Hibernate Optimization

CS544: Enterprise Architecture

Wholeness

- There are times when your application slows down, things don’t work as well as you expected them to. Optimization is the process of fixing these types of problems.
- *Science of Consciousness*: The source of all thought is also the source of all solutions. Aligning our mind with its source helps us think more clearly about solutions.

Hibernate Optimization:

PERFORMANCE PROBLEMS

Slow? → What to Look For

- There are two main problem categories:
  - Many selects to get similar, or closely related data
    - These selects can probably be combined
    - The N + 1 problem is an example of this
    - Caused by inappropriate lazy loading of data
  - Complex queries that use many joins
    - May be more efficient to use several simple queries
    - Cartesian Product problem is an example of this
    - Caused by incorrect (over) optimization

N + 1 with Collections

2 Sales Reps, each with a collection of customers

```
session = sessionFactory.openSession();
tx = session.beginTransaction();

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"));

session.persist(sr1);
session.persist(sr2);

tx.commit();
```

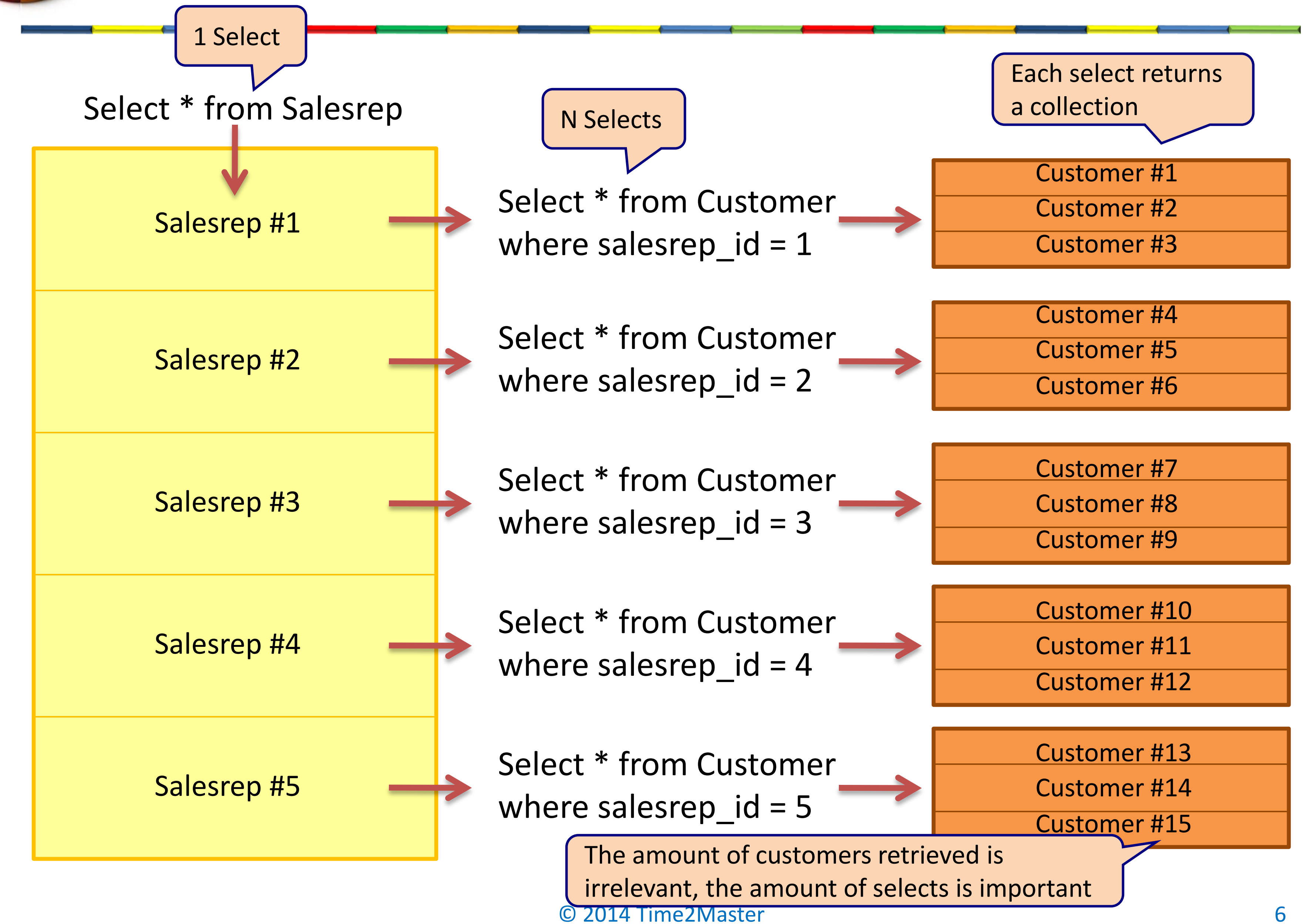
```
List<SalesRep> salesReps =
  session.createQuery("from SalesRep").list();
for (SalesRep s : salesReps) {
  Set<Customer> customers = s.getCustomers();
  for (Customer c : customers) {
    // do something with the customer
  }
}
```

Retrieve sales reps, and then work with related customers

Gets the sales reps (1 query), and then executes another query for each sales rep (N queries). Total N + 1 queries

```
Hibernate:
    select
        salesrep0_.id as id1_,
        salesrep0_.name as name1_
    from
        SalesRep salesrep0_
Hibernate:
    select
        customers0_.salesRep_id as salesRep4_1_,
        customers0_.id as id1_,
        customers0_.id as id0_0_,
        customers0_.firstname as firstname0_0_,
        customers0_.lastname as lastname0_0_,
        customers0_.salesRep_id as salesRep4_0_0_
    from
        Customer customers0_
    where
        customers0_.salesRep_id=?
Hibernate:
    select
        customers0_.salesRep_id as salesRep4_1_,
        customers0_.id as id1_,
        customers0_.id as id0_0_,
        customers0_.firstname as firstname0_0_,
        customers0_.lastname as lastname0_0_,
        customers0_.salesRep_id as salesRep4_0_0_
    from
        Customer customers0_
    where
        customers0_.salesRep_id=?
```

N + 1 with Collections (to-many)



1 Select

Select * from Salesrep

N Selects

Each select returns a collection

Salesrep #1

Select * from Customer where salesrep_id = 1

Customer #1
Customer #2
Customer #3

Salesrep #2

Select * from Customer where salesrep_id = 2

Customer #4
Customer #5
Customer #6

Salesrep #3

Select * from Customer where salesrep_id = 3

Customer #7
Customer #8
Customer #9

Salesrep #4

Select * from Customer where salesrep_id = 4

Customer #10
Customer #11
Customer #12

Salesrep #5

Select * from Customer where salesrep_id = 5

Customer #13
Customer #14
Customer #15

The amount of customers retrieved is irrelevant, the amount of selects is important

N + 1 With Lazy References

4 Customers each with their own (FetchType.Lazy) salesrep

```
session = sessionFactory.openSession();
tx = session.beginTransaction();

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"));

session.persist(cust1);
session.persist(cust2);
session.persist(cust3);
session.persist(cust4);

tx.commit();
```

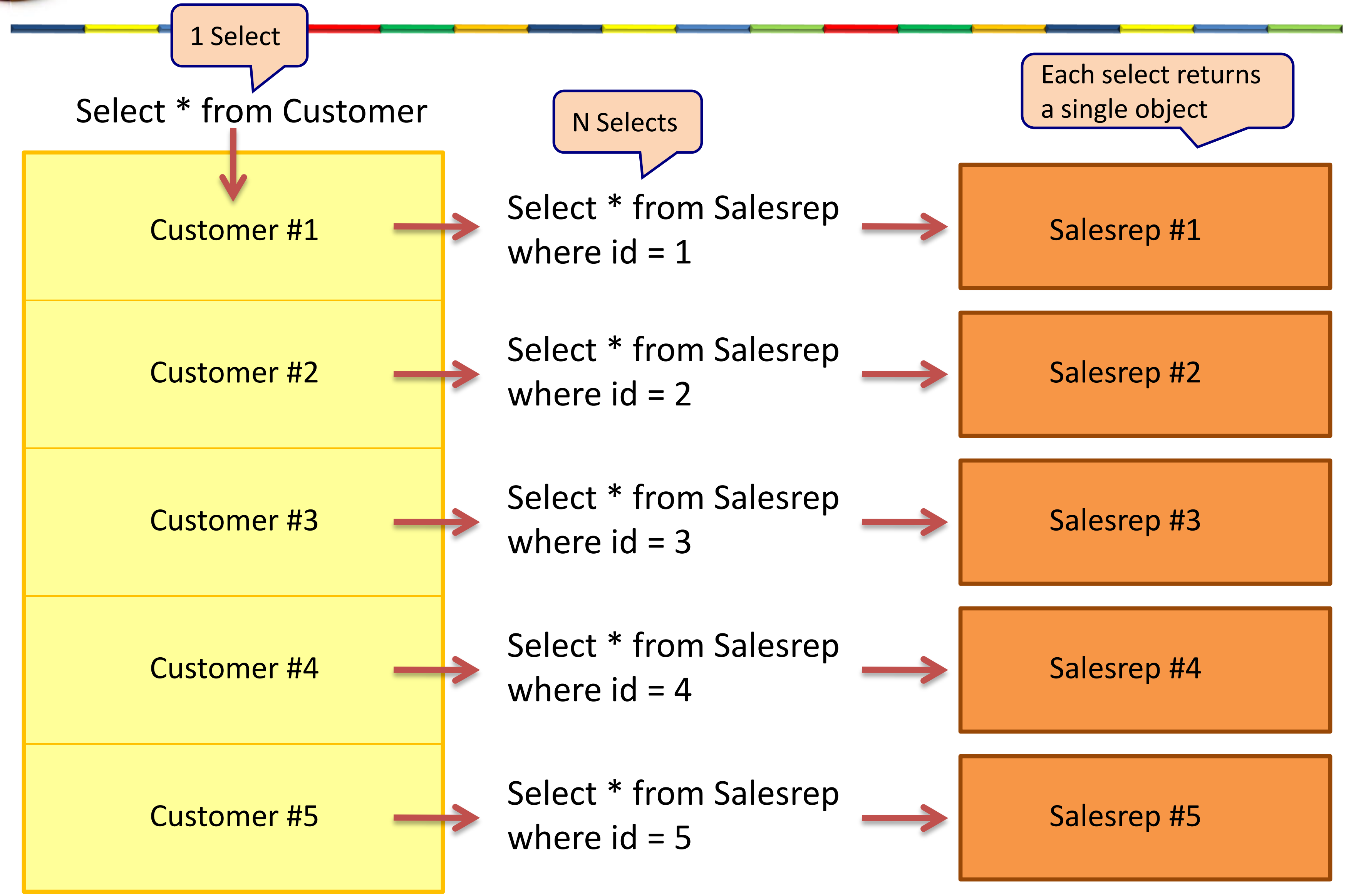
```
List<Customer> customers =
    session.createQuery("from Customer").list();
SalesRep salesrep = null;
for (Customer customer : customers) {
  salesrep = customer.getSalesRep();
  // do something with the salesrep
  salesrep.getName();
}
```

Retrieve customers and then work with related salesrep

Gets the customers (1 query), then executes another query for each salesrep (N queries). Total N + 1

```
Hibernate
    select
        customer0_.id as id0_,
        customer0_.firstname as firstname0_,
        customer0_.lastname as lastname0_,
        customer0_.salesRep_id as salesRep4_0_
    from
        Customer customer0_
Hibernate:
    select
        salesrep0_.id as id1_0_,
        salesrep0_.name as name1_0_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id=?
Hibernate:
    select
        salesrep0_.id as id1_0_,
        salesrep0_.name as name1_0_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id=?
ernate:
    select
        salesrep0_.id as id1_0_,
        salesrep0_.name as name1_0_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id=?
```

N + 1 with Lazy References (ToOne)



1 Select

Select * from Customer

N Selects

Each select returns a single object

Customer #1 → Select * from Salesrep where id = 1 → Salesrep #1

Customer #2 → Select * from Salesrep where id = 2 → Salesrep #2

Customer #3 → Select * from Salesrep where id = 3 → Salesrep #3

Customer #4 → Select * from Salesrep where id = 4 → Salesrep #4

Customer #5 → Select * from Salesrep where id = 5 → Salesrep #5

Cartesian Product Problem

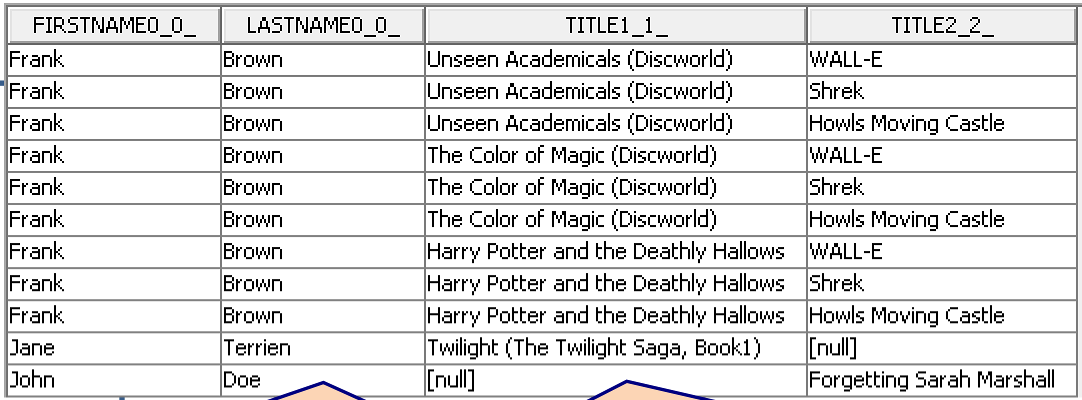
```
Customer cust1 = new Customer("Frank", "Brown");
Customer cust2 = new Customer("Jane", "Terrien");
Customer cust3 = new Customer("John", "Doe");

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"));
```

Customers have a set of books, and a set of movies that they like.

First customer has 3 books and 3 movies, second customer has a single book, third customer has a single movie

Retrieve customers, and also try to (eager) fetch the book and movie collections for the customers

```
Hibernate:
 select
        customer0_.firstname as firstname0_0_,
        customer0_.lastname as lastname0_0_,
        books1_.title as title1_1_,
        movies2_.title as title2_2_
    from
        Customer customer0_
    left outer join
        Book books1_
            on customer0_.id=books1_.customer_id
    left outer join
        Movie movies2_
            on customer0_.id=movies2_.customer_id
```



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

Outer Joining two or more collections creates many redundant rows

Row count per customers = related books * related movies

Cartesian Product

- Joining two collections creates: R x N x M
  - Creating a very in-efficient resultset

Frank Brown ✓	Discworld ✓	Pixar ✓
Frank Brown	Discworld	Dream Works ✓
Frank Brown	Discworld	Studio Ghibli ✓
Frank Brown	Harry Potter ✓	Pixar
Frank Brown	Harry Potter	Dream Works
Frank Brown	Harry Potter	Studio Ghibli
Frank Brown	Twilight ✓	Pixar
Frank Brown	Twilight	Dream Works
Frank Brown	Twilight	Studio Ghibli

9 rows, 3 columns to give 7 pieces of data

Main Point

- The most common Hibernate performance problems are the N+1 problem, and the Cartesian product problem. Both of these are caused by a misunderstanding about what happens at deeper levels.
- *Science of Consciousness*: Life is found in Layers. In Cosmic Consciousness our mind is permanently established in the transcendent and is no longer troubled by problems (like a millionaire at the market).

Hibernate Optimization

JOIN FETCH QUERY

Join Fetch Query

- A Join Fetch Query is the most flexible strategy
  - Other strategies are defined in mapping data

    →Mapping data is always used by all use cases
  - Join Fetch Queries are defined in code

    →Only executed in the use case that it is defined in

- Like Eager Joining, join fetch queries use SQL joins to pre-cache additional data
  - Extra data is not returned as part of the result set

Join Fetch Queries

- Queries can safely join multiple referenced objects
- Should not join more than one collection
  - Even for a single collection ‘distinct’ is needed
  - Multiple collections create a Cartesian product

```
Query query = session.createQuery("select distinct p "
    + "from Person p left join fetch p.accounts");
List<Person> people = query.list();
```

Fetch joins are outer joins even if you do not specify LEFT or OUTER

```
Criteria criteria = session.createCriteria(Person.class)
   .setFetchMode("accounts", FetchMode.JOIN)
   .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY);
List<Person> people = criteria.list();
```

```
Hibernate:
select
        distinct person0_.id as id0_0_,
        accounts1_.number as number1_1_,
        person0_.firstname as firstname0_0_,
        person0_.lastname as lastname0_0_,
        accounts1_.balance as balance1_1_,
        accounts1_.owner_id as owner3_1_1_,
        accounts1_.owner_id as owner3_0__,
        accounts1_.number as number0__
    from
        Person person0_
    left outer join
        Account accounts1_
            on person0_.id=accounts1_.owner_id
```

Loads person objects and pre-cache the associated accounts using a single select

Hibernate Optimization

FETCHTYPE: LAZY AND EAGER

Lazy Loading

- Lazy loading can be specified for:
  - Object References
    - one-to-one and many-to-one associations
    - By default don’t use lazy loading (not a bad thing)
  - Collections
    - one-to-many and many-to-many associations
    - Have the option to use ‘extra-lazy’ loading
  - Large Properties
    - CLOBs and BLOBs, e.g. large texts or image data
    - Need byte code instrumentation to use lazy loading

Object References

- The JPA specifies that both @ManyToOne and @OneToOne default to mostly eager-like loading

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  @OneToOne(cascade=CascadeType.PERSIST)
  private Address address;

  @ManyToOne
  private SalesRep salesRep;
```

```
Customer cust1 = (Customer)
    session.get(Customer.class, 1);
```

```
Hibernate:
    select
        customer0_.id as id0_2_,
        customer0_.address_id as address4_0_2_,
        customer0_.firstname as firstname0_2_,
        customer0_.lastname as lastname0_2_,
        customer0_.salesRep_id as salesRep5_0_2_,
        address1_.id as id1_0_,
        address1_.apt as apt1_0_,
        address1_.city as city1_0_,
        address1_.state as state1_0_,
        address1_.street as street1_0_,
        address1_.zip as zip1_0_,
        salesrep2_.id as id3_1_,
        salesrep2_.name as name3_1_
    from
        Customer customer0_
    left outer join
        Address address1_
            on customer0_.address_id=address1_.id
    left outer join
        SalesRep salesrep2_
            on customer0_.salesRep_id=salesrep2_.id
    where
        customer0_.address_id=?
```

Hibernate retrieves the customer, the address, *and* the salesrep

Specifying Lazy

- Using fetch = FetchType.LAZY @OneToOne and @ManyToOne can become lazy

```
@Entity
public class Customer {
@Id
@GeneratedValue
private int id;
private String firstname;
private String lastname;

@OneToOne(fetch = FetchType.LAZY,
    cascade=CascadeType.PERSIST)
private Address address;

@ManyToOne(fetch = FetchType.LAZY)
private SalesRep salesRep;

...
```

FetchType.LAZY

FetchType.LAZY

```
Customer cust1 = (Customer)
    session.get(Customer.class, 1);
```

Hibernate again only retrieves the customer object

```
Hibernate:
    select
        customer0_.id as id0_0_,
        customer0_.address_id as address4_0_0_,
        customer0_.firstname as firstname0_0_,
        customer0_.lastname as lastname0_0_,
        customer0_.salesRep_id as salesRep5_0_0_
    from
        Customer customer0_
    where
        customer0_.id=?
```

XML

- <one-to-one> defaults to eager-like loading
- <many-to-one> defaults to lazy loading

```
public class Customer {
  private int id;
  private String firstname;
  private String lastname;
  private Address address;
  private SalesRep salesRep;

  ...
```

```
Customer cust1 = (Customer)
    session.get(Customer.class, 1);
```

Eagerly loads <one-to-one> using a left outer join

```
Hibernate:
    select
        customer0_.id as id0_1_,
        customer0_.firstname as firstname0_1_,
        customer0_.lastname as lastname0_1_,
        customer0_.salesRep as salesRep0_1_,
        address1_.id as id1_0_,
        address1_.street as street1_0_,
        address1_.apt as apt1_0_,
        address1_.city as city1_0_,
        address1_.state as state1_0_,
        address1_.zip as zip1_0_,
        address1_.customer as customer1_0_
    from
        Customer customer0_
    left outer join
        Address address1_
            on customer0_.id=address1_.id
    where
        customer0_.id=?
```

```
<hibernate-mapping package="when.objRefs">
  <class name="Customer">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="firstname" />
    <property name="lastname" />
    <one-to-one name="address" cascade="persist" />
    <many-to-one name="salesRep" />
  </class>
</hibernate-mapping>
```

Collections

- By default the entire collection is retrieved when .size(), .isEmpty(), or .contains() is used
  - Good for small collections, bad for large collections

Customer with a collection of Credit Cards

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  @OneToMany(mappedBy = "customer",
      cascade = CascadeType.PERSIST)
  private Set<CreditCard> creditCards
      = new HashSet<CreditCard>();

  ...
```

Check credit card collection size

```
customer.getCreditCards().size();
```

Retrieves all credit cards

```
Hibernate:
    select
        creditcard0_.customer_id as customer5_1_,
        creditcard0_.id as id1_,
        creditcard0_.id as id1_0_,
        creditcard0_.customer_id as
          customer5_1_0_,
        creditcard0_.expiration as expiration1_0_,
        creditcard0_.name as name1_0_,
        creditcard0_.number as number1_0_
    from
        CreditCard creditcard0_
    where
        creditcard0_.customer_id=?
```

Extra Lazy Collections

- Setting the collection to Extra Lazy solves this problem for large collections

Extra Lazy Collection using annotations

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  @OneToMany(mappedBy = "customer", cascade = CascadeType.PERSIST)
  @org.hibernate.annotations.LazyCollection(
    org.hibernate.annotations.LazyCollectionOption.EXTRA
  )
  private Set<CreditCard> creditCards = new HashSet<CreditCard>();

  ...
```

Extra Lazy Collection using XML mapping

```
<hibernate-mapping package="entities">
  <class name="Customer">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="firstname" />
    <property name="lastname" />
    <set name="creditCards" lazy="extra"
        inverse="true" cascade="persist">
      <key column="customer" />
      <one-to-many class="CreditCard" />
    </set>
  </class>
</hibernate-mapping>
```

```
customer.getCreditCards().size();
```

Only retrieves size

```
Hibernate:
    select
        count(id)
    from
        CreditCard
    where
        customer_id =?
```

Large Properties

- Certain Properties may be so large that you only want to load them when really necessary
  - Lazy loading of properties is only available with byte-code instrumentation

```java
public class Book {
  private String isbn;
  private String title;
  private String author;
  private java.sql.Clob summary;
  private java.sql.Blob cover;

  ...
```

```xml
<hibernate-mapping package="when.lazyprops">
  <class name="Book">
    <id name="isbn" />
    <property name="title" />
    <property name="author" />
    <property name="summary" type="clob" lazy="true" />
    <property name="cover" type="blob" lazy="true" />
  </class>
</hibernate-mapping>
```

Without byte-code instr. lazy=true doesn't do anything

```java
Book b = (Book)session.get(Book.class, "978-0545139700");
```

```
Hibernate:
    select
        book0_.isbn as isbn0_0_,
        book0_.title as title0_0_,
        book0_.author as author0_0_
    from
        Book book0_
    where
        book0_.isbn=?
```

Summary and cover are not loaded (lazy)

Annotations – Lazy Properties

- Requires property access for lazy loading

```
Book b = (Book)session.get(Book.class, "978-0545139700");
System.out.println(b.getTitle());

java.sql.Clob sumData = b.getSummary();
int length = (int)sumData.length();
System.out.println(sumData.getSubString(1, length));
```

```
@Entity
public class Book {
  ...

  @Id
  public String getIsbn() { return isbn; }
  public String getTitle() { return title; }
  public String getAuthor() { return author; }

  @Basic(fetch=FetchType.LAZY)
  public java.sql.Clob getSummary() {
    return summary;
  }
  @Basic(fetch=FetchType.LAZY)
  public java.sql.Blob getCover() {
    return cover;
  }

  ...
```

Annotations on getter methods instead of fields for property access

```
Hibernate:
    select
        book0_.isbn as isbn0_0_,
        book0_.title as title0_0_,
        book0_.author as author0_0_
    from
        Book book0_
    where
        book0_.isbn=?
Harry Potter and the Deathly Hallows
Hibernate:
    select
        book_.summary as summary0_,
        book_.cover as cover0_
    from
        Book book_
    where
        book_.isbn=?
Readers beware. The brilliant,
breathtaking conclusion to J.K.
Rowling's spellbinding series is not for
the faint of heart
```

Only loads summary when needed

Both Summary and Cover are loaded

Byte-Code Instrumentation Ant File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="ByteCodeInstrument" default="instrument">
  <description>Byte Code instrument example</description>
  <property name="src" location="src" />
  <property name="build" location="bin" />

  <target name="compile">
    <javac srcdir="${src}" destdir="${build}" />
  </target>

  <target name="instrument" depends="compile">
    <taskdef name="instrument"
        classname="org.hibernate.tool.instrument.cglib.InstrumentTask">
      <classpath>
        <fileset dir="c:/hibernatetraining/libraries/">
          <include name="**/*.jar" />
        </fileset>
      </classpath>
    </taskdef>
    <instrument verbose="true">
      <fileset dir="${build}/when/properties/">
        <include name="**/*.class" />
      </fileset>
    </instrument>
  </target>
</project>
```

Code needs to be compiled before it can be instrumented

Requires the hibernate libraries

Location of the files that need to be instrumented

FetchType.EAGER

- Specifies ‘when’, not ‘how’

- Can be applied to:
  - Collections @OneToMany, @ManyToMany
  - Object References @ManyToOne, @OneToOne

- Do not recommend using:
  - easy to accidentally create N+1 problems.

FetchType.Eager & Query

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  @ManyToOne(fetch = FetchType.EAGER)
  private SalesRep salesRep;

  @OneToMany(fetch=FetchType.EAGER,
      mappedBy="customer",
      cascade=CascadeType.PERSIST)
  private Set<CreditCard> creditCards =
      new HashSet<CreditCard>();

  ...
```

Creates an N+1 problems without even having a loop!

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

```
Hibernate:
    select
        customer0_.id as id0_,
        customer0_.firstname as firstname0_,
        customer0_.lastname as lastname0_,
        customer0_.salesRep_id as salesRep4_0_
    from
        Customer customer0_
```

N selects, one for each customer retrieved

```
Hibernate:
    select
        salesrep0_.id as id1_0_,
        salesrep0_.name as name1_0_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id=?
```

N selects, one for each customer retrieved

```
Hibernate:
    select
        creditcard0_.customer_id as customer5_1_,
        creditcard0_.id as id1_,
        creditcard0_.id as id2_0_,
        creditcard0_.customer_id as customer5_2_0_,
        creditcard0_.expiration as expiration2_0_,
        creditcard0_.name as name2_0_,
        creditcard0_.number as number2_0_
    from
        CreditCard creditcard0_
    where
        creditcard0_.customer_id=?
```

Eager Collections

- Limit 1 eager collection per entity
  - To avoid creating a Cartesian Product

- When loading the entity that holds the collection
  - By following a reference:
    - The collection is loaded by adding an outer join to the select
  - With a Query that does not Join Fetch the collection
    - The collection(s) is loaded as soon as the result-set is in, using a single select for every entity in the result (N+1)

Eager Object References

- When loading entity that has the reference
  - By following a reference:
    - An additional outer join is added to the select

Default behavior, don't need eager for this

  - With a Query that does not also join to the object that should now be eagerly loaded:
    - The references will be loaded using an additional select statement for each entity in the result-set (N+1)

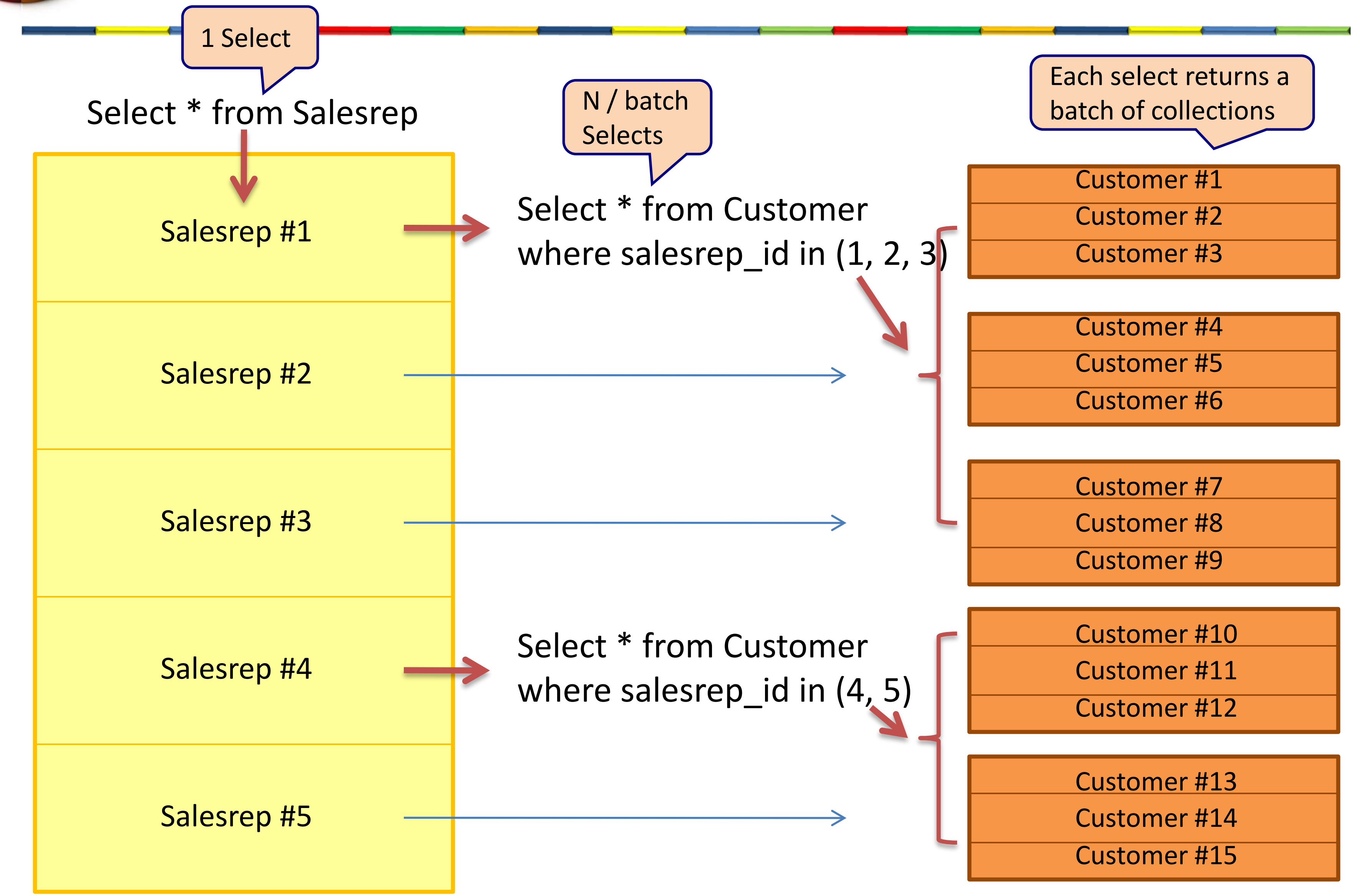
Hibernate Optimization

SUB SELECT & BATCH FETCHING (HIBERNATE SPECIFIC)

Batch Fetching

- Collections N+1:
  - We saw how N+1 loads the customer list for each salesrep using a separate select
  - Batch fetching helps: by loading the customer lists for several salesreps at a time – loading a batch
    - When the first collection is needed
- Lazy References N+1:
  - We also saw how N+1 loads the salesrep for each customer in a separate select
  - Batch fetching helps: by loading the salesrep for several customers simultaniously – loading a batch
    - When the first reference is needed

Batch Fetching Collections



1 Select

Select * from Salesrep

Salesrep #1

Salesrep #2

Salesrep #3

Salesrep #4

Salesrep #5

N / batch Selects

Select * from Customer where salesrep_id in (1, 2, 3)

Select * from Customer where salesrep_id in (4, 5)

Each select returns a batch of collections

Customer #1
Customer #2
Customer #3

Customer #4
Customer #5
Customer #6

Customer #7
Customer #8
Customer #9

Customer #10
Customer #11
Customer #12

Customer #13
Customer #14
Customer #15

Batch Fetching – Collections

```
@Entity
public class SalesRep {
  @Id
  @GeneratedValue
  private int id;
  private String name;

  @OneToMany(mappedBy="salesRep", cascade=CascadeType.PERSIST)
  @org.hibernate.annotations.BatchSize(size=3)
  private Set<Customer> customers = new HashSet<Customer>();

  ...
```

Try to load the customer collection for 3 salesreps when possible

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  @ManyToOne(fetch = FetchType.LAZY)
  private SalesRep salesRep;

  ...
```

```
Hibernate:
    select
        salesrep0_.id as id1_,
        salesrep0_.name as name1_
    from
        SalesRep salesrep0_
Hibernate:
    select
        customers0_.salesRep_id as salesRep4_1_,
        customers0_.id as id1_,
        customers0_.id as id0_0_,
        customers0_.firstname as firstname0_0_,
        customers0_.lastname as lastname0_0_,
        customers0_.salesRep_id as salesRep4_0_0_
    from
        Customer customers0_
    where
        customers0_.salesRep_id in (
            ?, ?, ?
        )
```

```
List<SalesRep> salesreps =
  session.createQuery("from SalesRep").list();
Set<Customer> customers = null;
for (SalesRep s : salesreps) {
  customers = s.getCustomers();
  for (Customer c : customers) {
    // do something with the customer
  }
}
```

Batch fetching only works because Hibernate knows of un-retrieved customer collections

Batch Collections – XML

```xml
<hibernate-mapping package="how.always.batch.collection">
  <class name="SalesRep">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="name" />
    <set name="customers" batch-size="3"
            inverse="true" cascade="persist">
      <key column="salesRep" />
      <one-to-many class="Customer" />
    </set>
  </class>
</hibernate-mapping>
```

XML uses the batch-size attribute on collection tags

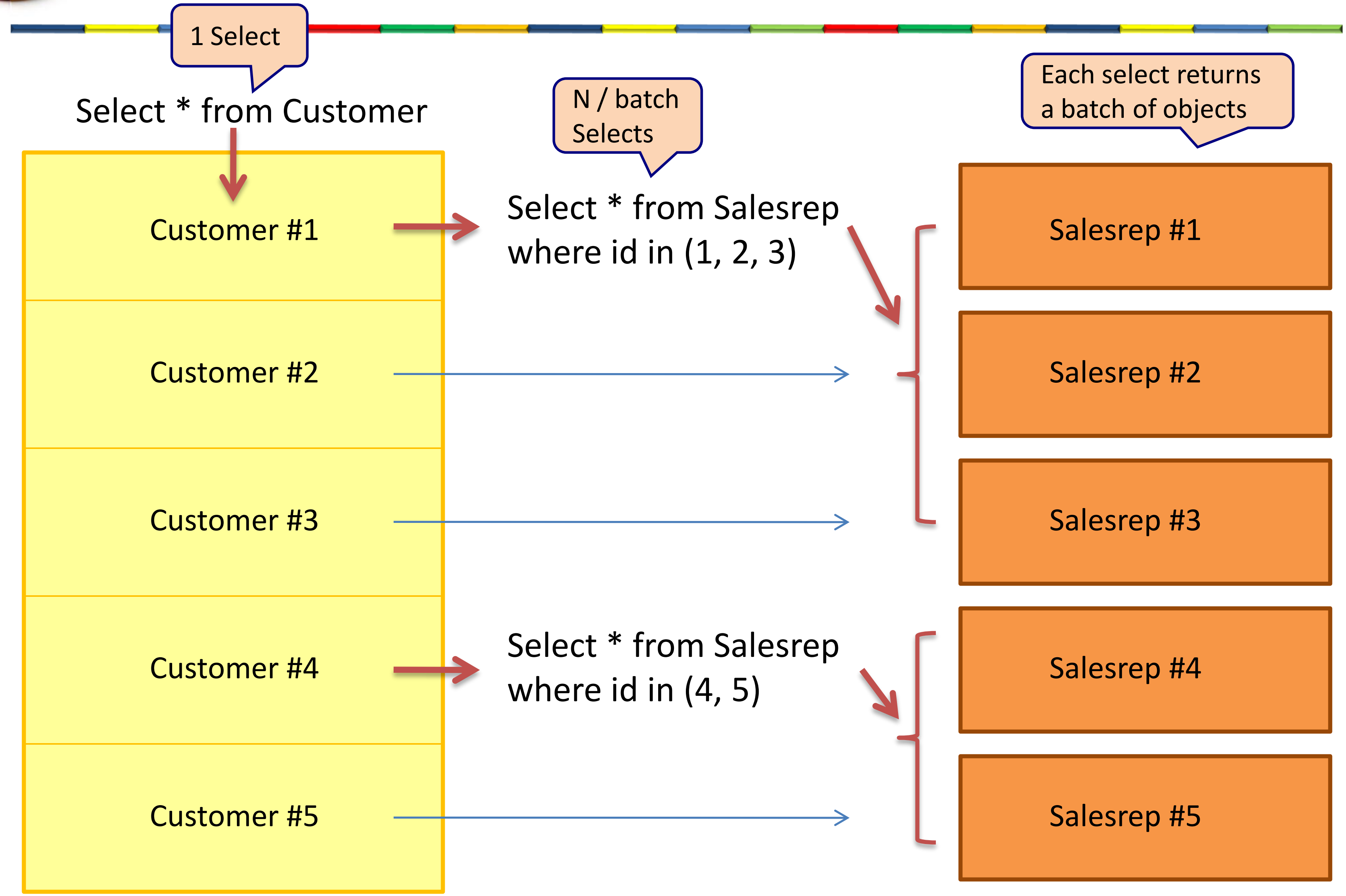
```xml
<hibernate-mapping >
  <class name="Customer">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="firstname" />
    <property name="lastname" />
    <many-to-one name="salesRep" />
  </class>
</hibernate-mapping>
```

```
List<SalesRep> salesreps =
  session.createQuery("from SalesRep").list();
Set<Customer> customers = null;
for (SalesRep s : salesreps) {
  customers = s.getCustomers();
  for (Customer c : customers) {
    // do something with the customer
  }
}
```

Batch fetching only works because Hibernate knows of un-retrieved customer collections

```
Hibernate:
    select
        salesrep0_.id as id1_,
        salesrep0_.name as name1_
    from
        SalesRep salesrep0_
Hibernate:
    select
        customers0_.salesRep_id as salesRep4_1_,
        customers0_.id as id1_,
        customers0_.id as id0_0_,
        customers0_.firstname as firstname0_0_,
        customers0_.lastname as lastname0_0_,
        customers0_.salesRep_id as salesRep4_0_0_
    from
        Customer customers0_
    where
        customers0_.salesRep_id in (
            ?, ?, ?
        )
```

Batch Fetching Lazy References



1 Select

Select * from Customer

N / batch Selects

Each select returns a batch of objects

Customer #1

Customer #2

Customer #3

Customer #4

Customer #5

Select * from Salesrep where id in (1, 2, 3)

Select * from Salesrep where id in (4, 5)

Salesrep #1

Salesrep #2

Salesrep #3

Salesrep #4

Salesrep #5

Batch Lazy References

```
@Entity
@org.hibernate.annotations.BatchSize(size=3)
public class SalesRep {
  @Id
  @GeneratedValue
  private int id;
  private String name;

  @OneToMany(mappedBy="salesRep”)
  private Set<Customer> customers = new HashSet<Customer>();

  ...
```

SalesRep will be loaded in batches of 3 or less, when possible

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  @ManyToOne(fetch = FetchType.LAZY)
  private SalesRep salesRep;

  ...
```

```
List<Customer> customers =
  session.createQuery("from Customer").list();
SalesRep salesrep = null;
for (Customer customer : customers) {
  salesrep = customer.getSalesRep();
  // do something with the salesrep
  salesrep.getName();
}
```

Batch fetching works because customers with un-retrieved salesrep have been loaded

```
Hibernate:
    select
        customer0_.id as id0_,
        customer0_.firstname as firstname0_,
        customer0_.lastname as lastname0_,
        customer0_.salesRep_id as salesRep4_0_
    from
        Customer customer0_
Hibernate:
    select
        salesrep0_.id as id1_0_,
        salesrep0_.name as name1_0_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id in (
            ?, ?, ?
        )
```

Batch References– XML

```
<hibernate-mapping >
  <class name="Customer">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="firstname" />
    <property name="lastname" />
    <many-to-one name="salesRep" />
  </class>
</hibernate-mapping>
```

```
<hibernate-mapping package="how.always.batch.entity">
  <class name="SalesRep" batch-size="3">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="name" />
    <set name="customers" inverse="true" cascade="persist">
      <key column="salesRep" />
      <one-to-many class="Customer" />
    </set>
  </class>
</hibernate-mapping>
```

batch-size attribute
on the <class> tag

```
List<Customer> customers =
  session.createQuery("from Customer").list();
SalesRep salesrep = null;
for (Customer customer : customers) {
  salesrep = customer.getSalesRep();
  // do something with the salesrep
  salesrep.getName();
}
```

Batch fetching works when Hibernate
knows about un-retrieved salereps

```
Hibernate:
    select
        customer0_.id as id0_,
        customer0_.firstname as firstname0_,
        customer0_.lastname as lastname0_,
        customer0_.salesRep_id as salesRep4_0_
    from
        Customer customer0_
Hibernate:
    select
        salesrep0_.id as id1_0_,
        salesrep0_.name as name1_0_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id in (
            ?, ?, ?
        )
```

Batch Fetching

- Batch fetching is an easy and safe optimization
  - If un-needed data is retrieved it’s never much
  - No joins are involved, no Cartesian Product
  - Can be specified for references and collections

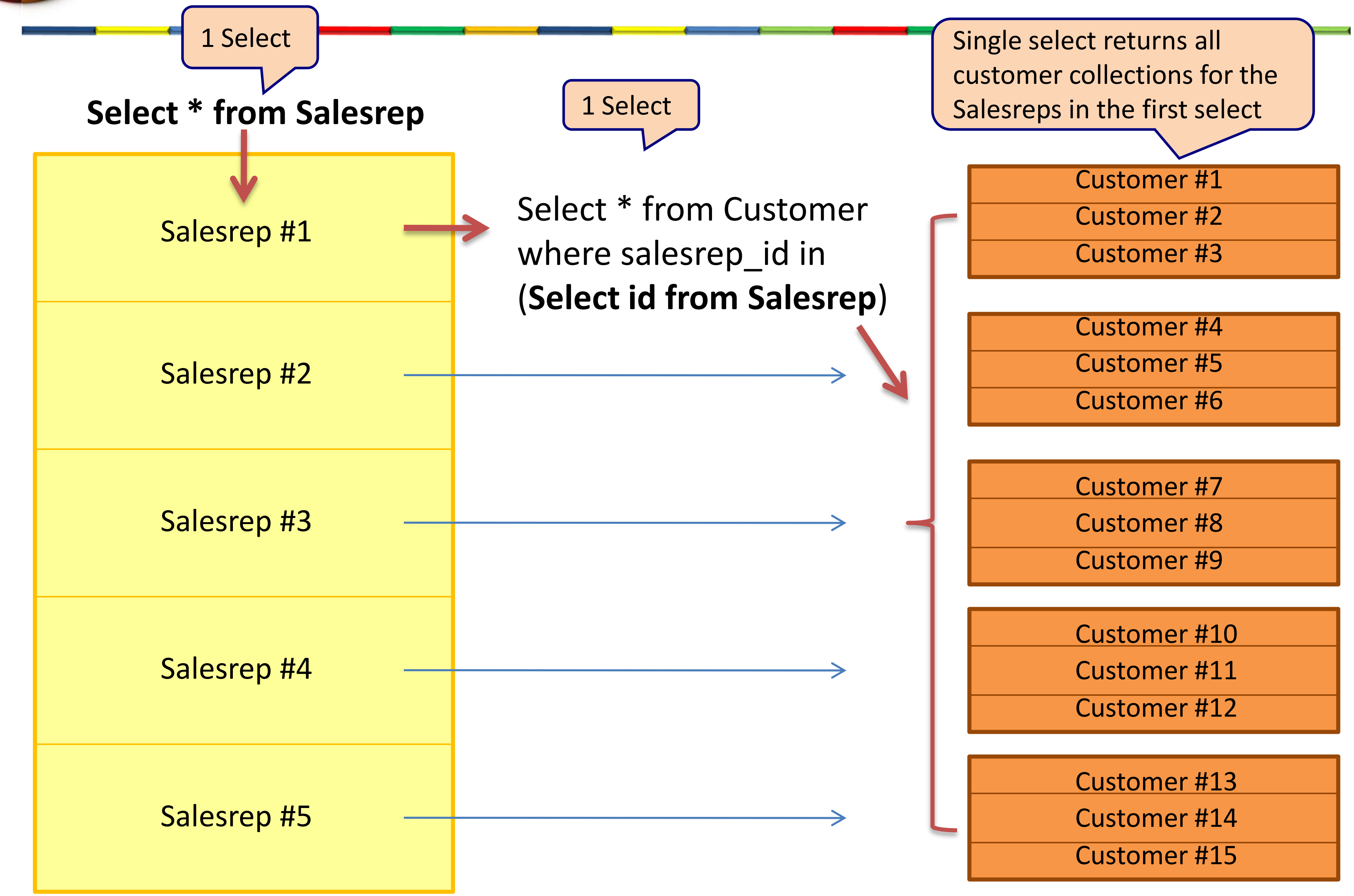
- Typical batch sizes are between 3 and 15

- Batch fetching reduces the N + 1 problem to → Ceil(N / Batch Size) + 1

Sub Select

- The sub select strategy is a specialized form of the batch fetching strategy for collections
    - Instead of loading a batch of collections it loads all related collections in one select
    - Just like batch fetching it doesn’t retrieve anything until the first time a collection is needed
- Sub select is not available for lazy references

Batch Fetching Collections



1 Select

Select * from Salesrep

1 Select

Single select returns all customer collections for the Salesreps in the first select

Salesrep #1

Salesrep #2

Salesrep #3

Salesrep #4

Salesrep #5

Select * from Customer where salesrep_id in (**Select id from Salesrep**)

Customer #1
Customer #2
Customer #3

Customer #4
Customer #5
Customer #6

Customer #7
Customer #8
Customer #9

Customer #10
Customer #11
Customer #12

Customer #13
Customer #14
Customer #15

Sub Select Collections

Keeps track of the query used to retrieve the salesreps

```
Hibernate:
    select
        salesrep0_.id as id1_,
        salesrep0_.name as name1_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id<1000
rnate:
select
        customers0_.salesRep_id as salesRep4_1_,
        customers0_.id as id1_,
        customers0_.id as id0_0_,
        customers0_.firstname as firstname0_0_,
        customers0_.lastname as lastname0_0_,
        customers0_.salesRep_id as salesRep4_0_0_
from
        Customer customers0_
where
        customers0_.salesRep_id in (
            select
                salesrep0_.id
            from
                SalesRep salesrep0_
            where
                salesrep0_.id<1000
        )
```

```
List<SalesRep> salesreps = session.createQuery(
    "from SalesRep where id < 1000").list();
Set<Customer> customers = null;
for (SalesRep s : salesreps) {
  customers = s.getCustomers();
  for (Customer c : customers) {
    // do something with the customer
  }
}
```

Sub-Select eager fetching only works for collections

Re-uses that query as a sub select to get the customer collections for those salesreps

```
@Entity
public class SalesRep {
  @Id
  @GeneratedValue
  private int id;
  private String name;

  @OneToMany(mappedBy="salesRep", cascade=CascadeType.PERSIST)
  @org.hibernate.annotations.Fetch(
    org.hibernate.annotations.FetchMode.SUBSELECT
  )
  private Set<Customer> customers = new HashSet<Customer>();
```

FetchMode.SUBSELECT

Sub Select – XML

```
Hibernate:
    select
        salesrep0_.id as id1_,
        salesrep0_.name as name1_
    from
        SalesRep salesrep0_
    where
        salesrep0_.id<1000
    rnate:
    select
        customers0_.salesRep_id as salesRep4_1_,
        customers0_.id as id1_,
        customers0_.id as id0_0_,
        customers0_.firstname as firstname0_0_,
        customers0_.lastname as lastname0_0_,
        customers0_.salesRep_id as salesRep4_0_0_
    from
        Customer customers0_
    where
        customers0_.salesRep_id in (
            select
                salesrep0_.id
            from
                SalesRep salesrep0_
            where
                salesrep0_.id<1000
        )
```

Keeps track of the query used to retrieve the salesreps

```
List<SalesRep> salesreps = session.createQuery(
    "from SalesRep where id < 1000").list();
Set<Customer> customers = null;
for (SalesRep s : salesreps) {
  customers = s.getCustomers();
  for (Customer c : customers) {
    // do something with the customer
  }
}
```

Sub-Select eager fetching only works for collections

```
<hibernate-mapping package="how.always.subselec
  <class name="SalesRep">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="name" />
    <set name="customers" fetch="subselect"
            inverse="true" cascade="persist">
      <key column="salesRep" />
      <one-to-many class="Customer" />
    </set>
  </class>
</hibernate-mapping>
```

Fetch="subselect"

Re-uses that query as a sub select to get the customer collections for those salesreps

Sub Select

- The Sub Select strategy solves the N + 1 problem by turning it into a 1 + 1
  - Only available for collections, not references
  - May retrieve too much data if you did not actually need to work with the collections
  - Like batch fetching, no joins, no Cartesian Product

- Internally Sub Select keeps track of the query used to retrieve the original objects

Main Point

- Specifying the FetchType as Lazy or Eager can change when Hibernate retrieves the data (Eager has issues, be careful). With Join Fetch, Batch Fetching, and Sub Select we can change how Hibernate retrieves data (tools to alliviate N+1 problems)
- *Science of Consciousness*: in order to do less and accomplish more we have to understand what is going on underneath. Being lazy can often end up causing us to do more.

Hibernate Optimization

FINDING AND SOLVING PROBLEMS

Cartesian Product

- Can be caused by / Look for:
  - An HQL query that joins multiple collections
  - FetchType.EAGER for multiple collections connected to a single entity
    - (Hibernate tries to stop you, but you can still make it)

- Solve by:
  - Fixing your query and/or removing EAGER mappings

N+1 Problems

- Can be caused by / Look for:
  - Looping over a result-set and accessing a collection for each entity
  - FetchType.LAZY for a references, and then looping over a result set, accessing that references
  - FetchType.EAGER for a collection of references, and then not joining them in an HQL query
- Solve by:
  - Have the data loaded (JoinFetch, Batch, SubSelect)

Repeatedly Loading the Same Data

- In case you find your application does not have a Cartesian Product Problem, or N+1 problem…
    - But seems to load the same data again and again

- Solve By:
    - 2nd level caching that data

Complex Mapping

IMMUTABLE ENTITIES

Immutable Entities

- An immutable entity is an entity that
  - Once created, does not change – no updates
  - Hibernate can perform several optimizations

- A Java immutable class:
  - Only has getters methods, no setters
  - Sets all fields in the constructor
  - Gives Hibernate field access

Immutability

```
@Entity
@org.hibernate.annotations.Entity(mutable=false)
public class Payment {
  @Id
  @GeneratedValue
  private final int id;
  private final double amount;
  @Column(name="`to`")
  private final String to;
  @Column(name="`from`")
  private final String from;

  public Payment() {}
  public Payment(double amount, String to, String from) {
    this.amount = amount;
    this.to = to;
    this.from = from;
  }

  public int getId() { return id; }
  public double getAmount() { return amount; }
  public String getTo() { return to; }
  public String getFrom() { return from; }
}
```

Set mutable false using Hibernate Entity extension

Field access through placement of @Id

Data is set in constructor

Getters, but no Setters

XML

Default Field access

```xml
<hibernate-mapping package="immutable" default-access="field">
  <class name="Payment" mutable="false">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="amount" />
    <property name="to" column="`to`" />
    <property name="from" column="`from`"/>
  </class>
</hibernate-mapping>
```

Mutable = false

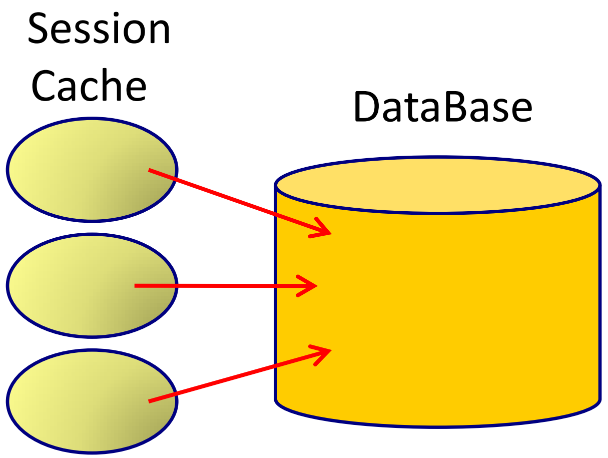
(To and From are SQL keywords)

Hibernate Optimization:

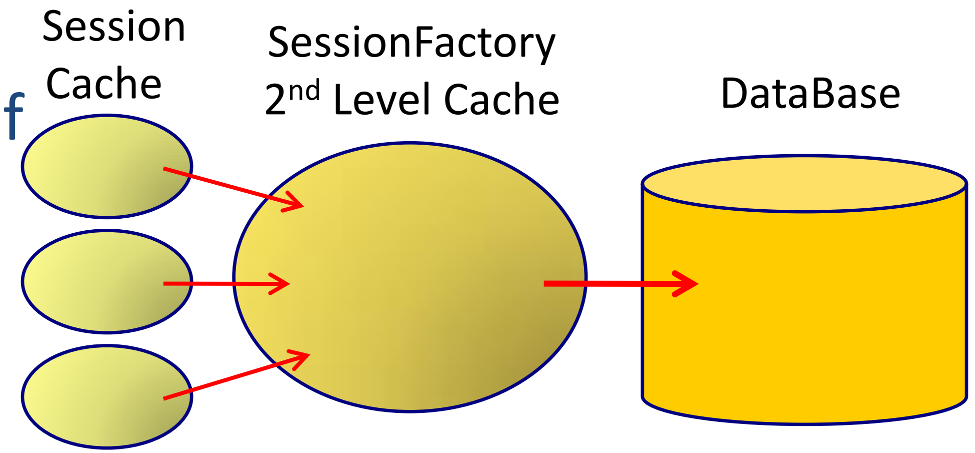
2ND LEVEL CACHING

2nd Level Caching

- By default Hibernate only uses Session Caches
  - Objects are cached for the duration of the session



- You can enable a second level cache
  - Lasts for the duration of the SessionFactory
  - Shared by all sessions



Caching and Optimization

- 2nd Level caching should never be used as an alternative to fetch optimizations
  - Can not solve problems, can attempt to hide them
  - Should be used to help scale the application

- Caching is a large and complex field
  - We will cover Hibernates basic caching features
  - Improper configuration can create problems that are difficult to debug

What to Cache

- Hibernate can cache entity objects and collections (collections of entity IDs)
  - But not all of them will benefit from being cached
- Good candidates for caching :
  - Do not change, or change rarely
  - Are modified only by your application
  - Are non-critical to the application
- Typical examples include reference data
  - Such as customer categories, or statuses

Caching Strategies

- Four different caching strategies:
  - Read Only: very fast caching strategy, but can only be used for data that never changes
  - Non Strict Read-Write: data may be stale for a while, but it does get refreshed at 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

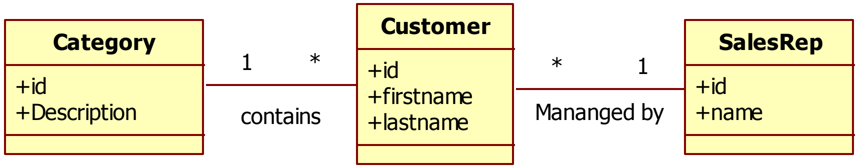
Stricter and therefore Slower

Cache Providers

- The following open source cache providers are bundled with Hibernate
- Only a single cache provider per SessionFactory

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

Caching Example – Entities



Category
+id
+Description

1 * contains

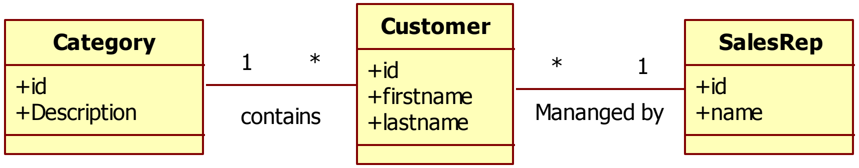
Customer
+id
+firstname
+lastname

* 1 Mananged by

SalesRep
+id
+name

- Category entities – Read Only
  - Typical reference data, categories are never updated
- SalesRep entities – Non Strict Read Write
  - Not many Salesreps, always needed when editing customers
  - SalesReps seldom change, stale SalesRep records are fine
- Customer entities – Not Cached
  - Too many customers, customer are updated frequently

Caching Example – Collections



- Customer Collection for each category – Read Write
  - Often used, try to avoid stale data as much as possible
- Customer Collection for each SalesRep – Not Cached
  - Not used frequently enough to warrant caching

Category

```
@Entity
@org.hibernate.annotations.Entity(mutable=false)
@org.hibernate.annotations.Cache(usage=
    CacheConcurrencyStrategy.READ_ONLY
)
public class Category {
  @Id
  private String abbreviation;
  private String description;

  @OneToMany(mappedBy="category")
  @org.hibernate.annotations.Cache(usage=
      CacheConcurrencyStrategy.READ_WRITE
  )
  private Set<Customer> customers = new HashSet<Customer>();

  ...
```

Mutable=false indicates to Hibernate that Categories can never change

Specify read only caching for Category Entities

Specify read write caching for the collection of customers each category has

```
<hibernate-mapping package="cacheDemo">
  <class name="Category" mutable="false">
    <cache usage="read-only" />
    <id name="abbreviation" />
    <property name="description" />
    <set name="customers" inverse="true" cascade="persist">
      <cache usage="read-write" />
      <key column="salesRep" />
      <one-to-many class="Customer" />
    </set>
</class>
</hibernate-mapping>
```

Mutable=false insinde <class> tag

XML uses <cache> tag inside <class> to specify category entity caching

<cache> tag inside <set> for the customers collection

SalesRep

```
@Entity
@org.hibernate.annotations.Cache(usage=
    CacheConcurrencyStrategy.NONSTRICT_READ_WRITE
)
public class SalesRep {
  @Id
  @GeneratedValue
  private int id;
  private String name;

  @OneToMany(mappedBy="salesRep", cascade=CascadeType.PERSIST)
  private Set<Customer> customers = new HashSet<Customer>();

  ...
```

Specify non strict read write caching for SalesRep Entities

```
<hibernate-mapping package="cacheDemo">
  <class name="SalesRep">
    <cache usage="nonstrict-read-write" />
    <id name="id">
      <generator class="native" />
    </id>
    <property name="name" />
    <set name="customers" inverse="true" cascade="persist">
      <key column="salesRep" />
      <one-to-many class="Customer" />
    </set>
  </class>
</hibernate-mapping>
```

XML uses the <cache> tag inside the <class> tag

Enabling Caching (EHCache)

```
<hibernate-configuration>
  <session-factory>
    <!-- HSQL DB running on localhost -->
    <property name="connection.url">jdbc:hsqldb:hsql://localhost/trainingdb</property>
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

    <!-- Enable Second Level Cache -->
    <property name="cache.provider_class">org.hibernate.cache.EhCacheProvider</property>

    <!-- Enable Statistics -->
    <property name="generate_statistics">true</property>

    <!-- Hibernate XML mapping files - Cache -->
    <mapping resource="cacheDemo/Customer.hbm.xml" />
    <mapping resource="cacheDemo/SalesRep.hbm.xml" />
    <mapping resource="cacheDemo/Category.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

Enable 2nd level caching by specifying a caching provider

Optionally enable statistics

Configuring EHCache – Cache Eviction

```
<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>
```

EHCache General configuration

Sets up a cache region for category entities

Sets up a cache region for the customer collections inside the category entities

cache region for the SalesRep entities

Hibernate Statistics

Stats object also holds general statistics for many Hibernate subsystems

General 2nd level cache statistics

```
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);
```

cache statistics for a specific cache region

```
SecondLevelCacheStatistics salesRepStats =
    stats.getSecondLevelCacheStatistics("cacheDemo.SalesRep");
long srCurrent = salesRepStats.getElementCountInMemory();
long srMemsize = salesRepStats.getSizeInMemory();
long srHits    = salesRepStats.getHitCount();
long srMisses  = salesRepStats.getMissCount();
long srPuts    = salesRepStats.getPutCount();
System.out.printf("\nSalesRep Cache Region - Size: %d Holds: %d\n", srMemsize, srCurrent);
System.out.printf("Hit: %d Miss: %d Put: %d\n", srHits, srMisses, srPuts);
```

```
Statistics stats = sessionFactory.getStatistics();
Stats.clear();
stats.setStatisticsEnabled(true);

...

stats.setStatisticsEnabled(false);
```

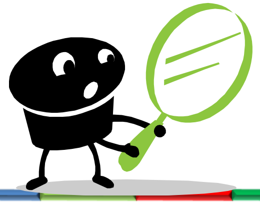
Statistics can also be enabled or disabled programmatically allowing you to do more targeted measurements

Main Point

- 2nd Level Caching can eliminate repeated requests to the database for the same data. Be careful though that you don’t create stale cache. Plus to find out how well your cache is really working it’s highly recommended to look at production cache statistics.
- *Science of Consciousness*: Rest and Activity are the steps of progress, don’t retrieve data again if you can just keep it from last time.

Hibernate Optimization:

WRAPPING UP



Analyze SQL

- Before changing any fetching strategies
  - Analyze the SQL Hibernate uses for all use cases
  - Look for things that can actually cause problems
    - Don’t over optimize, only update real problem areas

- Then after each change check the SQL again

```
<hibernate-configuration>
  <session-factory>

    <property name="show_sql">true</property>
    <property name="format_sql">true</property>
    <property name="use_sql_comments">true</property>

    ...

  </session-factory>
</hibernate-configuration>
```

The following three property can be used to check Hibernates SQL

Active Learning

- Describe the difference between batch fetching and sub select optimization.

- Why doesn't second level caching fix bad fetching strategies?

Module Summary

- Data Access Optimization changes when and how Hibernate retrieves data
- Hibernate mostly defaults to lazy loading
  - Lazy loading can lead to too many small selects
  - Incorrect eager loading can lead to slow queries
- 2nd level caching should not be used as an alternative to fetch optimizations
  - Caching can help boost performance under load
  - Incorrectly configured cache can create problems