# Eager vs. Lazy Loading Strategies for JPA 2.1

Patrycja Wegrzynowicz

CTO, Yonita, Inc.

# About Me

- 15+ professional experience
  - Software engineer, architect, head of software R&D
- Author and speaker
  - JavaOne, Devoxx, JavaZone, TheServerSide Java Symposium, Jazoon, OOPSLA, ASE, others
- Finalizing PhD in Computer Science
- Founder and CTO of Yonita
  - Bridge the gap between the industry and the academia
  - Automated detection and refactoring of software defectsSecurity, performance, concurrency, databases
- @yonlabs

# Agenda

- My dear JPA ☺
- Loading strategies hints
- Corner cases
- Conclusions

I do love JPA!

# I do love JPA!

But as in every relationship we have our ups and downs.

YONITA

# My Dear JPA and Its Providers ☺

YONITA

# My Dear JPA and Its Providers ☺

# My Dear JPA and Its Providers ☺

# Hibernate JPA Provider
# Heads of Hydra

```
@Entity
public class Hydra {
        private Long id;
        private List<Head> heads = new ArrayList<Head>();

        @Id @GeneratedValue
        public Long getId() {...}
        protected void setId() {...}
        @OneToMany(cascade=CascadeType.ALL)
        public List<Head> getHeads() {
                return Collections.unmodifiableList(heads);
        }
        protected void setHeads() {...}
}

// new EntityManager and new transaction: creates and persists the hydra with 3 heads

// new EntityManager and new transaction
Hydra found = em.find(Hydra.class, hydra.getId());
```

# How Many Queries in 2nd Tx?

(a) 1 select
(b) 2 selects
(c) 1+3 selects
(d) 2 selects, 1 delete, 3 inserts
(e) None of the above

```
@Entity
public class Hydra {
        private Long id;
        private List<Head> heads = new ArrayList<Head>();

        @Id @GeneratedValue
        public Long getId() {...}
        protected void setId() {...}
        @OneToMany(cascade=CascadeType.ALL)
        public List<Head> getHeads() {
                return Collections.unmodifiableList(heads);
        }
        protected void setHeads() {...}
}

// new EntityManager and new transaction: creates and persists the hydra with 3 heads

// new EntityManager and new transaction
Hydra found = em.find(Hydra.class, hydra.getId());
```

# How Many Queries in 2nd Tx?

(a) 1 select
(b) 2 selects
(c) 1+3 selects
(d) 2 selects, 1 delete, 3 inserts
(e) None of the above

During commit hibernate checks whether the collection property is dirty (needs to be re-created) by comparing Java identities (object references).

# Another Look

```
@Entity
public class Hydra {
        private Long id;
        private List<Head> heads = new ArrayList<Head>();

        @Id @GeneratedValue
        public Long getId() {...}
        protected void setId() {...}
        @OneToMany(cascade=CascadeType.ALL)
        public List<Head> getHeads() {
                return Collections.unmodifiableList(heads);
        }
        protected void setHeads() {...}
}
// new EntityManager and new transaction: creates and persists the hydra with 3 heads

// new EntityManager and new transaction
// during find only 1 select (hydra)
Hydra found = em.find(Hydra.class, hydra.getId());
// during commit 1 select (heads),1 delete (heads),3 inserts (heads)
```

# Lessons Learned

- Expect unexpected ;-)
- Prefer field access mappings
- Operate on collection objects returned by hibernate
  - Don’t change collection references unless you know what you’re doing

# Lessons Learned

- Expect unexpected ;-)
- Prefer field access mappings
- Operate on collection objects returned by hibernate
  - Don’t change collection references unless you know what you’re doing

```
List<Head> newHeads = new List<>(hydra.getHeads());
Hydra.setHeads(newHeads);
```

# Other Providers?

- EcpliseLink
  - 1 select
- Datanucleus
  - 1 select
- „A Performance Comparison of JPA Providers”

# Lessons Learned

- A lot of depends on a JPA Provider!
- JPA is a spec
  - A great spec, but only a spec
  - It says what to implement, not how to implement
- You need to tune an application in a concrete environment

# Loading Strategy: EAGER for sure!

- We know what we want
  - Known range of required data in a future execution path
- We want a little
  - A relatively small entity, no need to divide it into tiny pieces

# Loading strategy: Usually Better EAGER!

- Network latency to a database
  - Lower number of round-trips to a database with EAGER loading

# Loading Strategy: LAZY for sure!

- We don’t know what we want
  - Load only required data
  - „I’ll think about that tomorrow”
- We want a lot
  - Divide and conquer
  - Load what’s needed in the first place

# Large Objects

- Lazy Property Fetching
- @Basic(fetch = FetchType.LAZY)
- Recommended usage
  - Blobs
  - Clobs
  - Formulas
- Remember about byte-code instrumentation,
  - Otherwise will not work
  - Silently ignores

# Large Objects

- Lazy Property Fetching
- @Basic(fetch = FetchType.LAZY)
- Recommended usage
  - Blobs
  - Clobs
  - Formulas
- Remember about byte-code instrumentation,
  - Otherwise will not work
  - Silently ignores

# Large Objects

- Something smells here
- Do you really need them?

# Large Objects

- Something smells here
- Do you really need them?
- But do you really need them?

# Large Objects

- Something smells here
- Do you really need them?
- But do you really need them?
- Ponder on your object model and use cases, otherwise it’s not gonna work

# Large Collections

- Divide and conquer!
- Definitely lazy
- You don’t want a really large collection in the memory
- Batch size
  - JPA Provider specific configuration

# Hibernate: Plant a Tree

```
@Entity
public class Forest {
    @Id @GeneratedValue
        private Long id;
        @OneToMany
        private Collection<Tree> trees = new HashSet<Tree>();

        public void plantTree(Tree tree) {
                return trees.add(tree);
        }
}

// new EntityManager and new transaction: creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree(“oak”);
em.persist(tree);
Forest forest = em.find(Forest.class, id);
forest.plantTree(tree);
```

# How Many Queries in 2nd Tx?

(a) 1 select, 2 inserts
(b) 2 selects, 2 inserts
(c) 2 selects, 1 delete, 10.000+2 inserts
(d) 2 selects, 10.000 deletes, 10.000+2 inserts
(e) Even more ;-)

```
@Entity
public class Forest {
    @Id @GeneratedValue
        private Long id;
        @OneToMany
        private Collection<Tree> trees = new HashSet<Tree>();

        public void plantTree(Tree tree) {
                return trees.add(tree);
        }
}

// new EntityManager and new transaction: creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree(“oak”);
em.persist(tree);
Forest forest = em.find(Forest.class, id);
forest.plantTree(tree);
```

# How Many Queries in 2nd Tx?

(a) 1 select, 2 inserts

(b) 2 selects, 2 inserts

(c)  2 selects, 1 delete, 10.000+2 inserts

(d) 2 selects, 10.000 deletes, 10.000+2 inserts

(e) Even more ;-)

The combination of **OneToMany** and **Collection** enables a bag semantic. That’s why the collection is re-created.

# Plant a Tree Revisited

STILL BAG SEMANTIC

Use OrderColumn or IndexColumn for list semantic.

```
@Entity
public class Orchard {
    @Id @GeneratedValue
        private Long id;
        @OneToMany
        private List<Tree> trees = new ArrayList<Tree>();

        public void plantTree(Tree tree) {
                return trees.add(tree);
        }
}

// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree(“apple tree”);
em.persist(tree);
Orchard orchard = em.find(Orchard.class, id);
orchard.plantTree(tree);
```

# Plant a Tree

```
@Entity
public class Forest {
    @Id @GeneratedValue
        private Long id;
        @OneToMany
        private Set<Tree> trees = new HashSet<Tree>();

        public void plantTree(Tree tree) {
                return trees.add(tree);
        }
}

// new EntityManager and new transaction: creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree(“oak”);
em.persist(tree);
Forest forest = em.find(Forest.class, id);
forest.plantTree(tree);
```

1. Collection elements loaded into memory
2. Possibly unnecessary queries
3. Transaction and locking schema problems: version, optimistic locking

# Plant a Tree

```
@Entity public class Forest {
   @Id @GeneratedValue
       private Long id;
       @OneToMany(mappedBy = „forest”)
       private Set<Tree> trees = new HashSet<Tree>();

       public void plantTree(Tree tree) {
              return trees.add(tree);
       }
}

@Entity public class Tree {
    @Id @GeneratedValue
       private Long id;
       private String name;
       @ManyToOne
       private Forest forest;

       public void  setForest(Forest forest) {
              this.forest  = forest;
              Forest.plantTree(this);
       }
}
```

Set semantic on the inverse side forces of loading all trees.

# Other Providers?

- EclipseLink
  - 2 selects/2 inserts
- OpenJPA
  - 3 selects/1 update/2inserts
- Datanucleus
  - 3 selects/1 update/2inserts

# Loading strategy: It depends!

- You know what you want
  - But it’s dynamic, depending on an execution path and its parameters

# Loading strategy: It depends!

- You know what you want
  - But it's dynamic, depending on runtime parameters
- That was the problem in JPA 2.0
  - Fetch queries
  - Provider specific extensions
  - Different mappings for different cases
- JPA 2.1 comes in handy

# Entity Graphs in JPA 2.1

- „A template that captures the paths and boundaries for an operation or query”
- Fetch plans for query or find operations
- Defined by annotations
- Created programmatically

# Entity Graphs in JPA 2.1

- Defined by annotations
  - @NamedEntityGraph, @NamedEntitySubgraph, @NamedAttributeNode
- Created programmatically
  - Interfaces EntityGraph, EntitySubgraph, AttributeNode

# Entity Graphs in Query or Find

- Default fetch graph
  - Transitive closure of all its attributes specified or defaulted as EAGER
- javax.persistence.fetchgraph
  - Attributes specified by attribute nodes are EAGER, others are LAZY
- javax.persistence.loadgraph
  - Attributes specified by by attribute nodes are EAGER, others as specified or defaulted

# Entity Graphs in Query or Find

- Default fetch graph
  - Transitive closure of all its attributes specified or defaulted as EAGER
- javax.persistence.fetchgraph
  - Attributes specified by attribute nodes are EAGER, others are LAZY
- javax.persistence.loadgraph
  - Attributes specified by by attribute nodes are EAGER, others as specified or defaulted

# Entity Graphs in Query or Find

- Default fetch graph
  - Transitive closure of all its attributes specified or defaulted as EAGER
- javax.persistence.fetchgraph
  - Attributes specified by attribute nodes are EAGER, others are LAZY
- javax.persistence.loadgraph
  - Attributes specified by by attribute nodes are EAGER, others as specified or defaulted

# Entity Graphs Advantages

- Better hints to JPA providers
- Hibernate now generates smarter queries
  - 1 select with joins on 3 tables
  - 1 round-trip to a database instead of default N+1
- Dynamic modification of a fetch plan

# There is that question...

YONITA

# Conclusions

- Keep your model neat
- Apply hints on loading strategies
  - Especially use JPA 2.1 Entity Graphs
- In case of perfomance problems
  - Tune in your concrete environment
  - JPA Providers behave differently!
  - Databases behave differently!

# Q&A

patrycja@yonita.com

@yonlabs

YONITA