10 common pitfalls when using Hibernate

Marek Matczak, Capgemini

Software Solutions Center, Wrocław

Marek Matczak

- Graduated from Poznań University of Technology
- Since 2004 working for Capgemini
- Supporting various software development projects as a team leader and solution architect

Capgemini
Software Solutions Center Wrocław

- Almost 350 programmers, designers, architects, testers, project managers and consultants
- Projects from different sectors such as financial services, logistics, automotive and telecommunications

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING GeeCON

Hibernate: we‘ve come a long way...

- The main benefits of Hibernate are:
  - business domain model may be designed in an object oriented way
  - convenient APIs for accessing a database, e.g.: HQL, Criteria API

- Impression: it solves all my problems with the database access!

But the truth is that…

- although some problems are really solved,
- others are not, as they are not addressed by ORM tools,
- and some new appear because Hibernate is used incorrectly!
- Consequently, JDBC based applications may be more efficient than those ones using Hibernate!

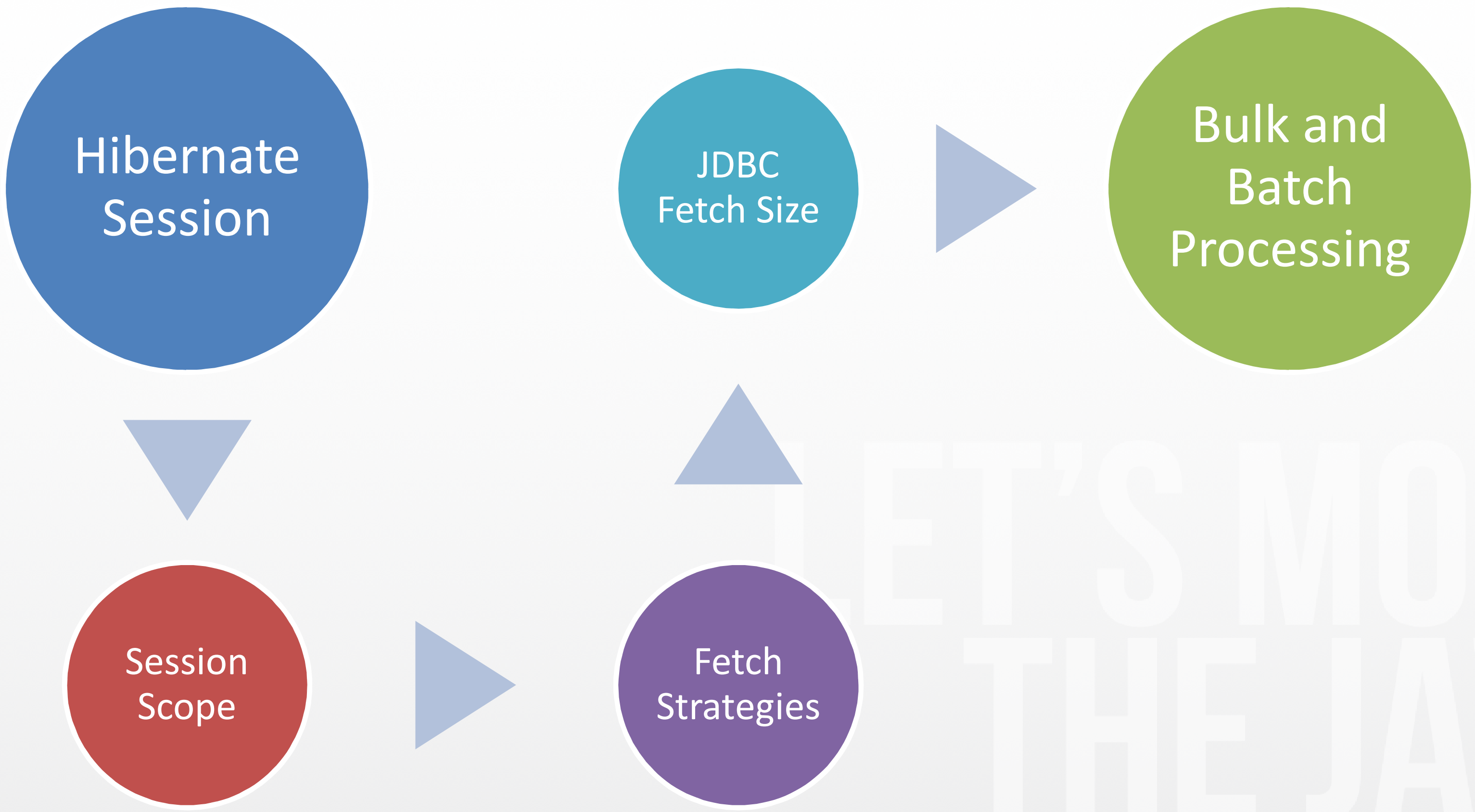
Hibernate: how to use it right?

Developers shouldn't expect to sprinkle magic pixie dust on POJOs in hopes they will become persistent

Dan Allen, *Seam in Action*

You can't use Hibernate successfully without paying attention to what is going on behind the scenes!

Hibernate topic map



HIBERNATE SESSION

LET'S MOVE THE JAVA WORLD

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

GeeCON

Hibernate session …

- is the main runtime interface between a Java application and Hibernate
- holds a persistence context (first-level-cache)
- does automatic dirty checking
- does transactional write-behind
- is not thread safe; each thread should obtain its own instance from a `SessionFactory`

Hibernate session: what is happening behind the scenes?

```
Session session = getSession();
session.beginTransaction();

Order order = (Order)
    session.load(Order.class, 1234L);

order.setReferenceNo("NewRef");

session.getTransaction().commit();
session.close();
```

1. A proxy is put in the persistence context

2. The proxy is initialized:

```
select
    …
from
    GC_ORDER order0_
where
    order0_.id=?
```

3. A snapshot (duplicate) is also held for dirty checking

4. The order is dirty, so update it:

```
update
    GC_ORDER
set
    customer_id=?,
    referenceNo=?
where id=?
```

Hibernate session: hints (1)

Consider if data queried needs really be fully managed by the persistence context

- some objects do not need dirty checking, e.g.:
  - master data

    Query.setReadOnly (), Criteria.setReadOnly ()
  - entities mapped to database views

```
@Entity
@org.hibernate.annotations.Entity(mutable = false)
@Table(name = "CUSTOMER_VIEW")
public class Customer {
```

Hibernate session: hints (2)

– report data does not need the persistence context

```
// DON'T DO THIS: all orders (and customers) are put in the persistence context
Query query = session.createQuery("from Order where referenceNo like :refNo");
query.setParameter("refNo", "ABC%");
List<Order> orders = query.list();

List<MyReport> myReports = new ArrayList<MyReport>();
for (Order order : orders) {
    myReports.add(
        new MyReport(order.getCustomer().getName(), order.getReferenceNo()));
}
```

```
// nothing is managed by the persistence context
Query query = session.createQuery(
    "select new MyReport(customer.name, referenceNo) " +
    "from Order where referenceNo like :refNo");
query.setParameter("refNo", "ABC%");
List<MyReport> myReports = query.list();
```

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING GeeCON

Hibernate pitfalls

1. Don’t put unnecessary data in the session context
2. Disable dirty-checking for read-only objects
3.
4.
5.
6.
7.
8.
9.
10.

http://www.projektwerk.com/blog/freelance/category/trends

HIBERNATE SESSION SCOPE

LET'S MOVE THE JAVA WORLD

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

GeeCON

Hibernate session scope

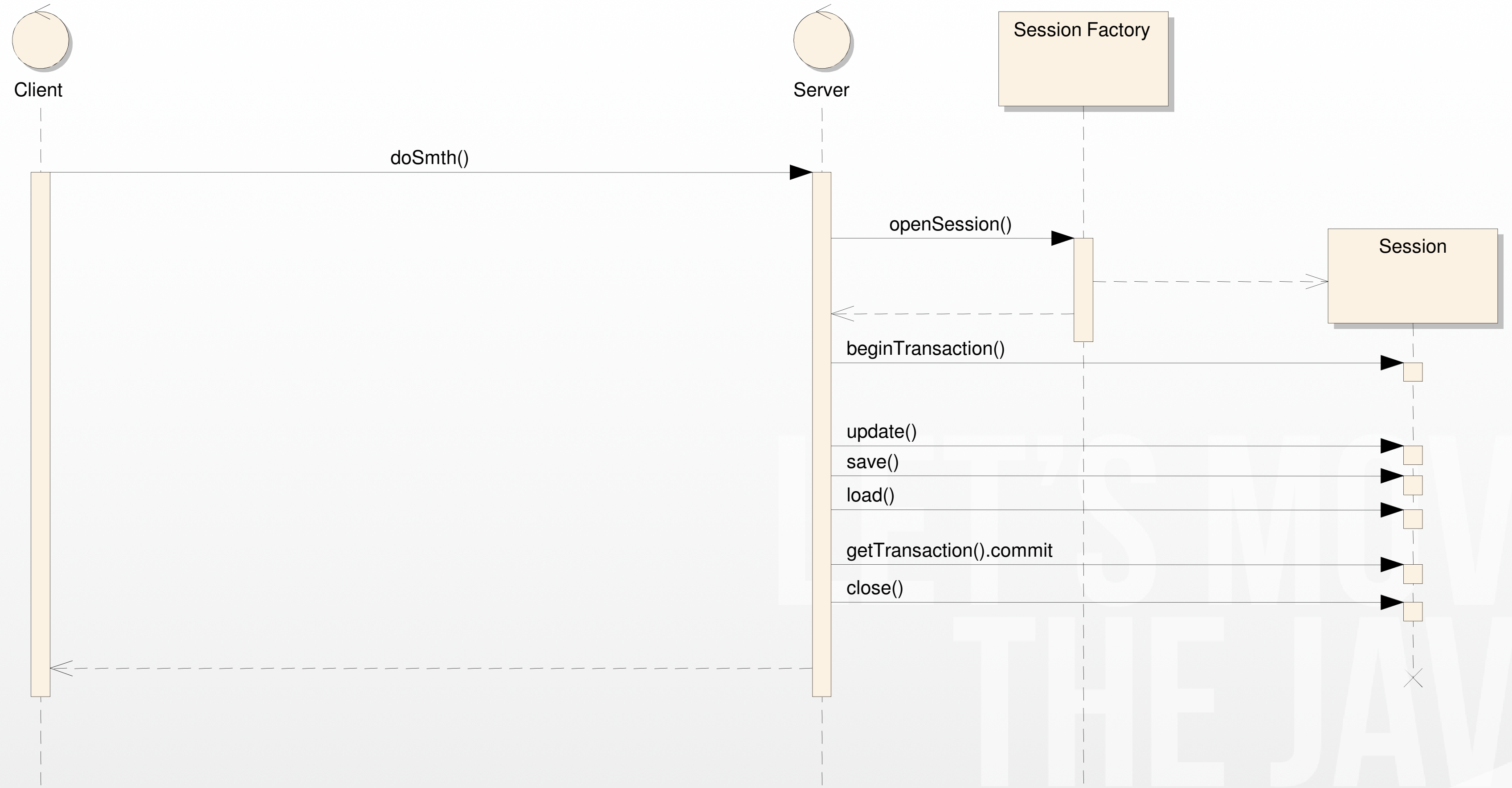
Hibernate session may be scoped to

– an operation (anti-pattern)
– a request / transaction
– a conversation

(...) when the persistence manager is scoped incorrectly, Java persistence can become more of a hindrance than a help

Dan Allen, *Seam in Action*

Session per request



Session per request: consequences

- The client operates on *detached* objects
- `LazyInitializationException`s may occur
- The detached objects are reattached to / merged with another session
- Typical problems in the client:
  - implementing conversations
  - deleting collections

Session per request

- The best choice for application architectures where the code that accesses data using Hibernate, and the code that uses it are in different application layers or different physical processes
- A common solution for opening and closing the session is an AOP interceptor with a pointcut on service methods
- In such a case make sure only one service method is called per client operation, e.g. displaying a concrete order. If you need to call more services, implement a service which groups the services' calls

Session per request: transactions (1)

```
public class OrderController {
    private IOrderService orderService;

    public Map<String, Object> loadOrderReferenceData(Long orderId) {
        Map<String, Object> model = new HashMap<String, Object>();
        model.put("someOrderData",
                orderService.getSomeOrderRelatedData(orderId));
        model.put("sotherOrderData",
                orderService.getOtherOrderRelatedData(orderId));

        return model;
    }
}
```

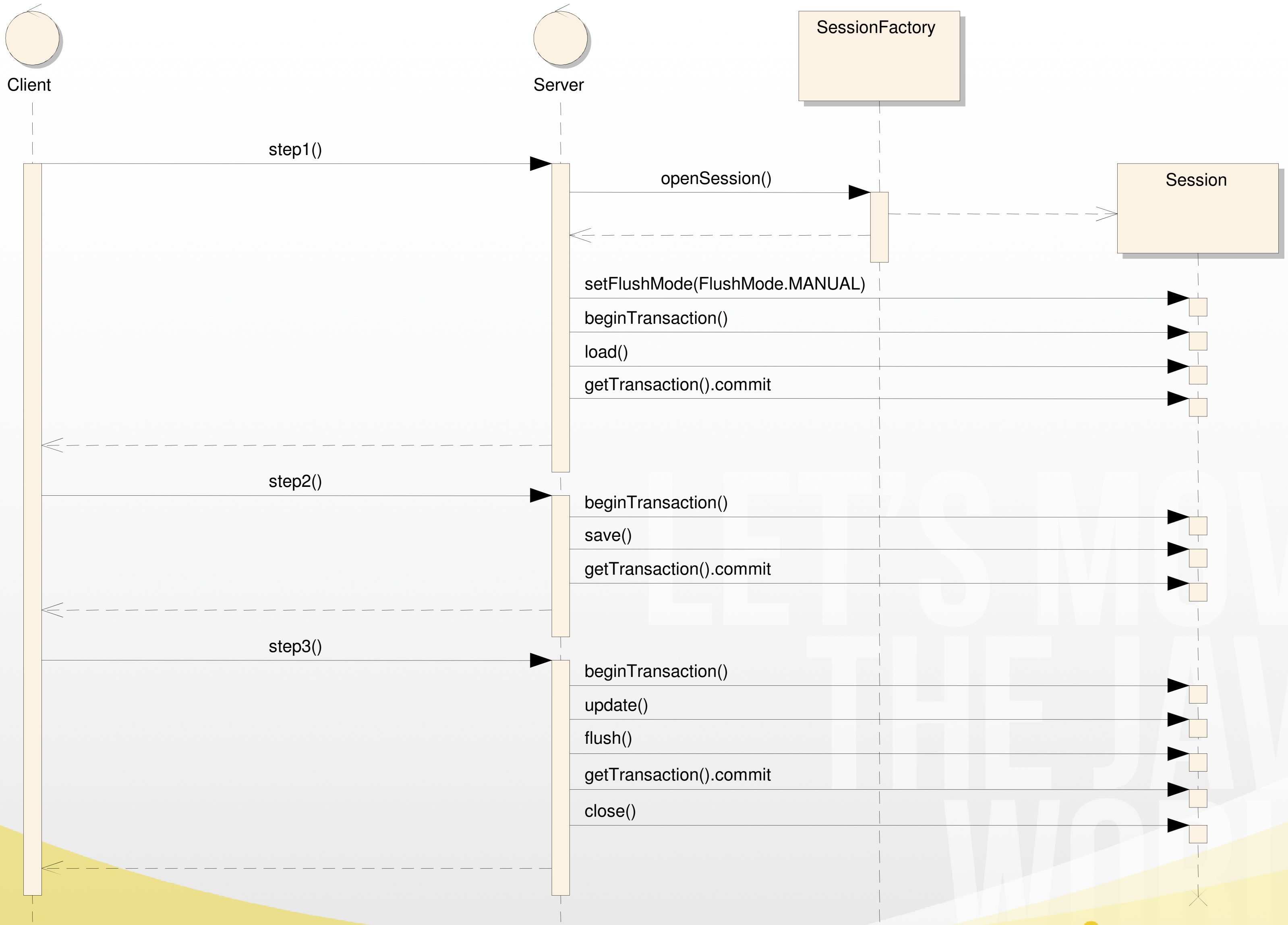
1. New transaction
2. New transaction

Session per request: transactions (2)

```
public class OrderController {
    private IOrderService orderService;

    public Map<String, Object> loadOrderReferenceData(Long orderId) {
        Map<String, Object> model = new HashMap<String, Object>();
        OrderReferenceData orderData =
            orderService.getAllOrderRelatedData(orderId);

        model.put("someOrderData",  orderData getSomeOrderRelatedData());
        model.put("sotherOrderData", orderData.getOtherOrderRelatedData());
        return model;
    }
}

public class OrderService implements IOrderService {
     public OrderReferenceData getAllOrderRelatedData(Long orderId) {
        return new OrderReferenceData(
                    getSomeOrderRelatedData(orderId),
                    getOtherOrderRelatedData(orderId));
     }
```

1. New transaction

Session per conversation



Session per conversation: consequences

- No more detached objects / LIEs / reattaching / merging!
- Application architectures require that both the code that accesses data and the code that uses it are in the same physical process
- Can be problematic if the session is too big to be stored during user think time
- Is more difficult to implement with automatic current session context management

Hibernate pitfalls

http://www.projektwerk.com/blog/freelance/category/trends

1. Don’t put unnecessary data in the session context
2. Disable dirty-checking for read-only objects
3. Don’t follow the session-per-operation anti-pattern
4. In the session-per-request pattern use one database transaction
5. Choose the session scope fitting your architecture
6.
7.
8.
9.
10.

FETCH STRATEGIES

Fetch strategies: n+1 selects problem

```
// find orders with references starting with „ABC“
List<Order> orders = session.createCriteria(Order.class)
        .add(Restrictions.like(
                "referenceNo", "ABC", MatchMode.START))
        .list();
// order list has now n elements
```

```
select
    ...
from
    GC_ORDER this_
where
    this_.referenceNo like ?
```

```
// for each order calculate total quantity
for (Order order : orders) {
    long totalQty = 0;
    for (OrderPosition position
                : order.getPositions()) {
        totalQty += position.getQuantity();
    }
    print(order.getReferenceNo(), totalQty);
}
```

n times

```
select
    ...
from
    GC_ORDER_POSITION positions0_
where
    positions0_.order_id=?
```

Fetch strategies: global vs. dynamic

- Global – configured in mapping metadata

```
@OneToMany(mappedBy = "order", fetch = FetchType.EAGER)
private Set<OrderPosition> positions = new HashSet<OrderPosition>();
```

- Dynamic - set while executing queries

```
List<Order> orders = session.createCriteria(Order.class)
    .add(Restrictions.like("referenceNo", "ABC", MatchMode.START))
    .setFetchMode("positions", FetchMode.JOIN)
    .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY)
    .list();
```

Fetch strategies: hints

- Start with disabling global fetch strategies; watch the difference in handling many-to-one relationships between XML and annotations

```
// defaults to EAGER; set LAZY explicitly
@ManyToOne(fetch = FetchType.LAZY)
private Customer customer;
```

- Ask yourself: what SQL queries would you write in a given use case scenario?
- Use dynamic fetching strategies dedicated to various use case scenarios

Fetch strategies:
queries dedicated to various use cases

```
public enum OrderRelatedData { CUSTOMER, POSITIONS; }

public Order getOrder(Long id, Set<OrderRelatedData> dataToFetch) {
    Criteria criteria = getSession().createCriteria(Order.class).add(Restrictions.eq("id", id));

    if (dataToFetch.contains(OrderRelatedData.CUSTOMER))
        criteria.setFetchMode("customer", FetchMode.JOIN);
    if (dataToFetch.contains(OrderRelatedData.POSITIONS))
        criteria.setFetchMode("positions", FetchMode.JOIN);
    return (Order) criteria.uniqueResult();
}

Order order = getOrder(1234l, EnumSet.of(OrderRelatedData.CUSTOMER))
order.getCustomer(); // no database hit; no LazyInitializationException
```

Hibernate pitfalls

http://www.projektwerk.com/blog/freelance/category/trends

1. Don’t put unnecessary data in the session context
2. Disable dirty-checking for read-only objects
3. Don’t follow the session-per-operation anti-pattern
4. In the session-per-request pattern use one database transaction
5. Choose the session scope fitting your architecture
6. Don’t run into the n+1 selects problem
7. Write HQL/Criteria API queries dedicated to various use cases
8.
9.
10.

FETCH SIZE: JDBC IS STILL THERE

JDBC fetch size: real life example (1)

- User may define complex search criteria
- A specialized table was introduced which
  - contains search criteria (redundant)
  - is updated asynchronously after each model change
- Searching was executed in two steps:
  - The search table was queried to get technical IDs of items matching the search criteria
  - Other tables were queried to get the items‘ data

JDBC fetch size: real life example (2)

- A problem while querying the search table occurred: its execution from within the Java application took noticeably longer (80%) comparing to running the SQL query using a database SQL command tool
- Guilty was Hibernate, of course…

JDBC fetch size

- Gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed for `ResultSet` objects (*JDBC API Documentation*)
- With `ResultSet` objects Hibernate is working behind the scenes; consequently you can make use of the fetch size hint in your queries
- The problem was solved by setting the fetch size to 1000 (Oracle driver defaults to 10):

```
criteria.setFetchSize(1000);
```

Hibernate pitfalls

http://www.projektwerk.com/blog/freelance/category/trends

1. Don’t put unnecessary data in the session context
2. Disable dirty-checking for read-only objects
3. Don’t follow the session-per-operation anti-pattern
4. In the session-per-request pattern use one database transaction
5. Choose the session scope fitting your architecture
6. Don’t run into the n+1 selects problem
7. Write HQL/Criteria API queries dedicated to various use cases
8. Don’t ignore the underlying JDBC API
9.
10.

BULK AND BATCH OPERATIONS

Bulk operations

- HQL offers bulk statements which enable to modify objects without loading them into the persistence context, e.g.:

```
Query query = session.createQuery(
        "delete from OrderPosition where order.id = :ordId");
query.setParameter("ordId", 6789l);
query.executeUpdate();
```

- Bulk statements bypass the persistence context, that is, their changes are not reflected in it; just execute them first, in a fresh persistence context

Batch operations

- Make use of the Hibernate session judiciously
- In iterations use `Session.clear()`; consider using `Query.scroll()`
- Make use of the batch mode (since JDBC 2.0)

The idea behind ORM is good if

(...) you're implementing a multiuser ***online*** *transaction processing application, with* ***small to medium*** *size data sets involved in each unit of work*

Christian Bauer, Gavin King,
Java Persistence with Hibernate

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
GeeCON

Hibernate pitfalls

http://www.projektwerk.com/blog/freelance/category/trends

1. Don’t put unnecessary data in the session context
2. Disable dirty-checking for read-only objects
3. Don’t follow the session-per-operation anti-pattern
4. In the session-per-request pattern use one database transaction
5. Choose the session scope fitting your architecture
6. Don’t run into the n+1 selects problem
7. Write HQL/Criteria API queries dedicated to various use cases
8. Don’t ignore the underlying JDBC API
9. Use bulk statements when modifying large amount of data
10. Don’t use Hibernate for complex batch operations

At the end of the day, Hibernate is great!

http://www.projektwerk.com/blog/freelance/category/trends

1. Don't put unnecessary data in the session context
2. Disable dirty-checking for read-only objects
3. Don't follow the session-per-operation anti-pattern
4. In the session-per-request pattern use one database transaction
5. Choose the session scope fitting your architecture
6. Don't run into the n+1 selects problem
7. Write HQL/Criteria API queries dedicated to various use cases
8. Don't ignore the underlying JDBC API
9. Use bulk statements when modifying large amount of data
10. Don't use Hibernate for complex batch operations

THANK YOU VERY MUCH FOR YOUR ATTENTION!

ANY QUESTIONS?

LET'S MOVE THE JAVA WORLD

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

GeeCON

Resources

- Bauer, Christian, and Gavin King. 2006. *Java Persistence with Hibernate*. *Manning Publications*.
- Allen, Dan, 2009. *Seam in Action*. *Manning Publications*.
- http://docs.jboss.org/hibernate/core/3.5/reference/en-US/pdf/hibernate_reference.pdf