

# Optimizasyon Yöntemleri (Fetch Stratejileri)



# N+1 SELECT Problemi

- Genellikle **lazy** 1:M ve M:N ilişkilerde ortaya çıkar

```
Query query = session.createQuery("from Owner");
List<Owner> owners = query.list();

for(Owner owner:owners) {
    int petsSize = owner.getPets().size();
    //...
}
```

Owner'ları çekmek için DB'de bir SELECT çalıştırılır

Her Owner'ın LAZY pets collection'ına erişildiği vakit, bu collection'ının da yüklenmesi için DB'de ayrı bir SELECT daha çalıştırılır. Bu döngü içinde iterate edilen bütün Owner'lar için tekrarlanır

Sonuç 1 + N SELECT'tir

- Fetch stratejileri** bu probleme çözüm üretmeye çalışır

# N+1 SELECT Problemine Çözüm :Fetch Stratejileri

- Fetch stratejilerinde **amaç SELECT sayısını en aza indirmektir**
- Değişik fetch stratejileri mevcuttur
  - Fetch Join
  - Batch Fetching
  - Sub-select ile Pre-Fetch
  - Select ile Eager Fetch (JOIN'in devre dışı bırakılması gereken durumlarda kullanılır)

# Fetch Join

- HQL'de join ifadelerine **FETCH** anahtar kelimesi eklenerek **lazy** ilişkinin **eager** yüklenmesi sağlanır

```
from Owner o left join fetch o.pets
where o.firstName like '%J%'
```

```
from Owner o left join fetch o.pets p
left join fetch p.type
left join fetch p.visits v
where o.firstName like '%J%'
```

# Fetch Join

- Fetch join'in **amacı performans optimizasyonudur**
- Fetch-join yapılan bir ilişkiler hiçbir zaman için **alias** yapıp **projeksiyonda** kullanılamaz
- Yada **with** ifadesi ile bu alias üzerinden join condition uygulanamaz
- Bu durumda ***left join fetch o.pets p with p.name*** = ... **geçersiz** bir sorgu parçacığdır

# Fetch Join

- Ancak fetch join yapılan ilişki üzerinde **where clause'da kısıtlamaya gidilebilir**, ancak dönen collection'ların içeriğinde de bu kısıtlamaya göre ilgili Pet nesneleri yer **almayacaktır**
- LEFT anahtar kelimesi **olmadan JOIN FETCH** kullanılırsa eager fetch “**inner join**” ile yapılır
- Bu durumda örneğin **Pet nesnesi içermeyen Owner nesneleri** döndürülmez
- **Zorunlu (optional=false) M:1 ilişkiler** de kullanılabilir

# Fetch Join ve Duplikasyon

- Eğer bir collection **eager fetch** yapılırsa, sorgu sonuçlarında **duplikasyon** söz konusu olabilir
- Tekrarlayan bu nesneleri filtrelemek için
  - ya dönen List nesnesi LinkedHashSet ile wrap edilmelidir
  - ya da select ifadesinde DISTINCT ifadesi kullanılmalıdır

***select distinct o from Owner o join fetch o.pets***

# Hibernate Criteria API ve Fetch Join

- Hibernate **Criteria API** ile çalıştırılan **sorgularda** da fetch join yapılabilir
- Bunun için **Criteria.setFetchMode()** metodu kullanılır

```
session.createCriteria(Pet.class)  
    .setFetchMode("visits", FetchMode.JOIN)  
    .add(Restrictions.like("name", "%Kitty%"));
```

FetchMode.JOIN değeri, HQL deki left join fetch'e karşılık gelir



# Hibernate Criteria API ve Fetch Join

- Criteria API ile çalışırken fetch join'in **INNER JOIN** ile yapılması da sağlanabilir

```
session.createCriteria(Pet.class)  
    .createAlias("visits", "v",  
                CriteriaSpecification.INNER_JOIN)  
    .setFetchMode("v", FetchMode.JOIN)  
    .add(Restrictions.like(  
        "v.description", "%Foo%"));
```

visits ilişkisi ile inner join yapılacağını belirtir

setFetchMode() ifadesi de fetch join kabiliyetinin uygulanacağını belirtir

# JPA Criteria API ve Fetch Join

```
CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);  
Root<Pet> pet = cq.from(Pet.class);  
  
pet.fetch("owner");  
  
pet.fetch("visits", JoinType.LEFT);  
  
cq.distinct(true);  
  
TypedQuery<Pet> tq = em.createQuery(cq);  
List<Pet> resultList = tq.getResultList();
```

Query root üzerinde fetch() metodu ile fetch join'ler tanımlanabilir  
Default olarak fetch INNER join yapar  
JoinType belirterek LEFT OUTER join de yaptırılabilir

CriteriaQuery.distinct() metodu ile Join sonucu ortaya çıkan duplikasyonlar da elemine edilebilir

# Batch Fetching

- Aynı anda birkaç LAZY ilişkinin **tek bir SELECT ile initialize** edilmesini sağlar
- **Entity** veya **collection düzeyinde** tanımlanabilir
- **@BatchSize** anotasyonu ile tanımlanır
- Örneğin lazy bir ilişki üzerinde **@BatchSize(size=10)** tanımlandığında N+1 select yerine **N/10+1** select çalıştırılacaktır
- Devreye girebilmesi için ana sorgunun yapıldığı **Session'ı açık olması** gerekir

# Entity Düzeyinde Batch Fetching Tanımı

- M:1 ve 1:1 LAZY tanımlanan ilişkilerdeki entity'lerin topluca çekilmesi için tanımlanır

Entity  
düzeyinde  
tanım

```
@Entity
@org.hibernate.annotations.BatchSize(size=5)
public class Owner {
}
```

Owner entity'sinin target entity olduğu M:1 ve 1:1 LAZY ilişkilerde devreye girecektir. Örneğin, Pet entity'lerini select eden bir sorgu çalıştırıldığında Hibernate M:1 LAZY Owner entity'lerini batch size değerine göre toplu biçimde tek sorgu ile yüklemeye çalışacaktır

```
@Entity
public class Pet {
```

```
@ManyToOne(fetch = FetchType.LAZY)
private Owner owner;
```

```
}
```

# Collection Düzeyinde Batch Fetching Tanımı

- 1:M ve N:M **LAZY** ilişkili entity'leri topluca yüklemek için tanımlanır

```
@Entity
public class Pet {

    @OneToMany
    @org.hibernate.annotations.BatchSize(size = 5)
    private Set<Visit> visits = new HashSet<Visit>();

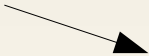
}
```

Collection  
düzeyinde  
tanım

Pet entity'lerini select eden sorgu sırasında batch size kadar Pet entity'sinin visits ilişkilerini de initialize edecek bir sorgu çalışmasını sağlar

# Batch Fetching Kullanım Örneği

```
List<Pet> pets = s.createQuery("from Pet", Pet.class)
                        .getResultList();
```



**select p.\* from PETS**

```
pets.forEach(pet->{
    System.out.println(pet.getOwner().getFirstName());
    System.out.println(pet.getVisits().size());
});
```

for döngüsüne girildiğinde ilk pet'in Owner ve Visit ilişkisine erişildiği vakit aşağıdaki sorgular çalıştırılacaktır

```
select o.* from OWNERS o where o.ID in (?, ?, ?, ?, ?)
select v.* from VISITS v where v.PET_ID in (?, ?, ?, ?, ?)
```

# SubSelect ile Pre-Fetch

- Sadece **Collection** tipli ilişkilerde kullanılabilir
- **@Fetch(FetchMode.SUBSELECT)** anotasyonu ile tanımlanır
- Lazy ilişkinin ait olduğu entity'lerin yüklenmesinin ardından ilişkiye ilk erişimde yüklenen bütün entity'lerin lazy ilişkileri de ayrı bir **subselect sorgu ile** yüklenir
- Devreye girebilmesi için ana sorgunun yapıldığı **Session'ı açık olması** gerekir

# SubSelect ile Pre-Fetch

Bir collection initialize edilirken yüklü bütün ana nesnelerin ilişkili collection'ları da initialize edilir

```
@Entity
public class Pet {

    @OneToMany
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SUBSELECT)
    private Set<Visit> visits = new HashSet<Visit>();

}
```

Sadece collection'lar için geçerlidir  
Sorgu sayısı 2 ye düşer

```
select p.* from PETS p
select v.* from VISITS v
where v.PET_ID in (select p.ID from PETS p)
```



# SELECT ile Eager Fetch

- PersistenceContext'in **PK üzerinden yükleme yapan metotlar** (get/load), EAGER ilişkileri **JOIN**'leyerek getirir
- **Criteria API** sorguları da EAGER ilişkileri **JOIN**'leyerek getirir
- **HQL** sorguları ise EAGER ilişkileri **ayrı SELECT** ifadeleri ile getirir

# SELECT ile Eager Fetch

- Normalde **JOIN** ile çekilecek **EAGER** bir ilişkinin **ayrı bir SELECT** ile yüklenmesini sağlar

```
@Entity
public class Pet {
    @ManyToOne(fetch = FetchType.EAGER)
    @Fetch(FetchMode.SELECT)
    private PetType type;
}
```

- Optimizasyondan çok bazı durumlarda (örneğin bir SQL ifadesindeki **max JOIN tablo sayısının aşılması** durumu) kullanışlıdır

# İletişim

- Harezmi Bilişim Çözümleri
- Kurumsal Java Eğitimleri
- <http://www.java-egitimleri.com>
- [info@java-egitimleri.com](mailto:info@java-egitimleri.com)

