ORM PERFORMANCE

Performance Considerations

- Object Relational Mapping tools provide an increase in productivity
- Removes the necessity of “handcrafting” boilerplate** SQL code.

HOWEVER

- Lack of “insight” into the interactions with the database can reduce performance
- Without proper consideration of performance, scalability is also restricted.

On the other hand

- A good ORM Framework has resources available for optimization.
- With the right approach, you can improve performance **BUT**

Ultimately You Need a DBA Expert

Someone on the team who understands how a RDB works [Database design].

Knowing SQL – is not database design

** rolled steel for making boilers

Approach to Performance

You should not do preemptive optimization

Within reason, you should not care about optimization unless you **need** to. You need to **after** you have implemented, done measurements and found issues.

You should identify the scenarios that cause problems and **then** optimize to fix them.

Obviously the focus of improvement is fetching relationships.

Obviously

The default Fetch strategy is fetchType.LAZY

Although

Better performance comes from eager loading relationships that are always used

REMEMBER:

Implementation involves functionality & code stability

Performance tuning comes AFTER

Hibernate FETCH Strategy

- ***Hibernate NOT JPA***
- ***Select fetching*:** a second SELECT [per parent N] is used to retrieve the associated collection. [***DEFAULT***] [N+1 Fetches] `@Fetch(FetchMode.SELECT)`
- ***Join fetching*:** associated collections are retrieved in the same SELECT, using an OUTER JOIN. [1 Fetch] [***EAGER***] `@Fetch(FetchMode.JOIN)`
- ***Subselect fetching*:** a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. [2 Fetches]
- `@Fetch(FetchMode.SUBSELECT)`
- ***Batch fetching*:** Optimization of Select Fetching. Associated collections are fetched according to declared Batch Size(N/Batch Size) + 1; **@BatchSize(size=n)**
- **EXAMPLE:**
- `@Fetch(FetchMode.SUBSELECT)`
- `private Set<Address> addresses;`

Hibernate Fetch Strategy Issues

Select

- N+1 TBA [To Be Avoided]
- **Join**

  Cartesian – need to watch collection sizes; can be useful strategy
- **SubSelect**

  depends on the "parent" query. If parent Query is complex, it could have performance impacts.

  If `fetch=FetchType.`***LAZY*** `need to "hydrate" children`
- **BatchSize**

  # of Fetches "unknown" UNLESS size of parent is constant

  Batch fetching is often called a blind-guess optimization

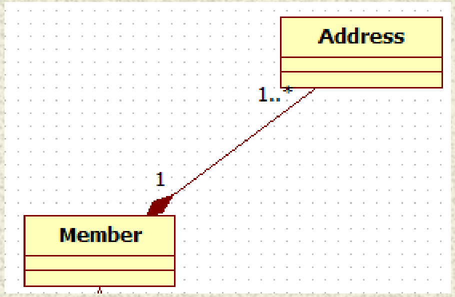
  If `fetch=FetchType.`***LAZY*** `need to "hydrate" children`

REMEMBER: `fetch=FetchType.`***LAZY*** is recommended Default

Example N = 3

N+1 Problem

- Member has a OneToMany relationship with Address



Uses “Default” strategy of SELECT

- Declared as Fetch EAGER:
- `@OneToMany(mappedBy="member",fetch=FetchType.EAGER)`
- `private Set<Address> addresses = new HashSet<Address>();`

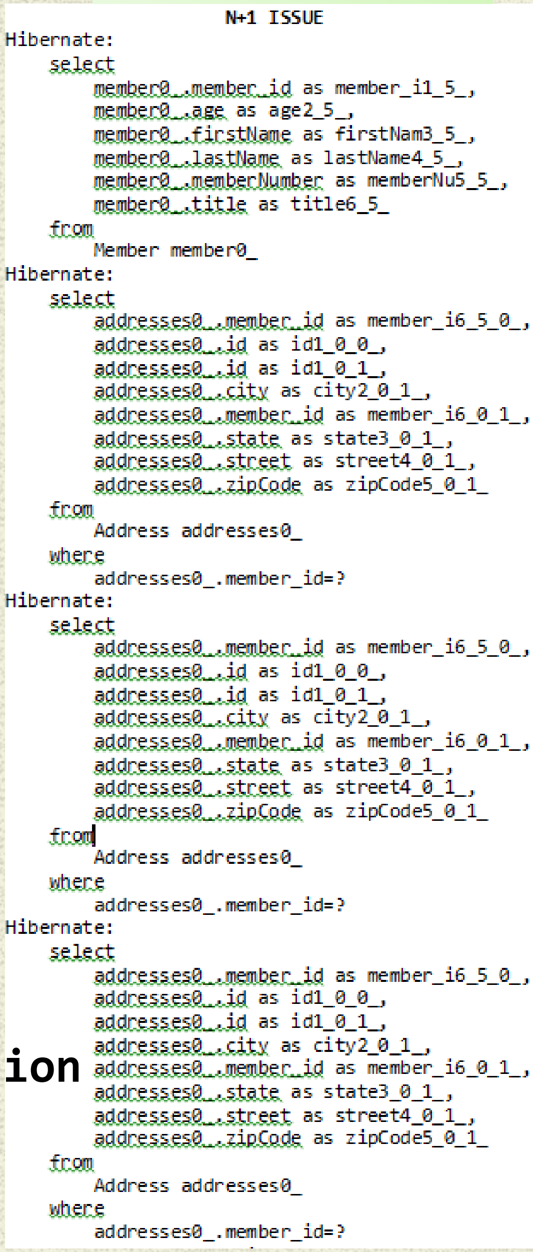
- For
- `entityManager.createQuery( "from Member”)`
- `        .getResultList();`

ORM will issue ONE fetch for ALL the Members

&

And N fetches; one for each member’s Address Collection

- So for 3 members we have ONE fetch for Members and THREE fetches for Address collections



```
                  N+1 ISSUE
Hibernate:
    select
        member0_.member_id as member_i1_5_,
        member0_.age as age2_5_,
        member0_.firstName as firstNam3_5_,
        member0_.lastName as lastName4_5_,
        member0_.memberNumber as memberNu5_5_,
        member0_.title as title6_5_
    from
        Member member0_
Hibernate:
    select
        addresses0_.member_id as member_i6_5_0_,
        addresses0_.id as id1_0_0_,
        addresses0_.id as id1_0_1_,
        addresses0_.city as city2_0_1_,
        addresses0_.member_id as member_i6_0_1_,
        addresses0_.state as state3_0_1_,
        addresses0_.street as street4_0_1_,
        addresses0_.zipCode as zipCode5_0_1_
    from
        Address addresses0_
    where
        addresses0_.member_id=?
Hibernate:
    select
        addresses0_.member_id as member_i6_5_0_,
        addresses0_.id as id1_0_0_,
        addresses0_.id as id1_0_1_,
        addresses0_.city as city2_0_1_,
        addresses0_.member_id as member_i6_0_1_,
        addresses0_.state as state3_0_1_,
        addresses0_.street as street4_0_1_,
        addresses0_.zipCode as zipCode5_0_1_
    from
        Address addresses0_
    where
        addresses0_.member_id=?
Hibernate:
    select
        addresses0_.member_id as member_i6_5_0_,
        addresses0_.id as id1_0_0_,
        addresses0_.id as id1_0_1_,
        addresses0_.city as city2_0_1_,
        addresses0_.member_id as member_i6_0_1_,
        addresses0_.state as state3_0_1_,
        addresses0_.street as street4_0_1_,
        addresses0_.zipCode as zipCode5_0_1_
    from
        Address addresses0_
    where
        addresses0_.member_id=?
```

Join Fetching Strategy

Hibernate Annotation: `@Fetch(FetchMode.JOIN)`

ALWAYS

Causes an ***EAGER*** fetch of the child collections

This is because the characteristic of a Join is ONE fetch

Parent & Child TOGETHER

This can only be accomplished by loading the child collection when the parent is fetched [~= **EAGER** fetch]

Best practice: Use Lazy Initialization [NOT Eager]

So we will implement the Join Fetch “Manually”

Join == Cartesian “LIKE” Product Problem

- For sets A and B, the Cartesian product is A × B
- For sets A,B and C, the Cartesian product is A × B × C - etc.

Member has a OneToMany relationship with Address

```
@OneToMany(mappedBy="member",fetch=FetchType.LAZY)
private Set<Address> addresses = new HashSet<Address>();
```

For:

```
Query query=entityManager.createQuery("SELECT m
        FROM Member AS m JOIN FETCH m.addresses AS a");
```

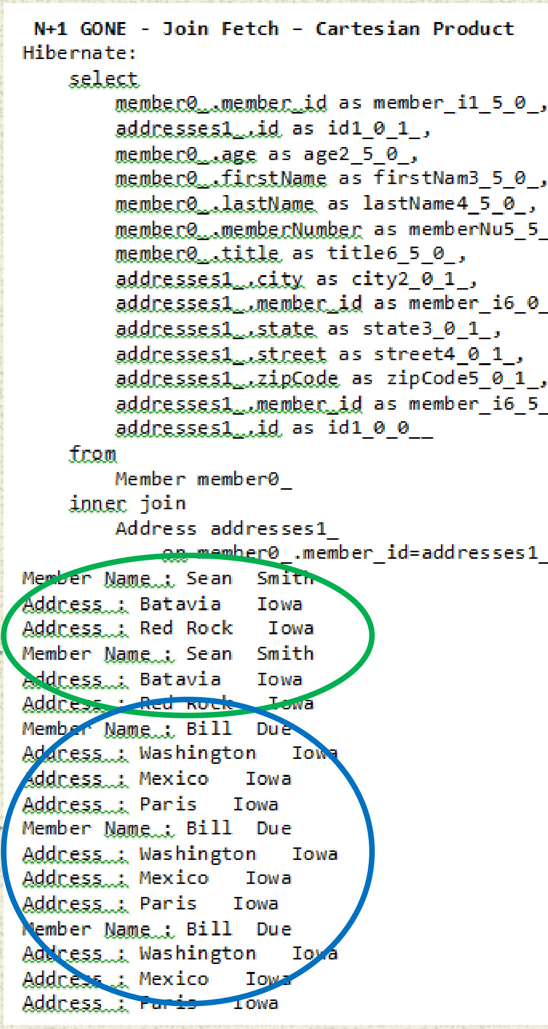
ORM will do *ONE FETCH* BUT will generate duplicates

]

ORM NOTE: “Product” =

∑ # Addresses [per Member.

Sean has 2 Addresses so 2 copies ; Bill has 3 so 3 copies



```
 N+1 GONE - Join Fetch - Cartesian Product
Hibernate:
    select
        member0_.member_id as member_i1_5_0_,
        addresses1_.id as id1_0_1_,
        member0_.age as age2_5_0_,
        member0_.firstName as firstNam3_5_0_,
        member0_.lastName as lastName4_5_0_,
        member0_.memberNumber as memberNu5_5_
        member0_.title as title6_5_0_,
        addresses1_.city as city2_0_1_,
        addresses1_.member_id as member_i6_0_
        addresses1_.state as state3_0_1_,
        addresses1_.street as street4_0_1_,
        addresses1_.zipCode as zipCode5_0_1_,
        addresses1_.member_id as member_i6_5_
        addresses1_.id as id1_0_0__
    from
        Member member0_
    inner join
        Address addresses1_
            on member0_.member_id=addresses1_
Member Name : Sean  Smith
Address : Batavia   Iowa
Address : Red Rock   Iowa
Member Name : Sean  Smith
Address : Batavia   Iowa
Address : Red Rock   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
```

Join == Cartesian Product Problem

For sets A and B, the Cartesian product is A × B

For sets A,B and C, the Cartesian product is A × B × C - etc.

Member has a OneToMany relationship with Address & Order

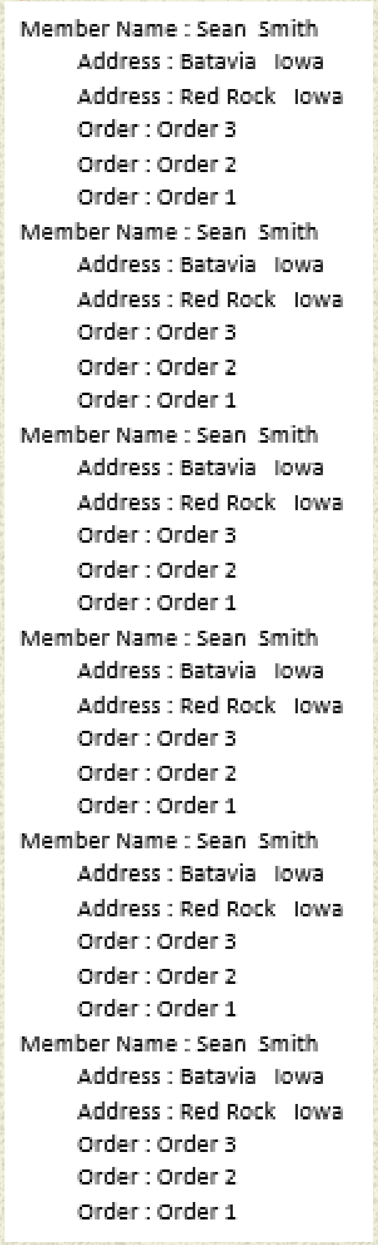
```
@OneToMany(mappedBy="member",fetch=FetchType.LAZY)
private Set<Address> addresses = new HashSet<Address>();
@OneToMany(mappedBy="member",fetch=FetchType.LAZY)
private Set<Order> orders = new HashSet<Order>();
```

For:

```
Query query =  entityManager.createQuery("SELECT m FROM Member
           AS m JOIN FETCH m.orders as o
                JOIN FETCH m.addresses as a where m.id = :id");
ORM will do ONE FETCH BUT will generate duplicates
```

ORM NOTE: Product = 2 Addresses X 3 Order

SIX Members



```
Member Name : Sean  Smith
     Address : Batavia  Iowa
     Address : Red Rock  Iowa
     Order : Order 3
     Order : Order 2
     Order : Order 1
Member Name : Sean  Smith
     Address : Batavia  Iowa
     Address : Red Rock  Iowa
     Order : Order 3
     Order : Order 2
     Order : Order 1
Member Name : Sean  Smith
     Address : Batavia  Iowa
     Address : Red Rock  Iowa
     Order : Order 3
     Order : Order 2
     Order : Order 1
Member Name : Sean  Smith
     Address : Batavia  Iowa
     Address : Red Rock  Iowa
     Order : Order 3
     Order : Order 2
     Order : Order 1
Member Name : Sean  Smith
     Address : Batavia  Iowa
     Address : Red Rock  Iowa
     Order : Order 3
     Order : Order 2
     Order : Order 1
Member Name : Sean  Smith
     Address : Batavia  Iowa
     Address : Red Rock  Iowa
     Order : Order 3
     Order : Order 2
     Order : Order 1
```

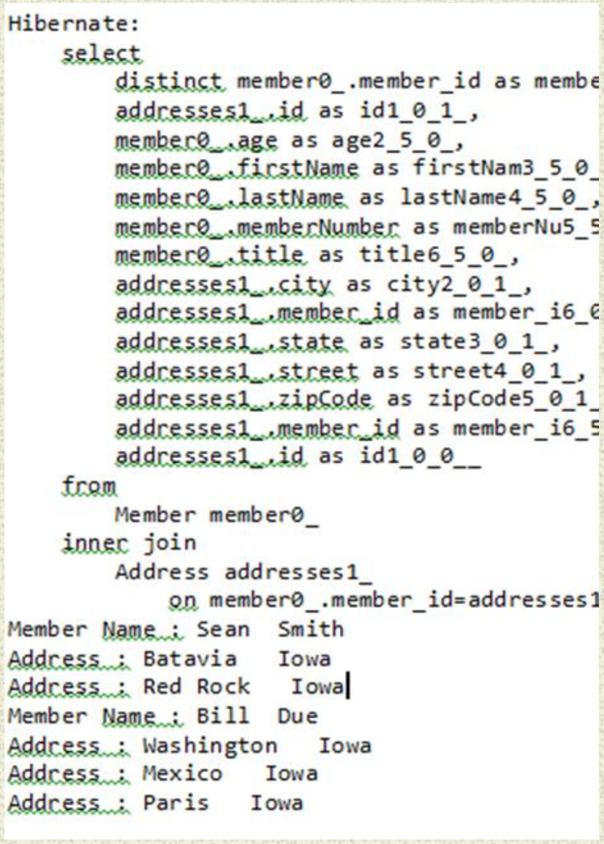
Clean Up
The Cartesian Product Data

-
  ```
  Query query=entityManager.createQuery("SELECT DISTINCT m
        FROM Member AS m JOIN FETCH m.addresses AS a");
  ```

- DISTINCT keyword removes duplicates

However

It accomplishes it in Memory [After DB fetch]



```
Hibernate:
    select
        distinct member0_.member_id as membe
        addresses1_.id as id1_0_1_,
        member0_.age as age2_5_0_,
        member0_.firstName as firstNam3_5_0_
        member0_.lastName as lastName4_5_0_,
        member0_.memberNumber as memberNu5_5
        member0_.title as title6_5_0_,
        addresses1_.city as city2_0_1_,
        addresses1_.member_id as member_i6_0
        addresses1_.state as state3_0_1_,
        addresses1_.street as street4_0_1_,
        addresses1_.zipCode as zipCode5_0_1_
        addresses1_.member_id as member_i6_5
        addresses1_.id as id1_0_0__
    from
        Member member0_
    inner join
        Address addresses1_
            on member0_.member_id=addresses1
Member Name : Sean  Smith
Address : Batavia   Iowa
Address : Red Rock   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
```

SubSelect Fetch

- **Declared as Fetch LAZY:**
- `@OneToMany(mappedBy="member",fetch=FetchType.LAZY)`
- `@Fetch(FetchMode.SUBSELECT)`
- `private Set<Address> addresses = new HashSet<Address>();`

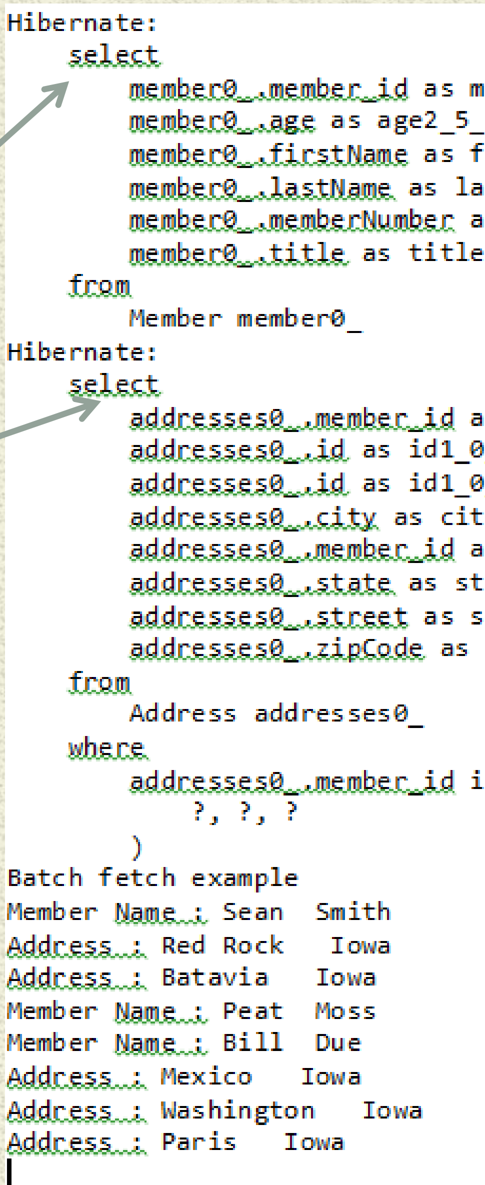
```
For FetchMode.SUBSELECT
 ORM will do ONE Fetch for All Parents
 ORM will do ONE Fetch for All child collections
```

- **Need to "Hydrate" collections:**

```
List<Member> members =  (List<Member>)this.findAll();
members.get(0).getAddresses().get(0);
```

```
We can also do this "Manually" WITHOUT
                              @Fetch(FetchMode.SUBSELECT)
```

See DEMO FetchSubSelect



```
Hibernate:
    select
        member0_.member_id as m
        member0_.age as age2_5_
        member0_.firstName as f
        member0_.lastName as la
        member0_.memberNumber a
        member0_.title as title
    from
        Member member0_
Hibernate:
    select
        addresses0_.member_id a
        addresses0_.id as id1_0
        addresses0_.id as id1_0
        addresses0_.city as cit
        addresses0_.member_id a
        addresses0_.state as st
        addresses0_.street as s
        addresses0_.zipCode as
    from
        Address addresses0_
    where
        addresses0_.member_id i
            ?, ?, ?
        )
Batch fetch example
Member Name : Sean  Smith
Address : Red Rock  Iowa
Address : Batavia  Iowa
Member Name : Peat  Moss
Member Name : Bill  Due
Address : Mexico  Iowa
Address : Washington  Iowa
Address : Paris  Iowa
```

Batch Size Fetch

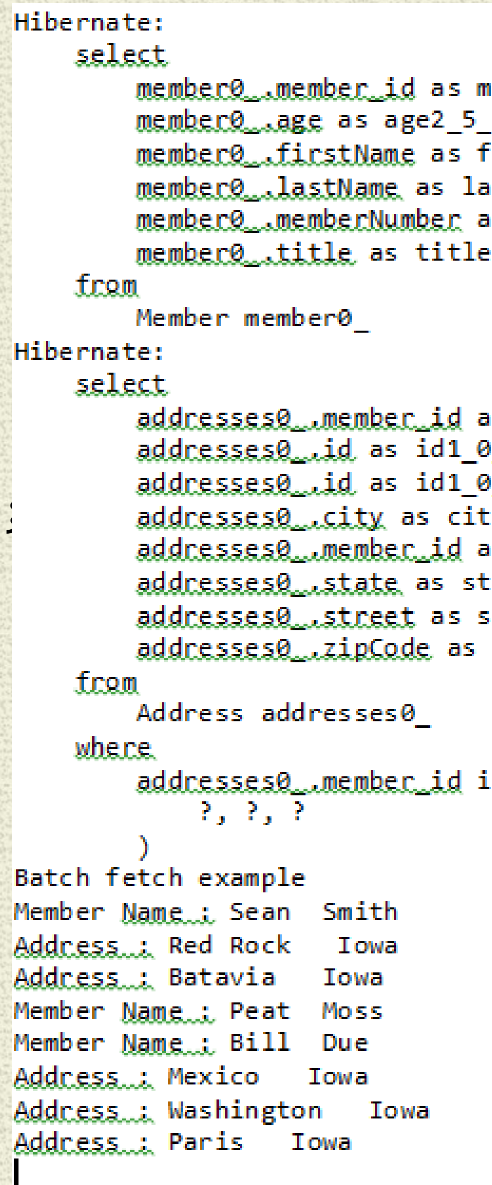
- **Declared as Fetch LAZY:**
- `@OneToMany(mappedBy="member",fetch=FetchType.LAZY)`
- `@Fetch(FetchMode.SELECT)`
- `@BatchSize(size = 3)`
- `private Set<Address> addresses = new HashSet<Address>();`
- **Need to "Hydrate" collections:**
- `List<Member> members =  (List<Member>)this.findAll();`
- `for (Member member : members)`
- `        if (!member.getAddresses().isEmpty())`
- `                member.getAddresses().get(0);`
- **In example, ORM will do ONE Collection Fetch [based on batch size = # parents]**

NOTE:

In example # members = 3; BatchSize = 3; 1 Collection fetch

If # members = 3; BatchSize = 2; 2 Collection fetches

If # members = 3; BatchSize = 1; 3 Collection fetches



```
Hibernate:
    select
        member0_.member_id as m
        member0_.age as age2_5_
        member0_.firstName as f
        member0_.lastName as la
        member0_.memberNumber a
        member0_.title as title
    from
        Member member0_
Hibernate:
    select
        addresses0_.member_id a
        addresses0_.id as id1_0
        addresses0_.id as id1_0
        addresses0_.city as cit
        addresses0_.member_id a
        addresses0_.state as st
        addresses0_.street as s
        addresses0_.zipCode as
    from
        Address addresses0_
    where
        addresses0_.member_id i
            ?, ?, ?
        )
Batch fetch example
Member Name : Sean  Smith
Address : Red Rock   Iowa
Address : Batavia   Iowa
Member Name : Peat  Moss
Member Name : Bill  Due
Address : Mexico   Iowa
Address : Washington   Iowa
Address : Paris   Iowa
```

Customized Fetching Strategies

HOW to Access an Association [for Performance Tuning]

WHEN is **FetchType.*LAZY*** OR FetchType.***EAGER***

The Default Fetching Strategy

is not used to customize fetching [it is not efficient]

Use & Master The Query API

to override the Default Fetching Strategy for custom fetching

Understand & Use The Join fetch & SubSelect

Avoid the N+1 fetch problem as a rule of thumb

`FetchMode.`***SELECT***(Default) is extremely vulnerable to the N+1 problem

Use Lazy Initialization

- ***Regardless of your fetching strategy WHEN you fetch takes "priority"***
- *** Non-lazy initialization [FetchType.***EAGER***] ***always*** loads the relationship ***

REMEMBER - Optimize Selectively

Main Point

- An eager fetch strategy often creates inefficient queries. To increase performance we can configure the entity associations with a lazy fetch strategy and fetch them in a more efficient manner.

- ***Science of Consciousness:*** *As we grow in creative intelligence we enjoy the advantage of increased efficient action..*

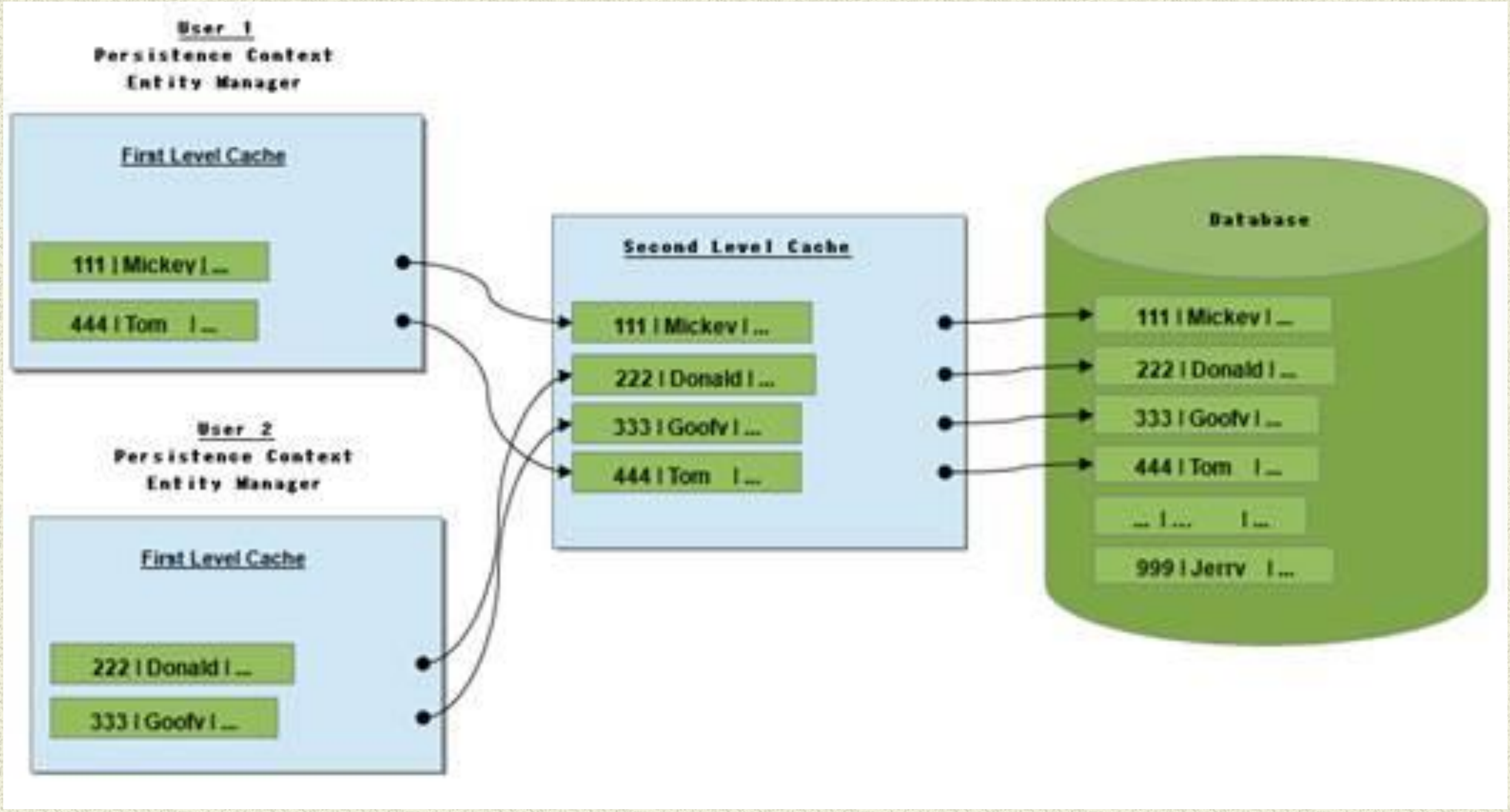
Second Level Cache

- A **second-level cache**

  A local store of entity data managed by the persistence provider to improve application performance

- Improves performance by avoiding expensive database calls
- Keeps the entity data local to the application
- Transparent to the application
- Available across all users [Application wide]
- Complements First level cache

Level 2 Cache



Query for Entity

```
• Check First level Cache
• If Found:
        Return Entity
• If not Found:
        Check Second level Cache
        If Found:
                Update First Level Cache
                Return Entity
        If not Found:
                Execute DB Query
                Update Second Level Cache
                Update First Level Cache
                Return Entity
```

Cache Concurrency Strategy

- **Transactional:**

  R**ead-Mostly** data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.

  Full Lock on all entities in transaction. Performance issue.
- **Read-write:**

  Read-Mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.

  Supports Repeatable Read Isolation - Allows phantom reads
- **Nonstrict-read-write:**

  No guarantee of consistency between the cache and the database. Use this strategy if data hardly ever changes and a small likelihood of stale data is not of critical concern.

Read-only:

Suitable for data which never changes. Use it for reference data only.

Simplest and optimal performing strategy.

READ Only – Low Hanging Fruit

- **Read-only caches are easy to handle**

  It is immutable (Modification Forbidden)

  No consistency issues.

  Always a good candidate for second level caching

- **Read-write caches are more “subtle” in their behavior.**
- Interaction with the Hibernate session can lead to unwanted behavior.
- The benefits of the C in ACID are compromised if cache is out of sync with DB

  Eventual consistency is NOT a primary use case of Relational DBs in an Enterprise.

-

Second Level Cache Decision

Database queries slow?

Second Level Cache is the *final resort*

- **Optimize ORM Queries**
  - Make sure the fetching strategy is properly designed,
  - Remove N+1 query problems
  - Involve the DBA [Expert]
  - Employ indexes
  - Investigate data base solutions [e.g. paritioning]

If performance is still an issue **THEN** consider a second level cache

Two Types of Cache

1. ***Second Level Cache***

   Second level cache is a key-value store.

   Applicable for accessing an entity by Primary Key

   These will hit Second Level Cache

   ```
   member = memberService.findOne(1L);
   entityManager.createQuery("SELECT m FROM Member m where
                 m.id= :member");
   ```

   These will NOT hit cache

   ```
   entityManager.createQuery("SELECT  m FROM Member AS m JOIN FETCH
                 m.addresses AS a where m.id= :member");
   entityManager.createQuery("select m from Member m  where
                 m.memberNumber =:number");
   ```

2. **Query Cache**

   Applicable for accessing entities by specific query

   The two preceding queries are Query Cache “candidates”

NOTE: Query cache store the query in query cache and the actually entities in the Second Level Cache

Configuring resources for Caching

For Second Level Cache:

```
@Cache(usage = org.hibernate.annotations.CacheConcurrencyStrategy.READ_ONLY)
public class Member {
```

For Query Cache:

```
@NamedQuery(name="Member.findById",
          query="select m from Member m where m.memberNumber = :memberNumber ",
          hints={ @QueryHint(name="org.hibernate.cacheable", value="true") } )
```

Configuration

- **pom.xml**

```
<dependency>
     <groupId>net.sf.ehcache</groupId>
     <artifactId>ehcache</artifactId>
 </dependency>
 <dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-ehcache</artifactId>
 </dependency>
```

- **Hibernate Properties**

```
 <prop key="hibernate.cache.provider_class">
                org.hibernate.cache.EhCacheProvider</prop>
 <prop  key="hibernate.cache.region.factory_class">
                org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory</prop>
 <prop  key="hibernate.cache.use_second_level_cache">true</prop>
 <prop  key="hibernate.cache.use_query_cache">true</prop>
```

ehCache.xml

The defaultCache is applied to any cache not explicitly configured

```
<ehcache>
    <diskStore path="java.io.tmpdir"/>

    <defaultCache
            maxElementsInMemory="1000"
            eternal="false"
            timeToIdleSeconds="120"
            timeToLiveSeconds="120"
            overflowToDisk="true"
            />

     <cache name="edu.mum.domain.Member"
           maxElementsInMemory="1000"
           eternal="false"
           timeToIdleSeconds="600"
           timeToLiveSeconds="3600"
           overflowToDisk="true"
            />

     <cache name="edu.mum.domain.Address"
           maxElementsInMemory="1000"
           eternal="false"
           timeToIdleSeconds="600"
           timeToLiveSeconds="3600"
           overflowToDisk="true"
            />


</ehcache>
```

Spring Cache Abstraction

- Applies caching to **Java methods**

  When method is invoked, the abstraction will apply a caching behavior checks whether the method has been ***already executed for the given arguments***.

  [Does ***NOT*** re-execute the method a second time…]

- **Will Work across “Query” methods**

  Does not distinguish between “Second Level Cache” & Query Cache

  Handles Both

- Spring Cache Abstraction is separate from Hibernate cache
- Hibernate Second Level Cache and Spring Cache (method caching)

  can co-exist with Ehcache as the underlying provider for both.

Cache Abstraction Annotations

- **The abstraction provides following Java annotations:**

- **@Cacheable**:
  Put the method returned value(s) into the cache [**READ ONLY**]
- **@CacheEvict**:
  Remove an entry from the cache [stale or unused…]
- **@CachePut**:
  Force a cache entry to be updated [**READ WRITE related**]

Spring Cache Configuration

- **Declaration – refers to ehCache.xml resource**
- `@Cacheable("edu.mum.domain.Member")`
- `public Member findOne(Long id) {`
- `return memberDao.findOne(id);`
- `}`

```
<cache name="edu.mum.domain.Member"
          maxElementsInMemory="1000"
          eternal="false"
          timeToIdleSeconds="600"
          timeToLiveSeconds="3600"
          overflowToDisk="true"
           />
```

- **XML Configuration**

```
<cache:annotation-driven />
<bean id="cacheManager"
      class="org.springframework.cache.ehcache.EhCacheCacheManager">
    <constructor-arg ref="ehcacheManager" />
</bean>
<bean id="ehcacheManager"
class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
    <property name="configLocation" value="ehcache.xml" />
</bean>
```

Main Point

1. The second level cache can permit more scalability for the application, when the same entities are retrieved by multiple users they could be retained in memory to avoid subsequent accesses to the database, increasing efficiency & speed.
2. ***Science of Consciousness****: Through the constant practice of Transcendental Meditation, the reactions of mind and body are faster.*