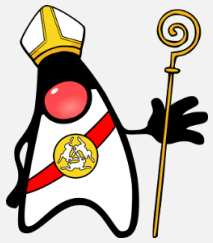
Hibernate in complex projects
Can we be a little faster?

Independent consultant,
trainer and author

Thorben Janssen

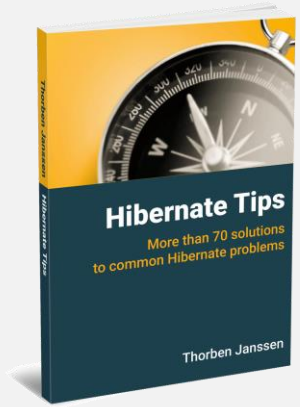


CDI 2.0 Expert
Group Member

Co-Organizer
JUG Paderborn

@thjanssen123
/c/ThoughtsOnJava
/thorbenjanssenofficial

www.thorben-janssen.com



Hibernate Tips
More than 70 solutions to common Hibernate problems
www.hibernate-tips.com

Performance

Performance

- Find performance problems as early as possible
    - Typical causes for performance problems
- Fix performance problems

Hibernate Statistics

Hibernate Statistics

- Activate via system property
  - *hibernate.generate_statistics = true*
- Configure logging
  - *org.hibernate.stat = DEBUG*

Demo

Typical causes for performance problems

Typical Causes

- Inefficient mappings
- Slow SELECT statements
- Wrong FetchType
- Load same data multiple times

Inefficient mappings

Many-to-Many

- Inefficient handling of List
  - Remove all associations
  - Add remaining ones
- Use Set instead

Demo

Slow SELECT-statements

Slow SELECTs

- In general no „real“ JPA or Hibernate problem
  - Check generated SQL
  - Check execution plan of the statement
  - Check indexes
  - Optimize SELECT statement
    - Consider to use a native query

Native Query

- Reasons to use native queries:
  - JPQL supports only a subset of SQL
  - Database specific features
- Native queries return an Object[] for each row
  - Needs to be mapped programmatically or declaratively

Demo

FetchType

FetchType

- Defines when the relationship will be fetched
- Static definition in entity mapping

```
@ManyToMany(mappedBy="authors", fetch = FetchType.LAZY)
private Set<Book> books;
```

FetchType

- Lazy
  - Relationship gets loaded at first access
  - Default for to-many relationships
- Eager
  - Loads relationships immediately
  - Default for to-one relationships

Recommendations

- To-many relationships
  - Stick to the default mapping (FetchType.LAZY)
  - Use eager fetching for specific queries, if required
- To-one relationships
  - Check individually
  - Change to FetchType.LAZY if necessary

Query Specific Fetching

N+1 Select?

- Most common cause for performance problems
- Lazy fetching of related entities creates too many queries

```
List<Author> authors = this.em.createQuery("SELECT a FROM Author a",
                                                    Author.class).getResultList();

for (Author a : authors) {
        System.out.println("Author " + a.getFirstName() + " " + a.getLastName()
                                + " wrote " + a.getBooks().size() + " Books."));
}
```

Query specific fetching

- Fetch all required entities with one query
  - Fetch Joins
  - @NamedEntityGraph / EntityGraph

Fetch Join

Fetch Join

- Use JOIN FETCH instead of JOIN in JPQL query

```
List<Author> authors = this.em.createQuery(
                    "SELECT DISTINCT a FROM Author a JOIN FETCH a.books b",
                    Author.class).getResultList();
```

Demo

Fetch Join

- Advantages
  - Relationships gets loaded in same query
- Disadvantages
  - Requires a special query for each use case
  - Creates cartesian product

NamedEntityGraph

NamedEntityGraph

- Introduced in JPA 2.1
- Declaratively defines a graph of entities which will be loaded
- Graph is query independent

Demo

NamedEntityGraph

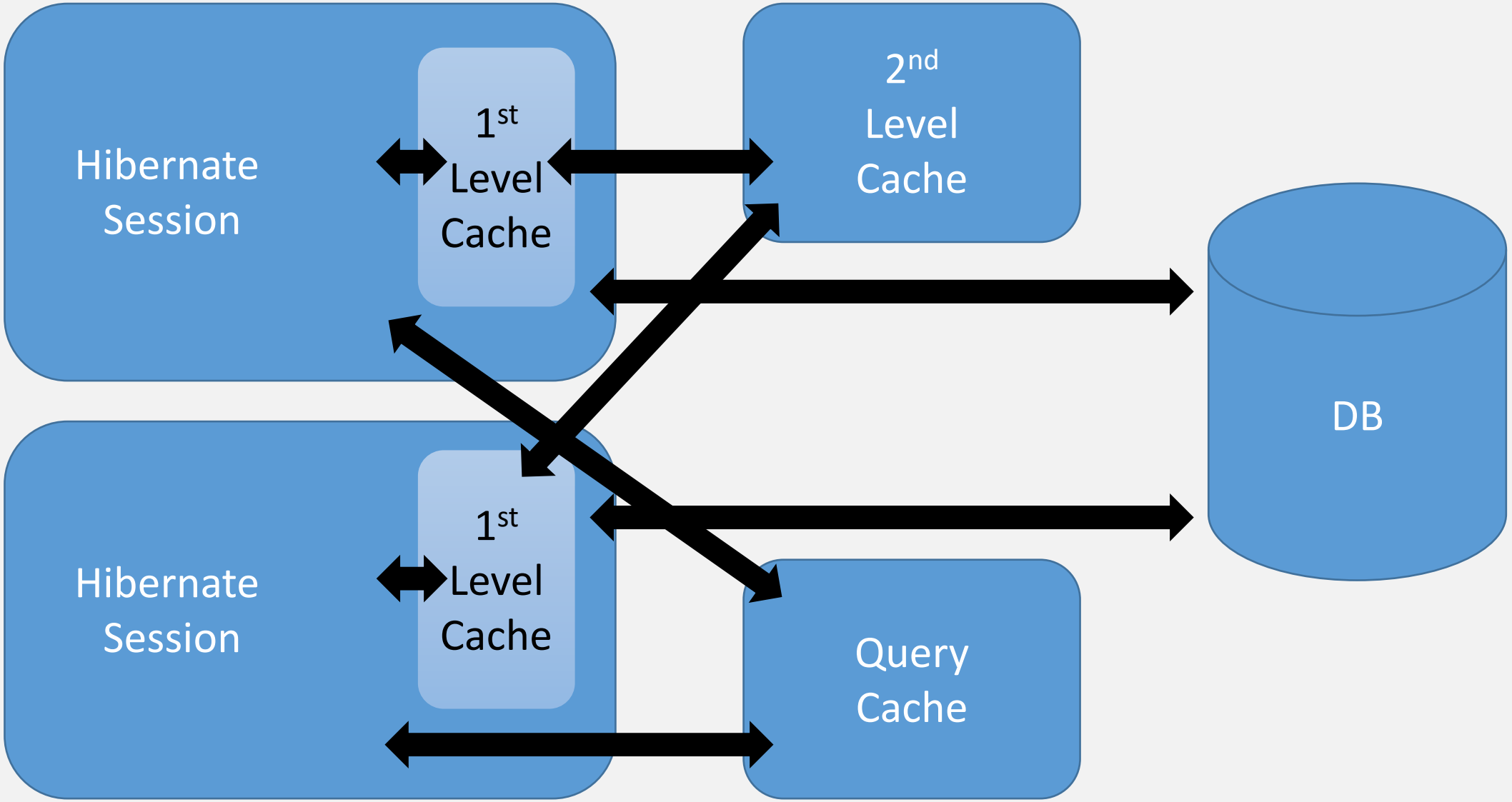
- Fetch graph
  - Eager loading for all elements of the graph
  - Lazy loading for all other attributes
- Load graph
  - Eager loading for all elements of the graph
  - Loads all other attributes with their defined FetchType

NamedEntityGraph

- Advantages
  - Query specific EAGER loading
  - Definition of the graph is independent of the query
- Disadvantages
  - Creates a product

Caching

Caches


Hibernate Session
1st Level Cache
2nd Level Cache
DB
Hibernate Session
1st Level Cache
Query Cache

1st Level Cache

1st Level Cache

- Activated by default
- Linked to the Hibernate session
- Stores all entities that were used within a session
- Transparent usage

2nd Level Cache

2nd Level Cache

- Session independent entity store
- Needs to be activated
  - persistence.xml or EntityManagerFactory
- Transparent usage
- PersistenceProvider doesn‘t need to provide it
  - Not always portable

2nd Level Cache

- Shared Cache Mode
  - ALL — cache all entities
  - NONE — cache no entities
  - ENABLE_SELECTIVE — cache needs to be activated for specific entities
  - DISABLE_SELECTIVE — cache can be deactivated for specific entities
  - UNSPECIFIED — use default settings of the PersistenceProvider

Demo

Query Cache

Query Cache

- Hibernate specific
- Stores query result session independent
- Needs to be activated (persistence.xml)
  - hibernate.cache.use_query_cache = true
- Activate caching for a specific query
  - org.hibernate.Query.setCacheable(true)
  - @NamedQuery(... hints = @QueryHint(name="org.hibernate.cacheable", value="true"))

Query Cache

- Stores query results for a query and its parameters
  - [„FROM Author WHERE id=?“, 1] → [1]
- Stores only entity references or scalars
  - Always use together with 2nd Level Cache

Demo

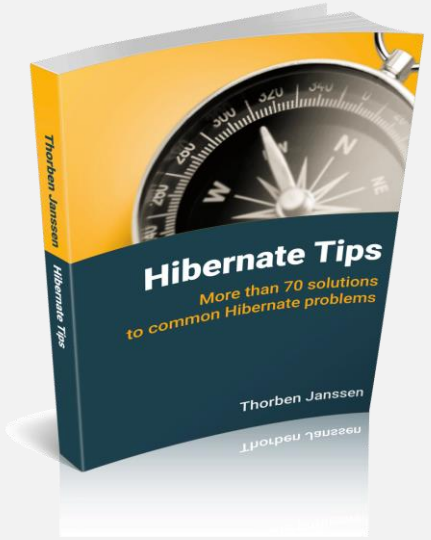
Recommendations

- Only cache data that is seldom updated
- Always benchmark the application when adding or changing caching
- Use Query Cache together with 2nd Level Cache
  - Configuration has to fit to each other

Tutorials, Online Courses and Workshops

www.thorben-janssen.com

More Hibernate Tips



Hibernate Tips
More than 70 solutions to common Hibernate problems

www.hibernate-tips.com

More resources

1. Deploy Java applications anywhere with Red Hat JBoss EAP
2. Red Hat JBoss Enterprise Application Platform technology overview
3. Data Grid for Development Use
4. Data grids meet data virtualization in modern data architectures

Download HERE

Developer Webinar Series 2020

#OpenShift #Microservices #Knative #Hibernate #Quarkus #Kafka #Cloudnative #Container

...will continue in July

…NEW @ redhat.com

...want to be notified about the next sessions? Send an email to Markus Eisele, Developer Adoption Lead, Red Hat, meisele@redhat.com

Red Hat