 즉시 로딩
- `@OneToMany`, `@ManyToMany` : 기본 페치 전략은 지연 로딩

따라서 기본값이 즉시 로딩인 `@OneToOne`과 `@ManyToOne`은 `fetch = FetchType.LAZY`로 설정해서 지연 로딩 전략을 사용하도록 변경하자.

읽기 전용 쿼리의 성능 최적화

엔티티가 영속성 컨텍스트에 관리되면 1차 캐시부터 변경 감지까지 얻을 수 있는 해택이 많다. 하지만 영속성 컨텍스트는 변경 감지를 위해 스냅샷 인스턴스를 보관하므로 더 많은 메모리를 사용하는 단점이 있다.

예를 들어 100건의 구매 내용을 출력하는 단순한 조회 화면이 있다고 가정해보자. 그리고 조회한 엔티티를 다시 조회할 일도 없고 수정할 일도 없이 딱 한 번만 읽어서 화면에 출력하면 된다. 이때는 읽기 전용으로 엔티티를 조회하면 메모리 사용량을 최적화할 수 있다.

다음 쿼리를 최적화해보자.

```
select o from Order o
```

스칼라 타입으로 조회

가장 확실한 방법은 다음처럼 엔티티가 아닌 스칼라 타입으로 모든 필드를 조회하는 것이다. 스칼라 타입은 영속성 컨텍스트가 결과를 관리하지 않는다.

```
select o.id, o.name, o.price from Order p
```

읽기 전용 쿼리 힌트 사용

```
TypedQuery<Order> query = em.createQuery("select o from Order o", Order.class);
query.setHint("org.hibernate.readOnly", true);
```

하이버네이트 전용 힌트인 `org.hibernate.readOnly`를 사용하면 엔티티를 읽기 전용으로 조회할 수 있다. 읽기 전용이므로 영속성 컨텍스트는 스냅샷을 보관하지 않는다. 따라서 메모리 사용량을 최적화할 수 있다. 단 스냅샷이 없으므로 엔티티를 수정해도 데이터베이스에 반영되지 않는다.

읽기 전용 트랜잭션 사용하기

스프링 프레임워크를 사용하면 트랜잭션을 읽기 전용 모드로 설정할 수 있다.

```
@Transactional(readOnly = true)
```

트랜잭션에 `readOnly=true` 옵션을 주면 스프링 프레임워크가 하이버네이트 세션의 플러시 모드를 MANUAL로 설정한다. 이렇게 하면 강제로 플러시를 호출하지 않는 한 플러시가 일어나지 않는다. 따라서 트랜잭션을 커밋해도 영속성 컨텍스트를 플러시하지 않는다. 영속성 컨텍스트를 플러시하지 않으니 엔티티의 등록, 수정, 삭제는 당연히 동작하지 않는다. 하지만 플러시 때 일어나는 스냅샷 비교와 같은 무거운 로직들을 수행하지 않으므로 성능이 향상된다. 물론 트랜잭션을 시작했으므로 트랜잭션 시작, 로직수행, 트랜잭션 커밋의 과정은 이루어는 진다. 단지 영속성 컨텍스트를 플러시 하지 않을 뿐이다.

> **참고:** 엔티티 매니저의 플러시 설정에는 MANUAL 모드가 없지만 하이버네이트의 세션의 플러시 설정에는 MANUAL 모드가 있다.

트랜잭션 밖에서 읽기

트랜잭션 밖에서 읽는다는 것은 트랜잭션 없이 엔티티를 조회한다는 뜻이다. 물론 JPA에서 데이터를 변경하려면 트랜잭션은 필수다. 따라서 조회가 목적일 때만 사용해야 한다.

스프링 프레임워크를 사용하면 다음처럼 설정한다.

```
@Transactional(propagation = Propagation.NOT_SUPPORTED) //Spring
```

J2EE 표준 컨테이너를 사용하면 다음처럼 설정한다.

```
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED) //J2EE
```

이렇게 트랜잭션을 사용하지 않으면 플러시가 일어나지 않으므로 조회 성능이 향상된다.

트랜잭션 밖에서 읽기를 조금 더 보충하자면 기본적으로 플러시 모드는 AUTO로 설정되어 있다. 따라서 트랜잭션을 커밋하거나 쿼리를 실행하면 플러시가 작동한다. 그런데 트랜잭션 자체가 존재하지 않으므로 트랜잭션을 커밋할 일이 없다. 그리고 JPQL 쿼리도 트랜잭션 없이 실행하면 플러시를 호출하지 않는다.

정리

읽기 전용 쿼리의 메모리를 최적화 하려면 스칼라 타입으로 조회하거나 하이버네이트가 제공하는 읽기 전용 쿼리 힌트를 사용하면 되고, 플러시 호출을 막아서 속도를 최적화 하려면 읽기 전용 트랜잭션을 사용하거나 트랜잭션 밖에서 읽기를 사용하면 된다. 참고로 스프링 프레임워크를 사용하면 읽기 전용 트랜잭션을 사용하는 것이 편리하다.

따라서 읽기 전용 트랜잭션(또는 트랜잭션 밖에서 읽기)과 읽기 전용 쿼리 힌트(또는 스칼라 타입으로 조회)를 동시에 사용하는 것이 가장 효과적이다. 다음 예를 보자.

```
@Transactional(readOnly = true)  //1. 읽기 전용 트랜잭션 //**
public List<DataEntity> findDatas() {

    return em.createQuery("select d from DataEntity d", DataEntity.class)
            .setHint("org.hibernate.readOnly", true) //2. 읽기 전용 쿼리 힌트 //**
            .getResultList();
}
```

1. 읽기 전용 트랜잭션 사용 : 플러시를 작동하지 않도록 해서 성능 향상
2. 읽기 전용 엔티티 사용 : 엔티티를 읽기 전용으로 조회해서 메모리 절약

배치 처리

수백만 건의 데이터를 배치 처리해야 하는 상황이라 가정해보자. 일반적인 방식으로 엔티티를 계속 조회하면 영속성 컨텍스트에 아주 많은 엔티티가 쌓이면서 메모리 부족 오류가 발생한다. 따라서 이런 배치 처리는 적절한 단위로 영속성 컨텍스트를 초기화해야 한다. 또한 2차 캐시를 사용하고 있다면 2차 캐시에 엔티티를 보관하지 않도록 주의해야 한다.

- JPA 등록 배치

수천에서 수만 건 이상의 엔티티를 한 번에 등록할 때 주의할 점은 영속성 컨텍스트에 엔티티가 계속 쌓이지 않도록 일정 단위마다 영속성 컨텍스트의 엔티티를 데이터베이스에 플러시하고 영속성 컨텍스트를 초기화해야 한다. 만약 이런 작업을 하지 않으면 영속성 컨텍스트에 너무 많은 엔티티가 저장되면서 메

모리 부족 오류가 발생할 수 있다.

다량의 엔티티를 등록하는 예제를 보자.

====== JPA 등록 배치 예제 ======

```
EntityManager em = entityManagerFactory.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

for (int i = 0; i < 100000; i++) {
    Product product = new Product("item" + i, 10000);
    em.persist(product);

    //100건 마다 플러시와 영속성 컨텍스트 초기화
    if ( i % 100 == 0 ) {
        em.flush();
        em.clear();
    }
}

tx.commit();
em.close();
```

예제는 100건 마다 플러시를 호출하고 영속성 컨텍스트를 초기화한다.

지금까지 등록 배치 처리를 알아보았다. 이번에는 수정 배치 처리를 알아보자.

배치 처리는 아주 많은 데이터를 조회해서 수정한다. 이때 수 많은 데이터를 한 번에 메모리에 올려둘 수 없어서 2가지 방법을 주로 사용한다.

- 페이징 처리: 데이터베이스 페이징 기능을 사용한다.
- 커서: 데이터베이스가 지원하는 커서 기능을 사용한다.

- JPA 페이징 배치 처리

JPA를 사용하는 페이징 배치 처리 예제를 보자.

====== JPA 페이징 배치 처리 예제 ======

```
EntityManager em = entityManagerFactory.createEntityManager();
EntityTransaction tx = em.getTransaction();
```

```
tx.begin();

int pageSize = 100;
for (int i = 0; i < 10; i++) {

    List<Product> resultList = em.createQuery("select p from Product p", Product
            .setFirstResult(i * pageSize)
            .setMaxResults(pageSize)
            .getResultList();

    //비즈니스 로직 실행
    for (Product product : resultList) {
        product.setPrice(product.getPrice() + 100);
    }

    em.flush();
    em.clear();
}
tx.commit();
em.close();
```

한 번에 100건씩 페이징 쿼리로 조회하면서 상품의 가격을 100원씩 증가한다. 다음으로 커서를 사용하는 방법을 알아보자.

JPA는 JDBC 커서(CURSOR)를 지원하지 않는다. 따라서 커서를 사용하려면 하이버네이트 세션(Session)을 사용해야 한다.

- 하이버네이트 `scroll` 사용

하이버네이트는 `scroll`이라는 이름으로 JDBC 커서를 지원한다.

====== 하이버네이트 `scrool` 사용 예제 ======

```
EntityTransaction tx = em.getTransaction();
Session session = em.unwrap(Session.class);

tx.begin();
ScrollableResults scroll = session.createQuery("select p from Product p")
        .setCacheMode(CacheMode.IGNORE)
        .scroll(ScrollMode.FORWARD_ONLY);

int count = 0;

while (scroll.next()) {
    Product p = (Product) scroll.get(0);
    p.setPrice(p.getPrice() + 100);
```

```
    count++;
    if (count % 100 == 0) {
        session.flush(); //플러시
        session.clear(); //영속성 컨텍스트 초기화
    }
}
tx.commit();
session.close();
```

- 하이버네이트 무상태 세션 사용

하이버네이트는 무상태 세션이라는 특별한 기능을 제공한다. 이름 그대로 무상태 세션은 영속성 컨텍스트를 만들지 않고 심지어 2차 캐시도 사용하지 않는다. 무상태 세션은 영속성 컨텍스트가 없다. 그리고 엔티티를 수정하려면 무상태 세션이 제공하는 `update` 메서드를 직접 호출해야 한다.

```
SessionFactory sessionFactory = entityManagerFactory.unwrap(SessionFactory.class
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();
ScrollableResults scroll = session.createQuery("select p from Product p").scroll

while (scroll.next()) {
    Product p = (Product) scroll.get(0);
    p.setPrice(p.getPrice() + 100);
    session.update(p); //직접 update를 호출해야 한다. //**
}
tx.commit();
session.close();
```

SQL 쿼리 힌트 사용하기

JPA는 데이터베이스 SQL 힌트 기능을 제공하지 않는다. SQL 힌트를 사용하려면 하이버네이트를 직접 사용해야 한다. (참고로 이 기능은 JPA 구현체에게 제공하는 힌트가 아니고 데이터베이스 벤더에게 제공하는 힌트다.)

SQL 힌트는 하이버네이트 쿼리가 제공하는 `addQueryHint` 메서드를 사용한다. Oracle 데이터베이스를 사용한 예제를 보자.

```
Session session = em.unwrap(Session.class); //하이버네이트 직접 사용
```

```
List<Member> list = session.createQuery("select m from Member m")
      .addQueryHint("FULL (MEMBER)") //SQL HINT 추가
      .list();
```

====== 실행된 SQL ======

```
select
    /*+ FULL (MEMBER) */ m.id, m.name
from
    Member m
```

실행된 SQL을 보면 추가한 힌트가 있다.

현재 하이버네이트 4.3.8 버전에는 Oracle 방언에만 힌트가 적용되어 있다. 다른 데이터베이스에서 SQL 힌트를 사용하려면 각 방언에서 `org.hibernate.dialect.Dialect`에 있는 다음 메서드를 오버라이딩해서 기능을 구현해야 한다.

```
public String getQueryHintString(String query, List<String> hints) {
    return query;
}
```

트랜잭션을 지원하는 쓰기 지연과 성능 최적화

- 트랜잭션을 지원하는 쓰기 지연과 JDBC 배치

SQL을 직접 다루는 경우를 생각해보자.

```
insert(member1); //INSERT INTO ...
insert(member2); //INSERT INTO ...
insert(member3); //INSERT INTO ...
insert(member4); //INSERT INTO ...
insert(member5); //INSERT INTO ...

commit();
```

네트워크 호출 한 번은 단순한 메서드를 수만 번 호출하는 것보다 더 큰 비용이 든다. 이 코드는 5번의 INSERT SQL과 1번의 커밋으로 총 6번 데이터베이스와 통신한다. 이것을 최적화 하려면 5번의 INSERT SQL을 모아서 한 번에 데이터베이스로 보내면 된다.

JDBC가 제공하는 SQL 배치 기능을 사용하면 SQL을 모아서 데이터베이스에 한 번에 보낼 수 있다. 하지만 이 기능을 사용하려면 코드의 많은 부분을 수정해야 한다. 특히 비즈니스 로직이 복잡하게 얽혀 있는 곳에서 사용하기는 쉽지 않고 적용해도 코드가 상당히 지저분해진다. 그래서 보통은 수백 수천 건 이상의 데이터를 변경하는 특수한 상황에 SQL 배치 기능을 사용한다.

JPA는 플러시 기능이 있으므로 SQL 배치 기능을 효과적으로 사용할 수 있다.

참고로 SQL 배치 최적화 전략은 구현체마다 조금씩 다르다. 하이버네이트에서 SQL 배치를 적용하려면 다음과 같이 설정하면 된다. 이제부터 데이터를 등록, 수정, 삭제할 때 하이버네이트는 SQL 배치 기능을 사용한다.

```
<property name="hibernate.jdbc.batch_size" value="50"/>
```

값을 50으로 주면 최대 50건씩 모아서 SQL 배치를 실행한다. 하지만 SQL 배치는 같은 SQL일 때만 유효하다. 중간에 다른 처리가 들어가면 SQL 배치를 다시 시작한다. 예를 들어보자.

```
em.persist(new Member()); //1
em.persist(new Member()); //2
em.persist(new Member()); //3
em.persist(new Member()); //4
em.persist(new Child());  //5, 다른 연산
em.persist(new Member()); //6
em.persist(new Member()); //7
```

이렇게 하면 1,2,3,4를 모아서 하나의 SQL 배치를 실행하고 5를 한 번 실행하고 6,7을 모아서 실행한다. 따라서 총 3번 SQL 배치를 실행한다.

> **주의:** 엔티티가 영속 상태가 되려면 식별자가 꼭 필요하다. 그런데 IDENTITY 식별자 생성 전략은 엔티티를 데이터베이스에 저장해야 식별자를 구할 수 있으므로 `em.persist()`를 호출하는 즉시 INSERT SQL이 데이터베이스에 전달된다. 따라서 쓰기 지연을 활용한 성능 최적화를 할 수 없다.

- 트랜잭션을 지원하는 쓰기 지연과 애플리케이션 확장성

트랜잭션을 지원하는 쓰기 지연과 변경 감지 기능 덕분에 성능과 개발의 편의성이라는 두 마리 토끼를 모두 잡을 수 있었다. 하지만 진짜 장점은 **데이터베이스에 로우(row)에 락(lock)이 걸리는 시간을 최소화** 한다는 점이다.

이 기능은 트랜잭션을 커밋해서 영속성 컨텍스트를 플러시하기 전까지는 데이터베이스에 데이터를 등록, 수정, 삭제하지 않는다. 따라서 커밋 직전까지 데이터베이스 로우에 락을 걸지 않는다.

다음 로직을 보자.

```
update(memberA); // UPDATE SQL A
비즈니스로직A(); // UPDATE SQL ...
비즈니스로직B(); // INSERT SQL ...
commit();
```

JPA를 사용하지 않고 SQL을 직접 다루면 `update(memberA)`를 호출할 때 UPDATE SQL을 실행하면서 데이터베이스에 로우에 락을 건다. 이 락은 `비즈니스로직A()`, `비즈니스로직B()`를 모두 수행하고 `commit()`을 호출할 때까지 유지된다. 트랜잭션 격리 수준에 따라 다르지만 보통 많이 사용하는 커밋된 읽기(Read Committed)격리 수준이나 그 이상에서는 데이터베이스에 현재 수정 중인 데이터를 수정하려는 다른 트랜잭션은 락이 풀릴 때까지 대기한다.

JPA는 커밋을 해야 플러시를 호출하고 데이터베이스에 수정 쿼리를 보낸다. 쿼리를 보내고 바로 트랜잭션을 커밋하므로 데이터베이스에 락이 걸리는 시간을 최소화한다. 예제에서 `commit()`을 호출할 때 UPDATE SQL을 실행하고 바로 데이터베이스 트랜잭션을 커밋한다.

사용자가 증가하면 애플리케이션 서버를 더 증설하면 된다. 하지만 데이터베이스 락은 애플리케이션 서버 증설만으로는 해결할 수 없다. 오히려 애플리케이션 서버를 증설해서 트랜잭션이 증가할수록 더 많은 데이터베이스 락이 걸린다. JPA의 쓰기 지연 기능은 데이터베이스에 락이 걸리는 시간을 최소화해서 동시에 더 많은 트랜잭션을 처리할 수 있는 장점이 있다.

> **참고:** `update(memberA)`를 호출할 때 즉시 락을 걸고 싶다면 JPA가 제공하는 락 기능을 사용하면 된다.
