Optimization & Performance

Optimization

- **Premature optimization is the root of all evil**
  - Donald Knuth (most important person in Algorithms)
  - See: https://en.wikiquote.org/wiki/Donald_Knuth

- Don't optimize until you have problems
  - Then test to make sure you fixed the right thing

Analyze Problems

- When you do **experience problems**:
  - Check the generated SQL to see what's happening

```
<properties>
   <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/cs544?useSSL=false"/>
   <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
   <property name="javax.persistence.jdbc.user" value="root"/>
   <property name="javax.persistence.jdbc.password" value="root"/>
   <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />

   <!-- Useful for analyzing problems -->
   <property name="hibernate.show_sql" value="true" />
   <property name="hibernate.format_sql" value="true" />

   <!-- 2nd Level Caching -->
   <property name="hibernate.cache.provider_class" value="org.hibernate.cache.NoCacheProvider"/>
   <property name="hibernate.generate_statistics" value="true" />

   <property name="hibernate.id.new_generator_mappings" value="false" />
   <property name="hibernate.hbm2ddl.import_files" value="test.sql" />
   <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
</properties>
```

Optimization

- First we will talk about LAZY and EAGER loading
  - And we will describe **common problems**

- To solve problems we'll look at:
  - Standard JPA ways to solve problems
  - Hibernate extensions to solve problems

CS544 EA

Hibernate

Lazy and Eager

What is Lazy (or Eager)?

- Lazy means it does not load it
  - Until it absolutely needs it
  - By default all **Collections** (*ToMany) **are lazy**

- Eager gets loaded right away
  - As soon as it knows about it
  - By default all **references** (*ToOne) **are eager**

Changing FetchType

- It's **possible to change**
  - References to LAZY
  - collections to EAGER
- Generally not needed

```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;
        @ManyToOne(fetch=FetchType.LAZY)
        private SalesRep salesRep;
        @OneToMany(fetch=FetchType.LAZY)
        @JoinColumn
        private List<Book> books
                = new ArrayList<>();
```

LAZY

LAZY or EAGER

- Some say that all associations should be lazy
  - EAGERly loaded objects may never be used
- Usually Hibernate loads *ToOne with Joins
  - Less expensive then separate selects
  - Still takes overhead, wasted if object not used

- On the whole **you can be safe never modifying**
  - Premature optimization is the root of all evil

@LazyCollection

- A Hibernate extension that:
  - Can be useful **for big collections**

- By default the entire collection is retrieved for:
  - .size(), .isEmpty(), .contains()
  - Instead of **using the DB** to count / check
  - 'Extra lazy' fixes that

@LazyCollection

```
import org.hibernate.annotations.LazyCollection;
import org.hibernate.annotations.LazyCollectionOption;

@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;
        @OneToMany(cascade=CascadeType.ALL)
        @JoinColumn
        @LazyCollection(LazyCollectionOption.EXTRA)
        private List<Movie> movies
                = new ArrayList<>();
```

Extra Lazy

```
Customer c = em.find(Customer.class, 1L);
System.out.println(c.getMovies().size());
```

```
Hibernate:
    select
        customer0_.id as id1_1_0_,
        customer0_.firstName as firstNam2_1_0_,
        customer0_.lastName as lastName3_1_0_,
        customer0_.salesRep_id as salesRep4_1_0_
    from
        Customer customer0_
    where
        customer0_.id=?

Hibernate:
    select
        count(id)
    from
        Movie
    where
        movies_id =?
```

Lazy Properties

- It is possible to make **individual properties lazy**
  - Needs Property access (getters)
  - @Basic(fetch=FetchType.LAZY) (on getters)

- **Needs ByteCode instrumentation** to work
  - Rewrites your getters (after compilation) for ability to load data

- Generally not recommended
  - **DTO projection is better** / easier solution

Hibernate Documentation calls it a "Marketing Feature"

Instead use DTO Projection

- Already discussed during **SELECT new Object**
  - Can select (project) only the properties you need
  - Better than lazy-loading properties!

```
TypedQuery<Home> query = em.createQuery(
        "select new hibernate06.Home(p, a) "
        + "from Person p " + "join p.address a ", Home.class);
List<Home> homes = query.getResultList();

Person p = null;
Address a = null;
for (Home home : homes) {
        p = home.getPerson();
        a = home.getAddress();

        System.out.println(p.getFirstName()
        + " " + p.getLastName()
        + " has a home in " + a.getCity());
}
```

Not an Entity
but a DTO class

```
public class Home {
        private Person person;
        private Address address;

        public Home(Person p, Address a) {
                this.person = p;
                this.address = a;
        }
```

Lazy and Eager Summary

- Lazy and Eager **specify WHEN not HOW**
  - When is generally not the problem

- It's good to know about these options
  - To understand how Hibernate works
  - While not the biggest source of problems or solutions

CS544 EA

Hibernate

Common Problems

Problems

- The essence of ORM related problems are that the database is being **overloaded**
  - Doesn't work properly anymore
  - Cannot handle the load

- Solution avenues are:
  - Lower the load by **using better techniques** (this chapter)
  - Spread the load by caching and scaling (we discuss caching)

Bad Queries

- The most common type of bad query is a **Cartesian Product**
  - **Caused by joining** 2 (or more) **collections**
  - Creates an 'exploded' resultset (takes the DB a long time)

- Hibernate will never generate such a query
  - Throws exception if 2 collections are set as eager on one Entity
  - But a (unaware) programmer can easily write such a query!

Code

Cust1 has 3 books and 3 movies
Cust2 1 book
Cust3 1 movie

```
Customer cust1 = new Customer("Frank", "Brown");
Customer cust2 = new Customer("Jane", "Terrien");
Customer cust3 = new Customer("John", "Doe");
cust1.addBook(new Book("Harry Potter and the Deathly Hallows"));
cust1.addBook(new Book("Unseen Academicals (Discworld)"));
cust1.addBook(new Book("The Color of Magic (Discworld)"));
cust1.addMovie(new Movie("Shrek"));
cust1.addMovie(new Movie("WALL-E"));
cust1.addMovie(new Movie("Howls Moving Castle"));
cust2.addBook(new Book("Twilight (The Twilight Saga, Book1)"));
cust3.addMovie(new Movie("Forgetting Sarah Marshall"));
em.persist(cust1);
em.persist(cust2);
em.persist(cust3);

em.getTransaction().commit();
em.clear();

em.getTransaction().begin();
TypedQuery<Customer> query = em.createQuery(
        "select c from Customer c left join c.movies left join c.books",
        Customer.class);
List<Customer> customers = query.getResultList();

em.getTransaction().commit();
```

Joining 2 collections

```
select
        customer0_.id as id1_3_0_,
        movies1_.id as id1_4_1_,
        books2_.id as id1_2_2_,
        customer0_.address_id as address_4_3_0_,
        customer0_.firstName as firstNam2_3_0_,
        customer0_.lastName as lastName3_3_0_,
        customer0_.salesRep_id as salesRep5_3_0_,
        movies1_.name as name2_4_1_,
        books2_.author_id as author_i3_2_2_,
        books2_.name as name2_2_2_
    from
        Customer customer0_
    left outer join
        Movie movies1_
            on customer0_.id=movies1_.movies_id
    left outer join
        Book books2_
            on customer0_.id=books2_.books_id
```

Resultset

- Joining 2 collections **creates R x N x M rows**
  - R normal rows, N size of cllct. 1, M size of cllct. 2

FIRSTNAME0_0_	LASTNAME0_0_	TITLE1_1_	TITLE2_2_
Frank	Brown	Unseen Academicals (Discworld)	WALL-E
Frank	Brown	Unseen Academicals (Discworld)	Shrek
Frank	Brown	Unseen Academicals (Discworld)	Howls Moving Castle
Frank	Brown	The Color of Magic (Discworld)	WALL-E
Frank	Brown	The Color of Magic (Discworld)	Shrek
Frank	Brown	The Color of Magic (Discworld)	Howls Moving Castle
Frank	Brown	Harry Potter and the Deathly Hallows	WALL-E
Frank	Brown	Harry Potter and the Deathly Hallows	Shrek
Frank	Brown	Harry Potter and the Deathly Hallows	Howls Moving Castle
Jane	Terrien	Twilight (The Twilight Saga, Book1)	[null]
John	Doe	[null]	Forgetting Sarah Marshall

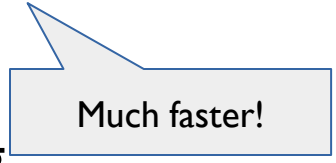
Redundancy

Very Inefficient!

27 cells to give
7 pieces of data

N + 1 Problem

- The N+1 problem is where Hibernate executes **many small selects** to load related data
  - This data could have been loaded in **one big select**



- People sometimes associate it with lazy loading
  - But happens with eager loading too!
  - It's Just Hibernate not knowing **how** to best load data

Lazy Collections N+1

- By default Hibernate lazily loads collections
  - A good default, they can contain a lot of data

- If we create **a query** for all SalesReps
  - Then **use a loop** to get the customers of those reps
  - 1 select for the salesreps (say there are 10)
  - 10 selects, one for each collection of customers

Code

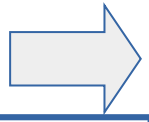
```
em.getTransaction().begin();

SalesRep sr1 = new SalesRep("John Willis");
SalesRep sr2 = new SalesRep("Mary Long");

sr1.addCustomer(new Customer("Frank", "Brown"));
sr1.addCustomer(new Customer("Jane", "Terrien"));
sr2.addCustomer(new Customer("John", "Doe"));
sr2.addCustomer(new Customer("Carol", "Reno"));

em.persist(sr1);
em.persist(sr2);
em.getTransaction().commit();
```

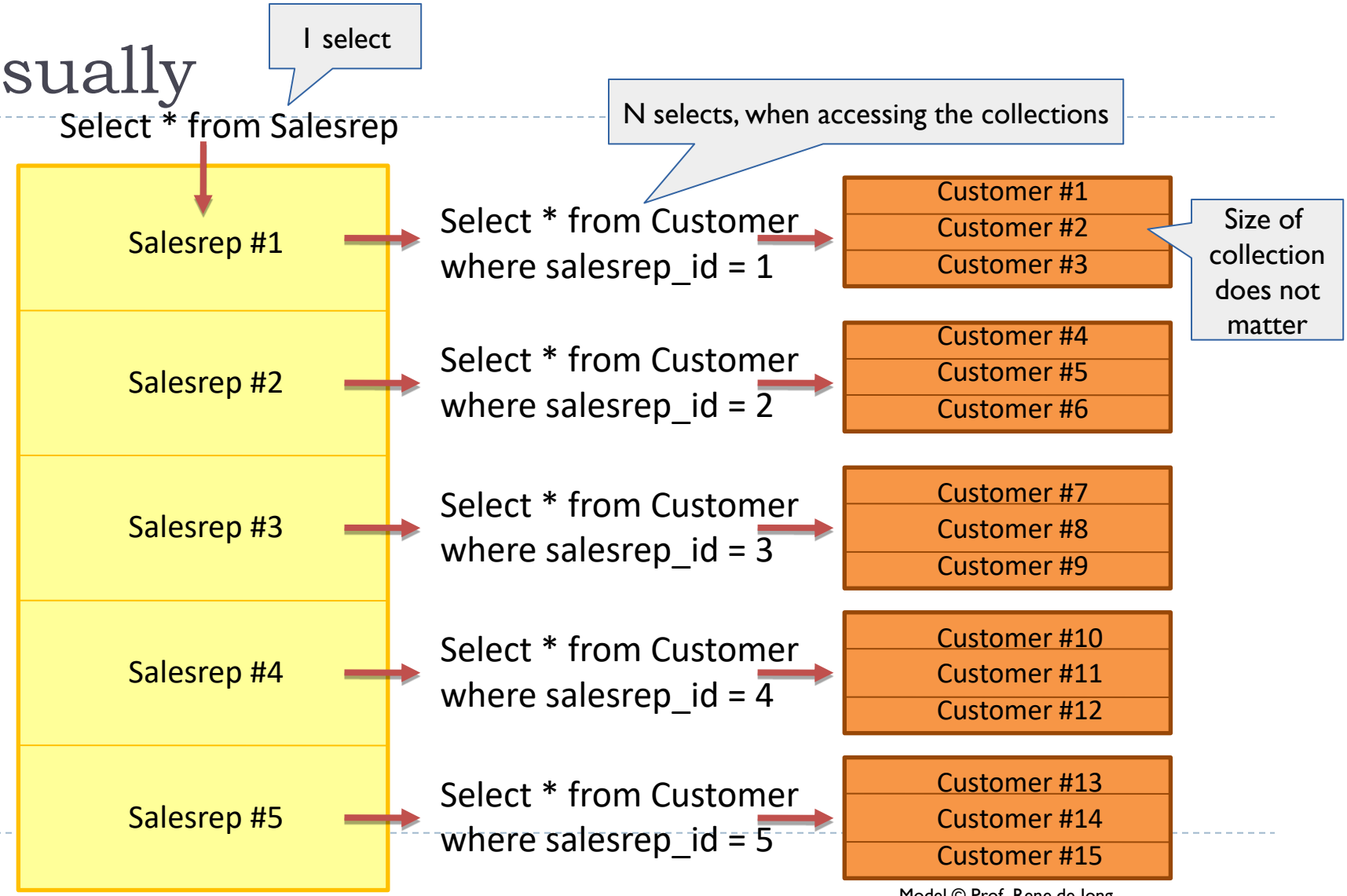
```
TypedQuery<SalesRep> query =
                em.createQuery("from SalesRep", SalesRep.class);
List<SalesRep> salesReps = query.getResultList();
for(SalesRep s : salesReps) {
  System.out.println(s.getCustomers().get(0));
}
```



1 select for the salesreps
N selects for each
list of customers

```
Hibernate:
    select
        salesrep0_.id as id1_1_,
        salesrep0_.name as name2_1_
    from
        SalesRep salesrep0_
Hibernate:
    select
        customers0_.salesRep_id as salesRep4_0_0_,
        customers0_.id as id1_0_0_,
        customers0_.id as id1_0_1_,
        customers0_.firstName as firstNam2_0_1_,
        customers0_.lastName as lastName3_0_1_,
        customers0_.salesRep_id as salesRep4_0_1_
    from
        Customer customers0_
    where
        customers0_.salesRep_id=?
Hibernate:
    select
        customers0_.salesRep_id as salesRep4_0_0_,
        customers0_.id as id1_0_0_,
        customers0_.id as id1_0_1_,
        customers0_.firstName as firstNam2_0_1_,
        customers0_.lastName as lastName3_0_1_,
        customers0_.salesRep_id as salesRep4_0_1_
    from
        Customer customers0_
    where
        customers0_.salesRep_id=?
```

Visually



Eager References N+1

- By default Hibernate uses eager loading for
  - @OneToOne and @ManyToOne
  - If eager associations are not yet fulfilled
    - Hibernate will execute select statements to fix it

Good policy.
Cost of joining a single row is low, and generally reduces selects

- If you execute **1 query** for all customers
  - Without Join Fetch-ing the @ManyToOne SalesRep
  - Hibernate will 'fix' this right away with **N extra selects**

References are Eager by default

Doesn't even need a loop

Code

Each customer has its own Rep

```
em.getTransaction().begin();
Customer cust1 = new Customer("Frank", "Brown");
Customer cust2 = new Customer("Jane", "Terrien");
Customer cust3 = new Customer("John", "Doe");
Customer cust4 = new Customer("Carol", "Reno");
cust1.setSalesRep(new SalesRep("John Willis"));
cust2.setSalesRep(new SalesRep("Mary Long"));
cust3.setSalesRep(new SalesRep("Ted Walker"));
cust4.setSalesRep(new SalesRep("Keith Rogers"));

em.persist(cust1);
em.persist(cust2);
em.persist(cust3);
em.persist(cust4);
em.getTransaction().commit();
```

```
List<Customer> customers = em.createQuery(
        "from Customer").getResultList();
```

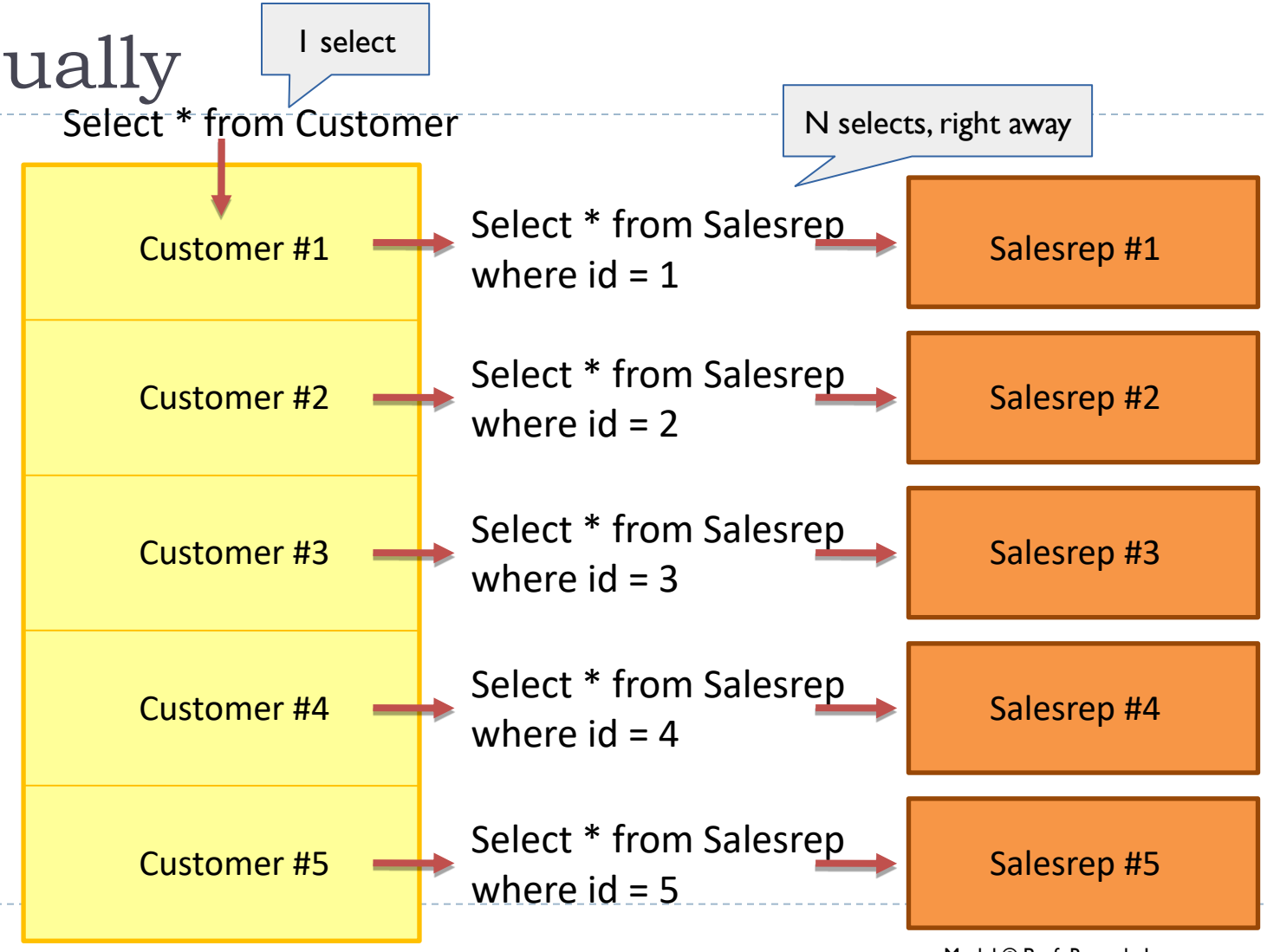
No loop or anything.

Hibernate executes the selects
to fix the missing eager references

1 select for customers,
N selects for the reps

```
Hibernate:
    select
        customer0_.id as id1_0_,
        customer0_.firstName as firstNam2_0_,
        customer0_.lastName as lastName3_0_,
        customer0_.salesRep_id as salesRep4_0_
    from
        Customer customer0_
Hibernate:
    select
        salesrep0_.id as id1_1_0_,
        salesrep0_.name as name2_1_0_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id=?
Hibernate:
    select
        salesrep0_.id as id1_1_0_,
        salesrep0_.name as name2_1_0_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id=?
Hibernate:
    select
        salesrep0_.id as id1_1_0_,
        salesrep0_.name as name2_1_0_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id=?
Hibernate:
```

Visually



Changing Doesn't Help

- Changing the references to LAZY
  - Just makes it so that Hibernate doesn't load the entities until you access them (with a loop)

```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;
        @ManyToOne(fetch=FetchType.LAZY
)
        private SalesRep salesRep;
```

LAZY

- Similarly, changing the collection to EAGER
  - Makes the N selects happen right away
  - The **problem is not in WHEN, but HOW**

Solutions

- The solution for the **Cartesian product** is simple:
  - **Don't join 2 or more** collections in one query
  - Join max 1, use separate queries for the others
  - Similarly other bad queries can be analyzed and fixed

- The solution for **N+1 is not that easy**
  - We'll look at potential strategies coming up

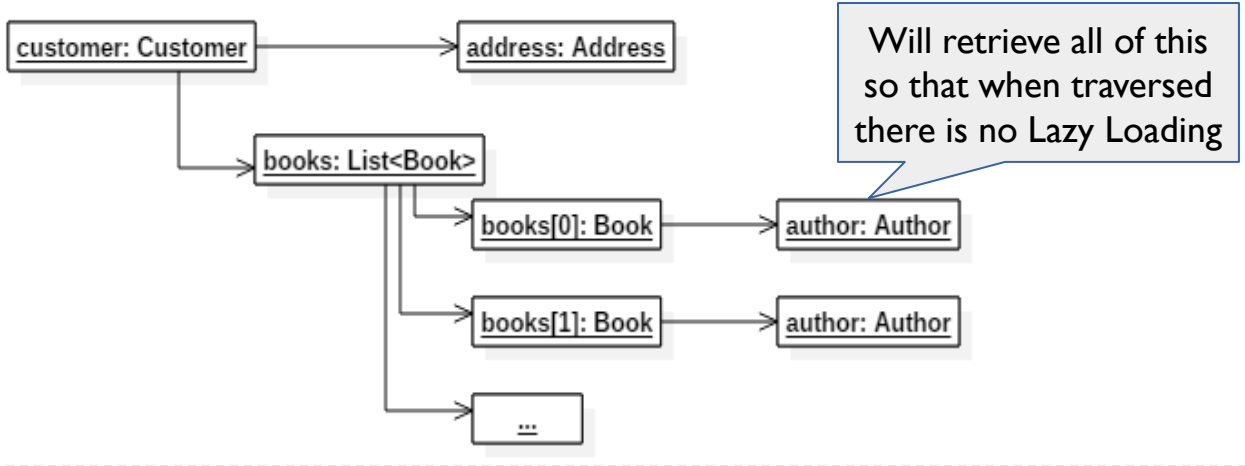
CS544 EA

Hibernate

Optimization: Entity Graph

Entity Graph

- Added in JPA 2.1 (most recent)
  - Specify a **Graph of connected Entities** to retrieve
  - Example: When retrieving a customer we also want to get his address, and all the books he bought, and the author of each of those books



Domain

```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;
        @OneToOne(cascade=CascadeType.ALL)
        private Address address;
        @OneToMany(cascade=CascadeType.ALL)
        @JoinColumn
        private List<Book> books = new ArrayList<>();
```

```
Customer cust1 = new Customer("Frank", "Brown");
Customer cust2 = new Customer("Jane", "Terrien");
Customer cust3 = new Customer("John", "Doe");
cust1.addBook(
        new Book("Harry Potter and the Deathly Hallows",
                new Author("J.K. Rowlings")));
cust1.addBook(
        new Book("Unseen Academicals (Discworld)",
                new Author("Terry Pratchett")));
cust1.addBook(
        new Book("The Color of Magic (Discworld)",
                new Author("Terry Pratchett")));
cust2.addBook(
        new Book("Twilight (The Twilight Saga, Book1)",
                new Author("Stephenie Meyer")));
cust1.setAddress(new Address("Fairfield", "Iowa"));
cust2.setAddress(new Address("Chicago", "Illinois"));
em.persist(cust1);
em.persist(cust2);
em.persist(cust3);
```

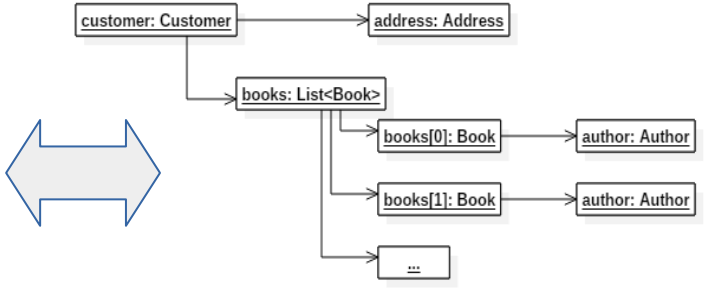
```
@Entity
public class Book {
        @Id
        @GeneratedValue
        private Long id;
        private String name;
        @OneToOne(cascade=CascadeType.ALL)
        private Author author;
```

```
@Entity
public class Address {
        @Id
        @GeneratedValue
        private Long id;
        private String city;
        private String state;
```

EntityGraph

- The purpose of the entity graph is:
  - To **indicate which** references should change to **load eagerly** (in a query or .find())
  - AttributeNodes specify attributes / references
  - SubGraph can be used to go into other Entities

```
EntityGraph<Customer> graph =
        em.createEntityGraph(Customer.class);
graph.addAttributeNodes("address");
graph.addSubgraph("books").addAttributeNodes("author");
```



.createQuery()

```
EntityGraph<Customer> graph =
em.createEntityGraph(Customer.class);
graph.addAttributeNodes("address");
graph.addSubgraph("books").addAttributeNodes("author");

TypedQuery<Customer> query = em.createQuery(
        "from Customer where firstName like :name",
        Customer.class);
query.setParameter("name", "J%");
query.setHint("javax.persistence.fetchgraph", graph);

List<Customer> customers = query.getResultList();
System.out.println(customers.size());
```

The EntityGraph is passed as a query Hint

Hibernate loads the entire graph into cache While returning the Customer as query result

```
Hibernate:
    select
        customer0_.id as id1_3_0_,
        customer0_.address_id as address_4_3_0_,
        customer0_.firstName as firstNam2_3_0_,
        customer0_.lastName as lastName3_3_0_,
        address1_.id as id1_0_1_,
        address1_.city as city2_0_1_,
        address1_.state as state3_0_1_,
        books2_.books_id as books_id4_2_2_,
        books2_.id as id1_2_2_,
        books2_.id as id1_2_3_,
        books2_.author_id as author_i3_2_3_,
        books2_.name as name2_2_3_,
        author3_.id as id1_1_4_,
        author3_.name as name2_1_4_
    from
        Customer customer0_
    left outer join
        Address address1_
            on customer0_.address_id=address1_.id
    left outer join
        Book books2_
            on customer0_.id=books2_.books_id
    left outer join
        Author author3_
            on books2_.author_id=author3_.id
    where
        customer0_.id=?
2
```

.find()

```
EntityGraph<Customer> graph
        = em.createEntityGraph(Customer.class);
graph.addAttributeNodes("address");
graph.addSubgraph("books").addAttributeNodes("author");

Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.fetchgraph", graph);

em.find(Customer.class, 1L, properties);
```

Hints are passed as properties Map to .find()

Hibernate loads the entire graph into cache giving us the Root (Customer) entity

```
Hibernate:
    select
        customer0_.id as id1_3_0_,
        customer0_.address_id as address_4_3_0_,
        customer0_.firstName as firstNam2_3_0_,
        customer0_.lastName as lastName3_3_0_,
        address1_.id as id1_0_1_,
        address1_.city as city2_0_1_,
        address1_.state as state3_0_1_,
        books2_.books_id as books_id4_2_2_,
        books2_.id as id1_2_2_,
        books2_.id as id1_2_3_,
        books2_.author_id as author_i3_2_3_,
        books2_.name as name2_2_3_,
        author3_.id as id1_1_4_,
        author3_.name as name2_1_4_
    from
        Customer customer0_
    left outer join
        Address address1_
            on customer0_.address_id=address1_.id
    left outer join
        Book books2_
            on customer0_.id=books2_.books_id
    left outer join
        Author author3_
            on books2_.author_id=author3_.id
    where
        customer0_.id=?
```


Entity Graph and N+1

- An entity graph can be a solution to N+1
  - Load all the needed entities in one query

- Potential problems:
  - You can not make more than one collection eager
  - Eager associations from your graph / result to other entities(outside your graph) **still cause N+1** (see eager references N+1)

If you have references going out make sure they're lazy!

CS544 EA

Hibernate

Optimization: Join Fetch

Join Fetch

- Before Fetch Graphs were added to JPA
  - Queries could already do "Join Fetch"

- Like EntityGraph Join Fetch-ed entities are:
  - **Added to the cache**
  - **Not added to the result set**
- Unlike EntityGraph
  - Join Fetch now uses inner join (instead of left outer)

No join means row is removed

Join Fetch

No SELECT clause needed
joined entities not added to ResultSet

```
TypedQuery<Customer> query = em.createQuery(
                "from Customer c "
                + "JOIN FETCH c.address a "
                + "JOIN FETCH c.books b "
                + "JOIN FETCH b.author "
                + "WHERE c.firstName like :name", Customer.class);
query.setParameter("name", "J%");
List<Customer> customers = query.getResultList();
System.out.println(customers.size());
```

Remember: don't join (fetch)
multiple collections!

```
Hibernate:
    select
        customer0_.id as id1_3_0_,
        address1_.id as id1_0_1_,
        books2_.id as id1_2_2_,
        author3_.id as id1_1_3_,
        customer0_.address_id as address_4_3_0_,
        customer0_.firstName as firstNam2_3_0_,
        customer0_.lastName as lastName3_3_0_,
        address1_.city as city2_0_1_,
        address1_.state as state3_0_1_,
        books2_.author_id as author_i3_2_2_,
        books2_.name as name2_2_2_,
        books2_.books_id as books_id4_2_0__,
        books2_.id as id1_2_0__,
        author3_.name as name2_1_3_
    from
        Customer customer0_
    inner join
        Address address1_
            on customer0_.address_id=address1_.id
    inner join
        Book books2_
            on customer0_.id=books2_.books_id
    inner join
        Author author3_
            on books2_.author_id=author3_.id
    where
        customer0_.firstName like ?
1
```

Join Fetch and N+1

- Join Fetch can be a solution for N+1
  - Load all the needed objects in one query

- Same potential problems:
  - You can not Join Fetch more than one collection eager
  - Eager associations from your graph / result to other entities **still cause N+1** (see eager references N+1)

If you have references going out make sure they're lazy!

CS544 EA

Hibernate

Optimization: BatchSize

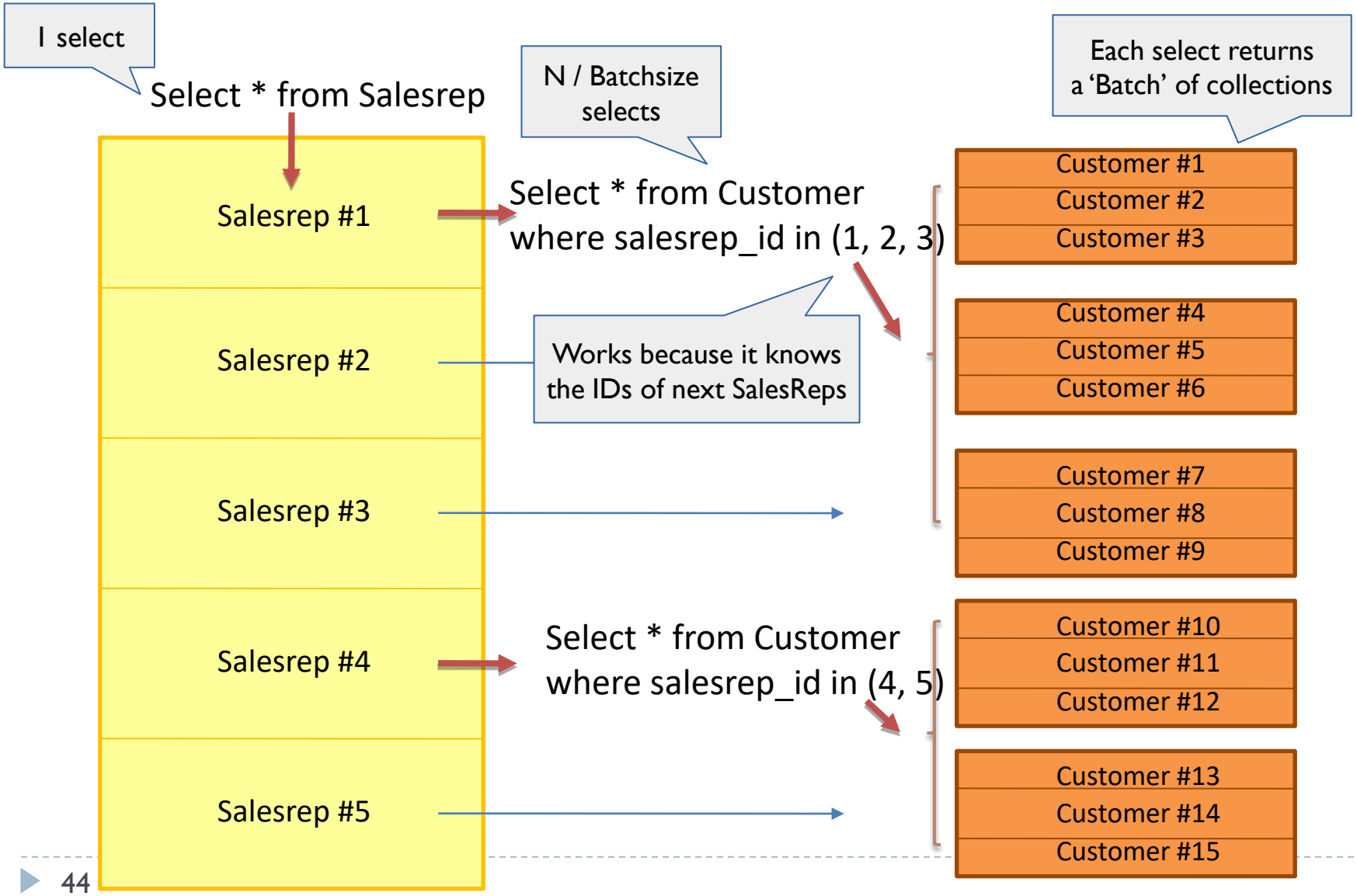
@BatchSize

- Hibernate extension commonly used for N+1
  - Helps by loading **several collections in one 'batch'**
  - Gets another batch when previous batch is empty
  - Turns N + 1 into: ceil(N / Batchsize) + 1

```
import org.hibernate.annotations.BatchSize;

@Entity
public class SalesRep {
        @Id
        @GeneratedValue
        private Long id;
        private String name;
        @OneToMany(mappedBy = "salesRep", cascade=CascadeType.ALL)
        @BatchSize(size=3)
        private List<Customer> customers = new ArrayList<>();
```

Typical size between 3 and 15



Code

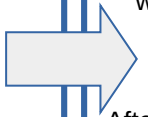
```
em.getTransaction().begin();

SalesRep sr1 = new SalesRep("John Willis");
SalesRep sr2 = new SalesRep("Mary Long");

sr1.addCustomer(new Customer("Frank", "Brown"));
sr1.addCustomer(new Customer("Jane", "Terrien"));
sr2.addCustomer(new Customer("John", "Doe"));
sr2.addCustomer(new Customer("Carol", "Reno"));

em.persist(sr1);
em.persist(sr2);
em.getTransaction().commit();
```

```
System.out.println("Before query");
TypedQuery<SalesRep> query =
                em.createQuery("from SalesRep", SalesRep.class);
List<SalesRep> salesReps = query.getResultList();
System.out.println("After query");
for (SalesRep s : salesReps) {
        s.getCustomers().get(0).getFirstName();
}
System.out.println("After loop");
```



```
Before query
Hibernate:
    select
        salesrep0_.id as id1_3_,
        salesrep0_.name as name2_3_
    from
        SalesRep salesrep0_
After query
Hibernate:
    select
        customers0_.salesRep_id as salesRep4_1_1_,
        customers0_.id as id1_1_1_,
        customers0_.id as id1_1_0_,
        customers0_.firstName as firstNam2_1_0_,
        customers0_.lastName as lastName3_1_0_,
        customers0_.salesRep_id as salesRep4_1_0_
    from
        Customer customers0_
    where
        customers0_.salesRep_id in (
            ?, ?
        )
After loop
```

Batch loaded when customer first needed

BatchSize and N+1

- @BatchSize does not completely eliminate N+1
  - It does significantly reduce it
  - N+ 1 becomes: **[N / batchsize] + 1**

- Potential Problems:
  - N+1 effects not completely removed (just reduced)
  - Static, always on, no way to not use it
  - May load (batchsize - 1) objects not needed

CS544 EA

Hibernate

Optimization: FetchMode.SUBSELECT

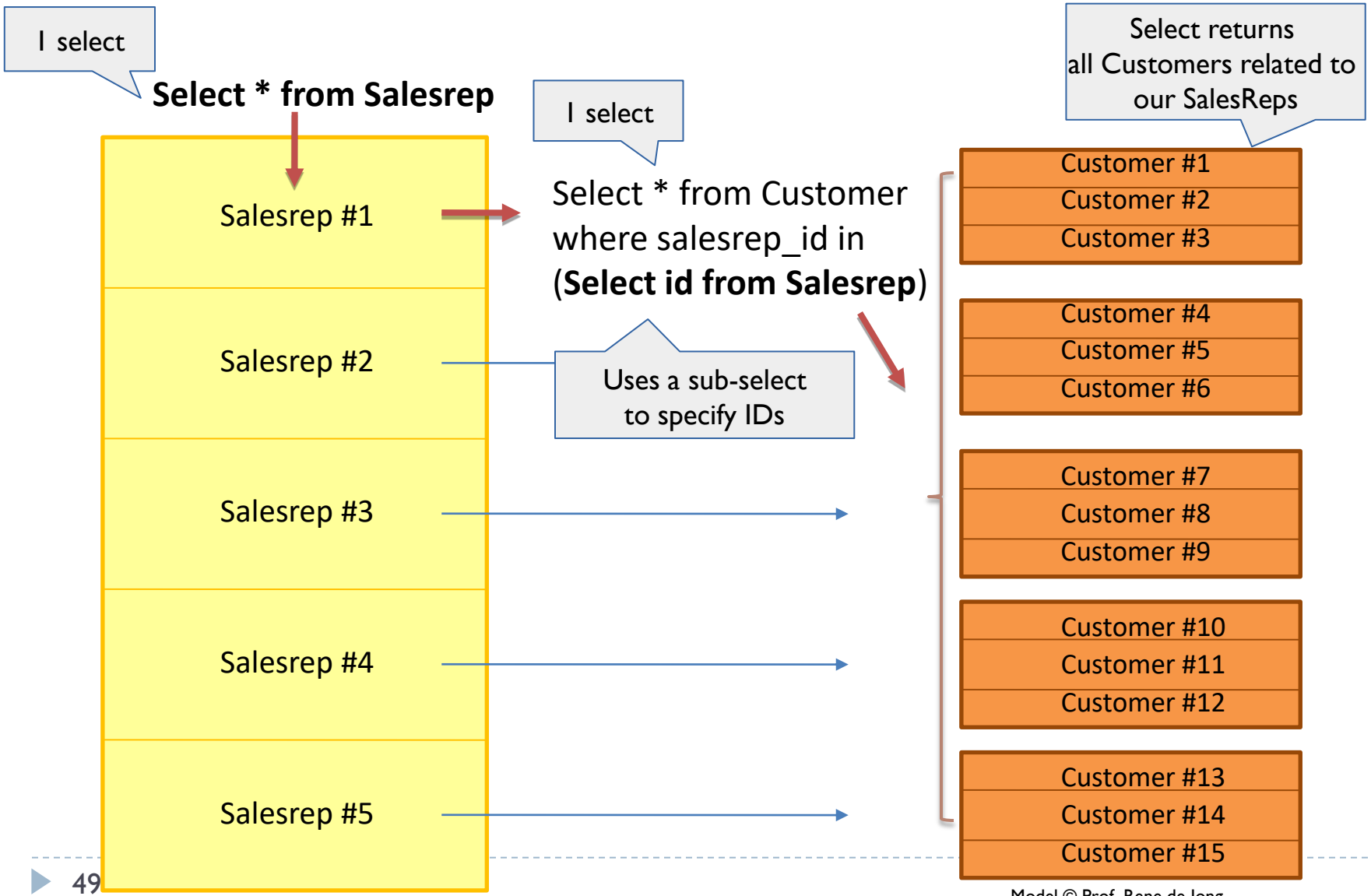
FetchMode.SUBSELECT

- Hibernate Extension commonly used for N+1
  - **Turns N + 1 into 1 + 1**
  - Can load **too much data**

@Fetch

```
import org.hibernate.annotations.Fetch;
import org.hibernate.annotations.FetchMode;

@Entity
public class SalesRep {
        @Id
        @GeneratedValue
        private Long id;
        private String name;
        @OneToMany(mappedBy = "salesRep", cascade=CascadeType.ALL)
        @Fetch(FetchMode.SUBSELECT)
        private List<Customer> customers = new ArrayList<>();
```



Code

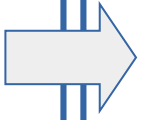
```
em.getTransaction().begin();

SalesRep sr1 = new SalesRep("John Willis");
SalesRep sr2 = new SalesRep("Mary Long");

sr1.addCustomer(new Customer("Frank", "Brown"));
sr1.addCustomer(new Customer("Jane", "Terrien"));
sr2.addCustomer(new Customer("John", "Doe"));
sr2.addCustomer(new Customer("Carol", "Reno"));

em.persist(sr1);
em.persist(sr2);
em.getTransaction().commit();
```

```
System.out.println("Before query");
TypedQuery<SalesRep> query =
                em.createQuery("from SalesRep where id < 1000",
                        SalesRep.class);
List<SalesRep> salesReps = query.getResultList();
System.out.println("After query");
for (SalesRep s : salesReps) {
        s.getCustomers().get(0).getFirstName();
}
System.out.println("After loop");
```



```
Before query
Hibernate:
    select
        salesrep0_.id as id1_3_,
        salesrep0_.name as name2_3_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id<1000
After query
Hibernate:
    select
        customers0_.salesRep_id as salesRep4_1_1_,
        customers0_.id as id1_1_1_,
        customers0_.id as id1_1_0_,
        customers0_.firstName as firstNam2_1_0_,
        customers0_.lastName as lastName3_1_0_,
        customers0_.salesRep_id as salesRep4_1_0_
    from
        Customer customers0_
    where
        customers0_.salesRep_id in (
            select
                salesrep0_.id
            from
                SalesRep salesrep0_
            where
                salesrep0_.id<1000
        )
After loop
```

SubSelect Query executes when first customer needed

Includes where

SubSelect and N+1

- SUBSELECT is like a **supercharged BatchSize**
  - Works on same principle but goes all the way
  - Turns N + 1 into 1 + 1

- Potential problem:
  - Static, always on, no way to not use it
  - May load everything even if we only needed 1

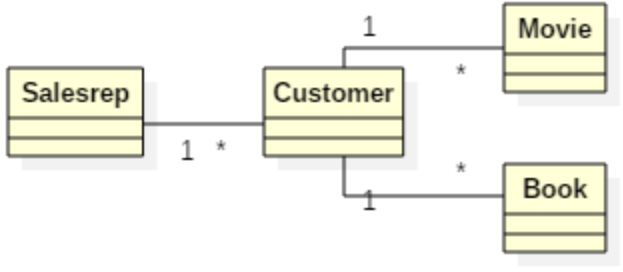
N+1 Summary

- Entity Graph & Join Fetch (very similar)
  - Dynamic, you can chose when to use them
  - Never join more than 1 collection
  - Be careful with eager associations outside result
- BatchSize
  - Static → may load a bit too much
- SubSelect
  - Static → may load way too much

While I personally like BatchSize and Join Fetch you really need to analyze what works best for your situation

Cartesian Product Summary

- To fix a Cartesian product, use more queries.
- For example if we need to make a report on which active Salesrep sold the most books or movies



- Create the following queries:
  - From Salesrep s join s.customers c where s.active = 1
  - From Customer c join c.books b where c.salesrep.active=1
  - From Customer c join c.movies m where c.salesrep.active=1
- Once data is loaded you can use it (without N+1 fear)

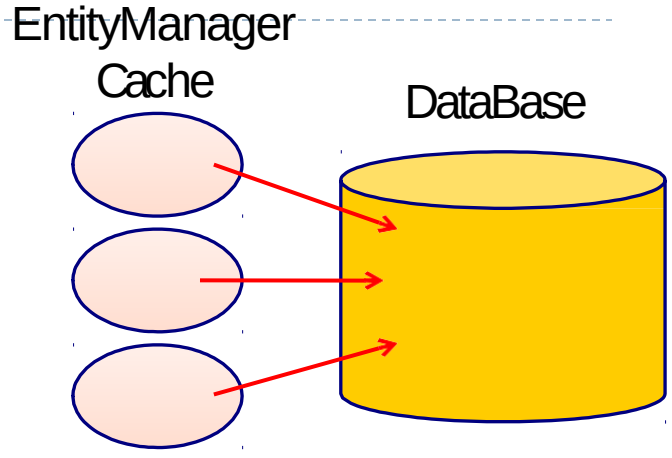
CS544 EA

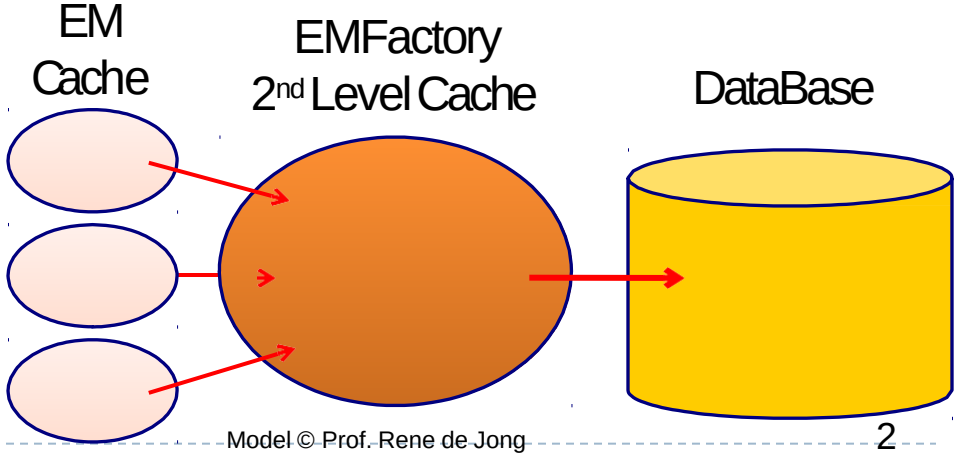
Hibernate

Optimization: 2nd Level Cache

2nd Level Caching

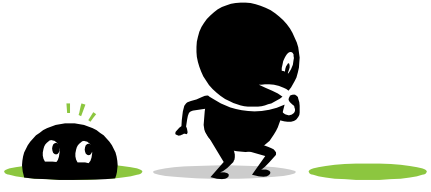
- By default JPA only uses EntityManager cache
  - Very short term cache
- To reduce hits on the DB
  - Objects can also be cached for longer
  - Managed by EntityManagerFactory
  - Shared by all EntityManagers





Caching VS Optimization

- Caching can be seen as a **form of scaling**
  - Doesn't solve bad queries
  - But can alleviate pressure on the DB
- Caching is a large and interesting field
  - We will look at some basics
  - Be aware that **improper configuration** can create situations that are **hard to debug** (cached versions != DB versions)



What to cache?

- Good candidates for caching:
  - **Do not change**, or change rarely
  - Are modified only by your app
  - Are non-critical to the app

- Typically: **Reference data**

4 Caching Strategies

Stricter and therefore Slower

- **Read Only**: very fast strategy, but can only be used for data that never changes
- **Non-Strict Read-Write**: data may be stale for a while, but gets refreshed at a timeout
- **Read-Write**: prevents stale data, but at a cost. Use for read-mostly data in a non-clustered setup
- **Transactional**: Can prevent stale data in a clustered environment. Can be used for read-mostly data

Cache Providers

- Hibernate can have **only one** provider per EMF

Provider	Read Only	Non Strict Read Write	Read Write	Transactional
EHCache	✓	✓	✓	
OSCache	✓	✓	✓	
SwarmCache	✓	✓		
JBoss Cache 1.x	✓			✓
JBoss Cache 2.x	✓			✓

Annotate Classes with Strategy

- Using Hibernate's **@Cache** annotation

```
@Entity
@Cache(usage= CacheConcurrencyStrategy.READ_ONLY)
public class Customer implements Serializable {
    @Id
    @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;

    public Customer() {
    }

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

}
```

Setup Cache Provider

- Inside **persistence.xml**

```
<properties>
   <!-- Useful for analyzing problems -->
   <property name="hibernate.show_sql" value="true"/>
   <property name="hibernate.format_sql" value="false"/>

   <!-- 2nd Level Caching -->
   <!--By default is true, second-level-cache is enabled-->
   <property name="hibernate.cache.use_second_level_cache" value="true"/>
   <!--enable query caching-->
   <!--<property name="hibernate.cache.use_query_cache" value="true" />-->
   <property name="hibernate.cache.region.factory_class"
         value="org.hibernate.cache.jcache.JCacheRegionFactory"/>
   <property name="net.sf.ehcache.configurationResourceName" value="/ehcache.xml"/>
   <property name="hibernate.javax.cache.provider" value="org.ehcache.jsr107.EhcacheCachingProvider"/>
   <!-- To analyze cache performance -->
   <property name="hibernate.generate_statistics" value="true"/>

    …

</properties>
```

Configure Cache Provider

```
<ehcache>
  <diskStore path="java.io.tmpdir"/>
  <defaultCache
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true" />

  <cache name="cacheDemo.Category"
    maxElementsInMemory="50"
    eternal="true"
    timeToIdleSeconds="0"
    timeToLiveSeconds="0"
    overflowToDisk="false" />

  <cache
name="cacheDemo.Category.customers"
    maxElementsInMemory="50"
    eternal="false"
    timeToIdleSeconds="3600"
    timeToLiveSeconds="7200"
    overflowToDisk="false" />

  <cache name="cacheDemo.SalesRep"
    maxElementsInMemory="500"
    eternal="false"
    timeToIdleSeconds="1800"
    timeToLiveSeconds="10800"
    overflowToDisk="false" />
</ehcache>
```

General Config

Config for an Entity

Config for a Collection

Statistics

```
SessionFactory sessionFactory = emf.unwrap(SessionFactory.class);

Statistics stats = sessionFactory.getStatistics();
long hits   = stats.getSecondLevelCacheHitCount();
long misses = stats.getSecondLevelCacheMissCount();
long puts   = stats.getSecondLevelCachePutCount();
System.out.printf("\nGeneral 2nd Level Cache Stats\n");
System.out.printf("Hit: %d Miss: %d Put: %d\n", hits, misses, puts);

org.hibernate.stat.CacheRegionStatistics customerCacheStats =
      stats.getCacheRegionStatistics("hibernate07.Customer");
long srCurrent = customerCacheStats.getElementCountInMemory();
long srMemsize = customerCacheStats.getSizeInMemory();
long srHits    = customerCacheStats.getHitCount();
long srMisses  = customerCacheStats.getMissCount();
long srPuts    = customerCacheStats.getPutCount();
System.out.printf("\ncustomerCache Cache Region - Size: %d Holds: %d\n", srMemsize, srCurrent);
System.out.printf("Hit: %d Miss: %d Put: %d\n", srHits, srMisses, srPuts);
```

General 2nd level cache statistics

Statistics for a specific cache region

```
SessionFactory sessionFactory = emf.unwrap(SessionFactory.class);
Statistics stats = sessionFactory.getStatistics();
Stats.clear();
stats.setStatisticsEnabled(true);
...
stats.setStatisticsEnabled(false);
```

You can also programmatically turn stats on and off for more targeted measuring

Summary

- Premature optimization is the root of all evil
- If your DB is under load use better techniques:
  - Check the SQL that's generated
    - Make sure no Cartesian Product
  - If you find N+1 problems use:
    - Dynamic: EntityGraph, or JoinFetch
    - Static: @BatchSize, or SubSelect
- Use Caching (or scaling) to further reduce load