성능튜닝

Hibernate Performance Tuning Tips

https://thorben-janssen.com/tips-to-boost-your-hibernate-performance/

소개

- **JPA의 성능이슈**
  - 추상화 계층이 추가되므로 대용량 데이터베이스나 동시사용자가 많은 환경에서는 JPA가 성능을 떨어뜨릴 것이라 예상할 수 있음
  - 그러나 제대로 사용하면 높은 확장성과 유지보수성을 가질 수 있음
  - JPA기반 프로젝트에서 성능 이슈를 겪는 것과 그렇지 않은 것은 차이는?
  - JPA를 사용할 때 성능에 가장 큰 영향을 미치는 두 가지 실수
    - 로그 메시지 부재 및 오용
    - 불필요한 SQL을 유발하는 잘못된 사용

Find performance issues during development

- **Activate Hibernate's statistics component**
  - 통계 컴포넌트 활성화 후 메시지

```
07:03:29,976 DEBUG [org.hibernate.stat.internal.StatisticsImpl] - HHH000117: HQL: SELECT p FROM ChessPlayer
p LEFT JOIN FETCH p.gamesWhite LEFT JOIN FETCH p.gamesBlack ORDER BY p.id, time: 10ms, rows: 4
07:03:30,028 INFO  [org.hibernate.engine.internal.StatisticalLoggingSessionEventListener] - Session Metrics {
    46700 nanoseconds spent acquiring 1 JDBC connections;
    43700 nanoseconds spent releasing 1 JDBC connections;
    383099 nanoseconds spent preparing 5 JDBC statements;
    11505900 nanoseconds spent executing 4 JDBC statements;
    8895301 nanoseconds spent executing 1 JDBC batches;
   ....
}
```

  - 5.4.5버전부터 Logging되는 쿼리동작의 시간의 임계값을 지정할 수 있음(전부 보여주지 말고 실행이 느린 것만 보고 싶음)

Improve slow queries

- **Improve slow queries**
  - 파악된 문제가 반드시 JPA와 연관된 문제는 아님. 즉, 다른 persistence 프레임워크나 일반 SQL에서도 성능이슈는 발생
  - 네이티브 쿼리(JPA를 사용하지 않는)를 실행하여 발생한 문제의 원인이 JPA인지, 쿼리 그 자체인지를 판별

```
Author a = (Author) em.createNativeQuery("SELECT * FROM Author a WHERE a.id = 1", Author.class).getSingleResult();
```

  - 하이버네티이트는 네이티브 쿼리에 관여하지 않음
  - 단, Object[] 으로 결과값을 반환 받아야하는 번거로움이 존재

Avoid unnecessary queries

- **Choose the right FetchType**
  - 불필요한 쿼리 실행 최소화
  - 주로 패치 전략을 EAGER로 설정했을 때 발생 in JPQL(N+1문제)
  - FetchType.LAZY는 실제로 사용될 때까지 연관 엔티티를 조회하지 않음
  - FetchType.EAGER는 무조건 연관 엔티티를 초기화함
  - to-one 연관관계에서 FetchType.LAZY로 설정

```
@ManyToOne(fetch=FetchType.LAZY)
```

  - to-many 연관관계의 기본 FetchType은 LAZY
  - 단 어차피 연관관계 엔티티도 사용한다면 한 번에 조회하는 것이 성능상 좋음

Avoid unnecessary queries

- **Use query-specific fetching**
  - LAZY전략은 분리된 쿼리를 실행하는 단점: 데이터베이스(테이블) 점유 시간, 네트워크 비용 ..
  - 이 또한 N+1문제를 야기함. 즉, 패치 전략을 통해 N+1문제를 해결할 수는 없음

```
List<Author> authors = em.createQuery("SELECT a FROM Author a", Author.class).getResultList();
for (Author author : authors) {
    log.info(author + " has written " + author.getBooks().size() + " books.");
}
```

  - 한 명의 작가에 연관된 책이 11권 있다고하면, 총 12번의 select쿼리가 실행
  - 해결책
    - EAGER를 사용
    - 다른 방법

Avoid unnecessary queries

- **Use a JOIN FETCH clause**
  - 대상 엔티티를 조회할 때 연관 엔티티도 함께 조회
  - Fetch 타입과 상관 없음

```
List<Author> authors = em.createQuery("SELECT a FROM Author a JOIN FETCH a.books b", Author.class).
                                                    getResultList();
```

  - DISTINCT 키워드 사용 가능
    - 위 예시에서는 저자가 중복적으로 fetch될 수 있음

```
List<Author> authors = em.createQuery("SELECT DISTINCT a FROM Author a JOIN FETCH a.books b", Author.class)
                        .setHint(QueryHints.PASS_DISTINCT_THROUGH, false)
                        .getResultList();
```

Avoid unnecessary queries

- **Use a @NamedEntityGraph**
  - 엔티티 조회시점에 연관된 엔티티들을 함께 조회하는 기능
  - 정적, 동적으로 정의할 수 있음
  - Join Fetch는 Inner Join, Entity Graph는 Outer Join이라는 차이점이 있음

```
@NamedEntityGraph(name = "graph.AuthorBooks",  attributeNodes = @NamedAttributeNode(value = "books"))
```

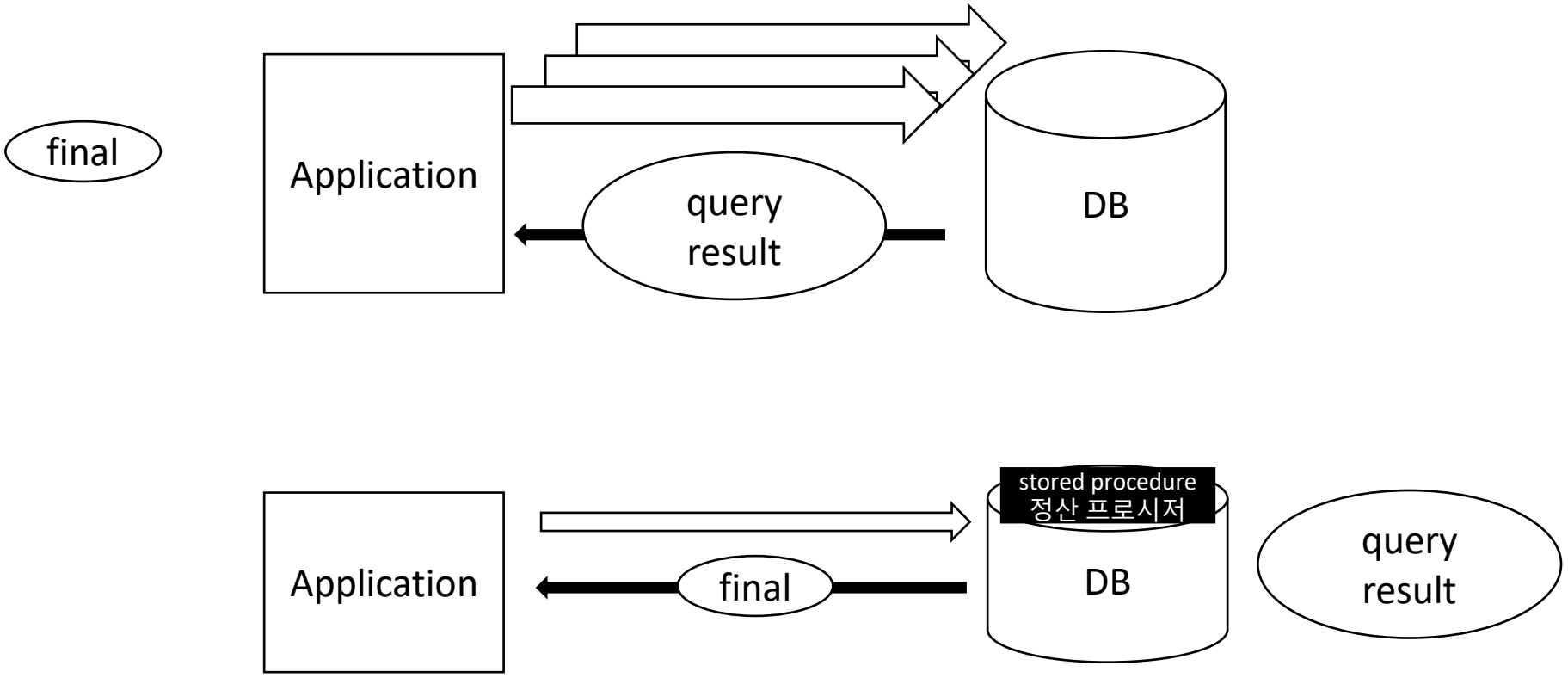
- **Use an EntityGraph**
  - 동적으로 엔티티 그래프를 동적으로 구성

```
EntityGraph graph = em.createEntityGraph(Author.class);
Subgraph bookSubGraph = graph.addSubgraph(Author_.books);

List<Author> authors = em
        .createQuery("SELECT a FROM Author a", Author.class)
        .setHint(QueryHints.JAKARTA_HINT_FETCH_GRAPH, graph)
        .getResultList();+
```

Let the database handle data-heavy operations

- **Heavy 데이터 처리 주체 변경**
  - Java코드: 프로그램 상에서 데이터 관리
  - DB: 비즈니스 로직이 DB에서 수행(stored procedure)



Let the database handle data-heavy operations

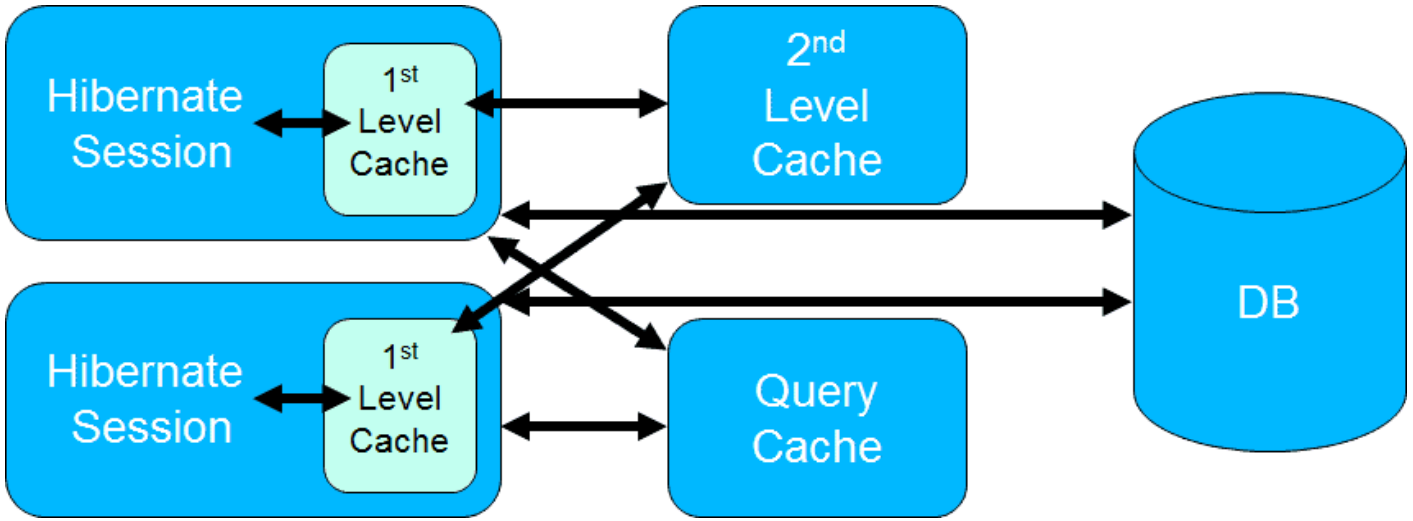
- **Heavy 데이터 처리 주체**
  - @NamedStoredProcedure

```
@NamedStoredProcedureQuery(
  name = "getBooks",
  procedureName = "get_books",
  resultClasses = Book.class,
  parameters = { @StoredProcedureParameter(mode = ParameterMode.REF_CURSOR, type = void.class) }
)
```

```
List<Book> books = (List<Book>) em.createNamedStoredProcedureQuery("getBooks").getResultList();
```

Use caches to avoid reading the same data multiple times

- **Cache**
  - 하이버네이트는 세 가지의 cache를 제공



  - 1st Level Cache: 기본적으로 활성화되어 있음. 동일한 세션(트랜잭션)에서는 한 번 호출된 엔티티를 재사용
  - 2nd Level Cache: 트랜잭션이 종료되더라도 조회 엔티티 유지(동시성 문제가 발생할 수 있으므로 유의)
  - Query Cache: 쿼리 결과와 더불어 엔티티 참조와 scalar 값을 저장

Perform updates and deletes in bulks

- **bulk연산**
  - Updating , deleting을 한 번에 처리
  - CriteriaUpdate and CriteriaDelete

```
CriteriaBuilder cb = this.em.getCriteriaBuilder();

// create update
CriteriaUpdate<Order> update = cb.createCriteriaUpdate(Order.class);

// set the root class
Root e = update.from(Order.class);

// set update and where clause
update.set("amount", newAmount);
update.where(cb.greaterThanOrEqualTo(e.get("amount"), oldAmount));

// perform update
this.em.createQuery(update).executeUpdate();
```

Conclusion

- **정리**
  - JPA 사용시 고려해야 할 여러 성능이슈가 존재
  - Hibernate statistics를 통한 쿼리 실행 파악이 중요
  - 적절한 FetchType 사용이 필요
  - 불필요하게 발생하는 추가 쿼리 실행을 회피
    - EAGER
    - JOIN FETCH
    - Entity Graph
  - +전통적인 데이터베이스 쿼리 튜닝 상기
    - ex) index, covering index