Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1463–1469

Performance Antipatterns of One To Many Association in Hibernate

Patrycja Węgrzynowicz
Institute of Informatics
University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
Email: patrycja@mimuw.edu.pl

***Abstract*—Hibernate is the most popular ORM framework for Java. It is a straightforward and easy-to-use implementation of Java Persistence API. However, its simplicity of usage often becomes mischievous to developers and leads to serious performance issues in Hibernate-based applications. This paper presents five performance antipatterns related to the usage of one-to-many associations in Hibernate. These antipatterns focus on the problems of the owning side of collections, the Java types and annotations used in mappings, as well as processing of collections. Each antipattern consists of the description of a problem along with a sample code, negative performance consequences, and the recommended solution. Performance is analyzed in terms of the number and complexity of issued database statement. The code samples illustrate how the antipatterns decrease performance and how to implement the mappings to speed up the execution times.**

I. Introduction

HIBERNATE [1] is a remarkably popular implementation of Java Persistence API, i.e., the official standard of object-relational mapping in Java. However, many developers, especially of enterprise systems, complain on the correctness and the performance of Hibernate-based applications. It turns out that even relatively simple Hibernate applications impose a significant overhead on communication with a database, producing too many and/or inefficient SQL statements.

In this paper, we analyze the subject of the most common mapping, namely one-to-many associations of entities. Because of its wide usage as well as popularity of Hibernate, it seemed that the implementation of this association should have been well optimized. Our research revealed that even simple applications of one-to-many associations can result in (1) unexpected SQL statements being executed, (2) too many SQL statement being executed, and/or (3) too many objects being loaded into memory. Therefore, naïve mapping of one-to-many associations can introduce a significant performance overhead in a Hibernate-based application.

The main contribution of this paper consists of five performance antipatterns (i.e., bad practices with a significant impact on performance) related to the usage of one-to-many associations in Hibernate. Each antipattern consists of the description of a problem, performance consequences, the recommended solution, and a sample code.

II. Related Work

Antipatterns are conceptually similar to design patterns as they describe recurring problems and provide solutions to them. A performance antipattern describes a bad practice that has a significant impact on performance. The articles [2] and [3] provide a good explanation of performance antipatterns along with the definition of 14 performance antipatterns. These papers have formed a good basis for further research on performance antipatterns, mainly in the field of automated detection and fixing of performance problems. The paper [4] introduces a framework for automated detection and assessment of performance antipatterns in component based systems. The authors of [5] focus on detection of performance antipatterns in architectural models. The article [6] explains the method to remove performance antipatterns from software by analyzing UML models. The paper [7] describes modelling and analysis of software performance antipatterns along with their constraints and solvability using different models.

As mentioned above there has been work done in the field of performance antipatterns, defining solid means in terms of their definitions, detection, and fixing. These efforts focus on generic and domain-independent antipatterns, mostly in the application layer. However, they do not touch the data layer. Numerous performance problems have their source in an inefficient way of retrieving or storing large amount of data in a database. Therefore, this area is important to correctly identify the source of performance issues in software. It is especially true nowadays, when the amount of data processed continually increases and the use of automated frameworks for object-relational mapping becomes more and more common.

There is a number of popular ORM libraries, like Hibernate [1], EclipseLink [8], Open JPA [9], and Data Nucleus [10] to name a few. Even though they implement the same specification — Java Persistence API, they differ in mapping policies, generate different schemata and SQL statements that have different performance characteristics. The article [11] analyzes the influence of optimizations on the performance of Hibernate. In another paper [12], the same authors compare the performance of an object-relational mapping tool (Hibernate) vs. an object-oriented database (db4o) using OO7 benchmark. Another comparative study of the performance of object-relational mapping tools for .NET platform can be found

978-1-4673-4471-5/$25.00

in [13]. However, these comparative studies focus mainly on queries, which usually are as efficient as SQL queries since they are direct translation to SQL. Some authors (e.g., [14]) identify the need to improve the efficiency of ORM tools by utilizing the features provided by the database engines.

Even though there has been work done on comparing performance of ORM tools, we still lack a systematic approach to identification of their strengths and weaknesses in terms of impedance mismatch [15]. In this paper, we aim at the identification of common performance problems for one-to-many associations in Hibernate. We also define five new antipatterns and provide recommendations how to fix them.

III. Antipattern: Inadequate Collection Type on Owning Side

A. Description

In JPA, *@OneToMany* annotation used on the owning side (i.e., the side used to manage persistency of elements) is one of the most common implementations of a one-to-many association between persistent entities. Such an approach is commonly used in enterprise development mainly due its simplicity and little coding overhead. However, it can introduce a serious performance overhead in Hibernate when combined with an inadequate Java collection type.

Table I presents three types of semantics available in Hibernate, dependent on the combination of a Java collection type and JPA annotations. Each of these semantics has a different performance characteristic. Table II shows the numbers of statements issued while persisting a collection of a given semantics after an addition or removal of a single element.

The bag semantics has the worst performance when it comes to the number of operations since it always re-creates the entire collection. Hibernate issues a delete statement to remove all associations of the old collection from the association table. Then, it issues N inserts to add all associations representing the new collection to the association table. Hibernate does not analyze how many elements have been changed in the collection.

In the list semantics, an addition or removal of a single element from a collection of N elements results in a single insert or delete respectively and M updates. The updates are needed to correct the indices of M elements. In case of addition, Hibernate needs to correct the indices of the elements before the one being added. In case of removal, Hibernate needs to correct the indices of the elements after the one being removed.

The set semantics seems to be the most efficient. For a single operation on a collection, it requires only a single database operation. However, it is worthwhile remembering that the Java set semantics requires a uniqueness check on the elements of a set. It implies that, in case of any additions of new elements to a persistent set, all elements of the set must be loaded into the main memory.

The antipattern relates to the usage of an inefficient collection semantics for a given usage pattern of a collection:

- For usage patterns of collections where in a single transaction the collection is usually left unmodified or only a few elements are added or removed, the usage of the bag semantics (i.e., *java.util.Collection* or *java.util.List* without the index or order annotations) is notably inefficient.
- For usage patterns of collections where in a single transaction most of the elements of the collection are removed, the usage of the set or list semantics (i.e., *java.util.Set* or *java.util.List* with the index or order annotation) is inefficient.

TABLE I
THREE SEMANTICS FOR COLLECTIONS WITH @ONETOMANY ANNOTATION IN HIBERNATE.

Semantics	Java Type	Annotation
Bag semantics	java.util.Collection java.util.List	@OneToMany
List semantics	java.util.List	@OneToMany $\wedge$ (@IndexColumn $\vee$ @OrderColumn)
Set semantics	java.util.Set	@OneToMany

TABLE II
THE NUMBERS OF DML STATEMENTS ISSUED WHILE PERSISTING A COLLECTION OF N ELEMENTS WITH A GIVEN SEMANTICS (DEFAULT TABLE MAPPING WITH AN ASSOCIATION TABLE; NO CASCADE OPTION).

Semantics	One Element Added	One Element Removed
Bag semantic	1 delete, N inserts	1 delete, N inserts
List semantic	1 insert, M updates	1 insert, M updates
Set semantic	1 insert	1 delete

B. Consequences

The performance consequences of the usage of an inadequate collection type on the owning side of a one-to-many association include an increased workload on the database engine because:

- For the bag semantics, Hibernate re-creates an entire collection, performing one delete to clear the collection and as many inserts as there are elements in the collection. For the common usage pattern of collections, where only a few elements are added or removed, such a re-creation results in suboptimal performance due to the operations on data which actually have not been changed. The bigger the collection is, the performance overhead is more significant.
- For the list or set semantics, Hibernate performs single delete or insert per each removal or addition respectively. For the usage pattern of collections, where most elements of a collection are removed, such a strategy results in suboptimal performance due to many deletes instead of one delete to clear the collection in one operation.

C. Solution

The solution to this antipattern is to analyze the usage profile of collections in an application and adjust the type of Java collections accordingly:

- If a collection is constant or is the subject to minimal changes or is relatively small, the recommended collection type is *java.util.Set* with the set semantics.
- If a collection is heavily modified, the recommended collection type is *java.util.Collection* or *java.util.List* with the bag semantics.

D. Sample Code

Listing 1 presents an example of the antipattern related to an inadequate collection type on the owning side. The sample code has two persistent entities (`Forest` and `Tree`), which are connected by an unidirectional association (`Forest` has a collection of `Tree` objects). The classes meet all requirements imposed by JPA and Hibernate on persistent entities (e.g., no-arg constructor). We use a minimal set of additional annotation and configuration parameters, instead relying on default values.

To test the persistency mechanism implemented in Hibernate, we execute two transactions. To ensure the same runtime environment (e.g., clear caches), each transaction is executed with a new `EntityManager` instance. The first transaction creates a `Forest` instance and 10000 `Tree` instances planted in the newly created `Forest`, whereas the second transaction finds the previously created `Forest` instance, creates a new `Tree` instance and plants it in the found `Forest`. It turns out that Hibernate for such a piece of code as in the second transaction re-creates the entire collection (i.e., executes one delete and 10 001 inserts). This behavior of Hibernate is not performance-wise since in our example one insert would be enough to synchronize the state of the object in the main memory with the state of the records in the database. For large collections with only a few changes it imposes a significant performance overhead.

Listing 1. The example of an inadequate collection type on the owning side.

```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany
    Collection<Tree> trees =
        new HashSet<Tree>();
    ...
    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}
@Entity
public class Tree {
    @Id @GeneratedValue
    private Long id;
    private String name;
    ...
}
// Transaction 1
// creates and persists a forest...
// ... with 10.000 trees
...
// Transaction 2
Tree tree = new Tree("oak");
em.persist(tree);
Forest forest = em.find(Forest.class, id);
forest.plantTree(tree);
```

IV. Antipattern: OneToMany as Owning Side

A. Description

The antipattern relates to the usage of the collection side (i.e., *@OneToMany*) as the owning side of an association for large collections, especially the ones which expect only a few changes in a single transaction.

According to Section III, for such a use case we should use *java.util.Set* to minimize the performance overhead. However, even usage of *java.util.Set* does not guarantee the optimal performance.

First, due to the Java set semantics the entire collection needs to be loaded into the main memory in order to enforce a uniqueness check in case of addition of elements to the collection. It results in an additional database query (or queries due to batch loading) issued and additional processing dedicated to the transformation of each returned row into an object. These additional queries and processing happen even if the application logic does not access the elements of the collection in question.

Second, there might be an overhead connected with transactions and locking, when the entity containing the collection in question uses optimistic locking with versioning. In such cases, Hibernate locks not only the entity but also the collection, which lowers the capability of an application to serve concurrent requests.

B. Consequences

The performance consequences of the *@OneToMany* as the owning side antipattern are as follows:

- There is an increased workload on the database engine. Hibernate issues an additional database query (or a number of queries in case of batch loading) to retrieve the elements of a collection.
- There is an increased workload on the application server (CPU). Hibernate needs to convert each returned row into an object, even if the application logic does not access those objects.
- The memory footprint is significant, especially for large collections. It can result in slower performance due to insufficient memory available, leading to more frequent garbage collections or even page swaps.

- There may be a decreased throughput. Due to transaction and locking issues, the capability of an application to serve concurrent requests may be significantly lowered.

C. Solution

The solution to this antipattern is to manage a collection from the *@ManyToOne* side instead of *@OneToMany* side, i.e., to make *@ManyToOne* the owning side of the association. In case we are not able to change the owning side of an association, we should at least exclude such a collection from locking by using a Hibernate-specific annotation: *@OptimisticLock(excluded=true)*.

D. Sample Code

Listing 2 presents an example of *@OneToMany* as the owning side antipattern. It mimics Listing 1, introducing only a few changes: (1) the Java type of a collection has been changed to *java.util.Set* and (2) version fields have been added in `Forest` and `Tree` classes. The key piece of code is located in the second transaction, which adds a single `Tree` to the previously created `Forest` with 10 000 trees. It turns out that in the second transaction, the entire forest is loaded into memory. Moreover, we are not able to plant trees in parallel because Hibernate locks the entire `Forest` instance along with all its `Tree` associations.

V. Antipattern: Inadequate Collection Type on Inverse Side

A. Description

This antipattern relates to the usage of *java.util.Set* on the inverse side (also known as the mapped collection) of a bidirectional one-to-many association in Hibernate.

In case of bidirectional associations, it is recommended to synchronize the state of the objects (i.e., the mapped collection and the element entities with *@ManyToOne* pointers) in memory. The common implementation to ensure the consistence of the objects relies on an automated update of the mapped collection by the setter method in an element entity responsible for setting the *@ManyToOne* pointer (see Section V-D).

Therefore, when the type of the mapped collection is *java.util.Set*, it means that the entire collections needs to be loaded into the main memory in response to each addition of new elements, even though the application does not access neither the collection nor its elements. It happens due to the Java set semantics mentioned earlier, which requires a uniqueness check on the elements of a set.

B. Consequences

The performance consequences of the usage of *java.util.Set* on the inverse side of a bidirectional one-to-many association with in-memory state synchronization are as follows:

- There is an increased workload on the database engine. Hibernate issues an additional database query (or a number of queries in case of batch loading) to retrieve the elements of a collection.

Listing 2. The example of @OneToMany as the owning side.

```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @Version
    private Integer version;
    @OneToMany
    Set<Tree> trees = new HashSet<Tree>();
    ...
    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}
@Entity
public class Tree {
    @Id @GeneratedValue
    private Long id;
    @Version
    private Integer version;
    private String name;
    ...
}
// Transaction 1
// creates and persists a forest...
// ... with 10.000 trees
...
// Transaction 2
Tree tree = new Tree("oak");
em.persist(tree);
Forest forest = em.find(Forest.class, id);
forest.plantTree(tree);
```

- There is an increased workload on the application server (CPU). Hibernate needs to convert each returned row into an object, even if the application logic does not access those objects.
- For large collections, there is a significant memory footprint. It can slow down performance of an application due to insufficient memory available, more frequent garbage collections, or even page swaps.

C. Solution

The recommended Java types to be used on the inverse side of a one o many association are *java.util.Collection* or *java.util.List*. These types do not force loading the elements into memory until the elements are accessed by client code.

D. Sample Code

Listing 2 presents an example of *java.util.Set* used on the *@OneToMany* inverse side with in-memory state synchronization. The example continues the previously described `Forest` and `Tree` classes presented in Sections III and IV. Here, `Tree` is the owning side of an association, while `Forest`

Listing 3. The example of an inadequate collection type on the inverse side.

```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany (mappedBy = "forest")
    Set<Tree> trees = new HashSet<Tree>();
    ...
    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}
@Entity
public class Tree {
    @Id @GeneratedValue
    private Long id;
    private String name;
    @ManyToOne
    Forest forest;
    ...
    public void setForest(Forest forest) {
        this.forest = forest;
        this.forest.plantTree(this);
    }
}
// Transaction 1
// creates and persists a forest...
// ... with 10.000 trees
...
// Transaction 2
Tree tree = new Tree("oak");
Forest forest = em.find(Forest.class, id);
tree.setForest(forest);
em.persist(tree);
```

is the inverse side. Therefore, to save changes in a database, we need to set a right `Forest` reference in a `Tree` instance. While setting the `Forest` instance in a `Tree`, the collection of `Trees` is automatically updated to include the new tree (the invocation of `plantTree`). Such a pattern is widely used and recommended to ensure up-to-date states of objects. However, it results in a significant performance overhead, when combined with *java.util.Set* on the inverse side. In the second transaction, which only adds a new tree and does not access other trees in the forest, Hibernate has to load the entire collection into the main memory.

VI. Antipattern: Lost Collection Proxy on Owning Side

A. Description

The antipattern relates to the assignment of a new collection object to a persistent field, representing the owning side of a one-to-many association. Thus, a collection proxy returned by Hibernate is lost.

In such a case, Hibernate is not able to track what has been changed in a collection and its policy is to re-create the entire collection regardless of the actual modifications. Therefore, even if the elements of the collection have not been changed, Hibernate issues a delete to remove the associations from the association table and then performs as many inserts as there are elements in the collection.

B. Consequences

The performance consequences of a lost proxy on the owning side are as follows:

- There is an increased workload on the database engine. There are unnecessary database operations performed. Hibernate re-creates an entire collection, performing one delete to clear the collection and as many inserts as there are elements in the collection. Such a re-creation results in suboptimal performance due to the operations on data which actually have not been changed.

C. Solution

The recommended solution to the antipattern is to operate on collection objects returned by Hibernate as in most cases it is a much more efficient approach. However, it might be performance-wise to re-create the entire collection in cases where most elements of the collection have been removed. On the other hand, it is one of the places where Hibernate could apply a smarter policy, especially as it has all required data available.

D. Sample Code

Listing 4 presents an example of a lost (according to Hibernate) collection proxy on the owning side. The example consists of two persistent entities: `Hydra` and `Head`. `Hydra` is a mythical creature that re-grows three heads in place of one head cut off. In order to model this feature, we need to provide strict encapsulation of heads. Therefore, `getHeads` returns an unmodifiable wrapper over the mutable collection of heads. In the first transaction, we create and persist a `Hydra` instance with three `Heads`. In the second transaction, we simply read the previously stored instance. It turns out that for this piece of code Hibernate executes two selects, one delete and three inserts, even though the code is purely read-only. The problem lies in the way Hibernate checks whether or not a property is dirty during the commit of a transaction. In order to check the dirtiness of a collection, Hibernate compares the references on the actual collection and the proxy originally loaded. Unfortunately, in our example Hibernate uses `getHeads` method to access the collection (due to property access mapping). The method returns an unmodifiable wrapper over the original proxy returned by Hibernate. Obviously, it returns a different Java object that the original one loaded by Hibernate. Thus, Hibernate decides that the collection has been changed and re-creates the entire collection.

Listing 5 presents a more straightforward example of a lost collection proxy on the owning side. Again we implement two

Listing 4. The example 1 of a lost collection proxy on the owning side.

```
@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(List<Head> heads)
  {...}
}

// Transaction 1
// creates and persists the hydra...
// ...with 3 heads
...
// Transaction 2
Hydra found = em.find(Hydra.class, id);
```

Listing 5. The example 2 of a lost collection proxy on the owning side.

```
@Entity
public class Hydra {
  @Id @GeneratedValue
  private Long id;
  @OneToMany(cascade=CascadeType.ALL)
  private List<Head> heads =
    new ArrayList<Head>();
  ...
  public Long getId() {...}
  protected void setId() {...}
  public List<Head> getHeads() {
      return heads;
  }
  public void setHeads(List<Head> heads) {
    this.heads = heads;
  }
}

// Transaction 1
// creates and persists the hydra...
// ...with 3 heads
...
// Transaction 2
Hydra found = em.find(Hydra.class, id);
List<Head> currentHeads =
  new ArrayList<Head>(found.getHeads());
found.setHeads(currentHeads);
```

persistent classes: `Hydra` and `Head`. However, we do not introduce strict encapsulation. Instead we implement simple getters and setters for all fields. In the second transaction, we create a new collection containing the current heads of our `Hydra` instance. In terms of our business model, nothing has changed — the heads are the same heads as originally loaded. However, Hibernate observes a different collection reference and applies the policy of re-creation of the entire collection.

VII. Antipattern: One-By-One Processing of Collection

A. Description

The antipattern refers to a sequential processing of a persistent collection, i.e., a piece of code iterates over the collection and for each element in a collection, it may perform a database operation.

Best practices of database programming highlight the need to operate on the sets of records instead of single records. While SQL provides the means to such a paradigm switch in programming, Java is an object-oriented language without support to relational algebra. Therefore, it is a common approach in the Java world to iterate over persistent collections and process their elements one-by-one. Such an approach often leads to a significant performance overhead, considering the number of database round-trips and the volume of data passed between a database and an application.

B. Consequences

The performance consequences of one-by-one processing of a persistent collection are as follows:

- A high number of database operations is executed. It is proportional to the size of the collection.
- RDBMS engine is used ineffectively.
- Network latency can sum up to a significant performance overhead.

C. Solution

The solution to this antipattern is to utilize the capabilities of a relational database by the usage of bulk statements and aggregate functions. Frequently it requires a different object model.

D. Sample Code

Listing 6 presents an example of a one-by-one processing of a collection. The example continues the previous examples consisting of `Forest` and `Tree`. Here, in the second transaction we want to delete the entire `Forest`. A simple remove causes a constraint violation exception since there are trees associated with the forest to be removed. Therefore, we need to unbind the trees first. Unfortunately, in Hibernate there is no other way to do this than setting the `Forest` reference in each `Tree` instance to `null`. In our example it results in

Listing 6. The example of a one by one processing of a collection.

```
@Entity
public class Forest {
  @Id @GeneratedValue
  private Long id;
  @OneToMany (mappedBy = "forest")
  Set<Tree> trees = new HashSet<Tree>();
  ...
  public void plantTree(Tree tree) {
    trees.add(tree);
  }
}
@Entity
public class Tree {
  @Id @GeneratedValue
  private Long id;
  private String name;
  @ManyToOne
  Forest forest;
  ...
}
// Transaction 1
// creates and persists a forest...
// ... with 10.000 trees
...
// Transaction 2
Tree tree = new Tree("oak");
Forest forest = em.find(Forest.class, id);
for (Tree tree : forest.getTrees()) {
  tree.setForest(null);
}
em.remove(forest);
```

10 000 updates. To fix this inefficiency in Hibernate, we need to change the object model and introduce an explicit class representing the association.

VIII. Conclusion

In this paper, we presented five performance antipatterns related to one-to-many associations in Hibernate. Each antipattern consists of the description of a problem, performance consequences and the recommended solution, as well as a sample code to better illustrate the problem. The identified antipatterns introduce a significant performance overhead in terms of the number of SQL statements executed as well as the number of objects loaded into the main memory. These are two critical factors that have serious impact on the performance of applications. The number of SQL statements executed directly increases the load on the database engine. Usually it also introduces an additional performance overhead due to the network latency which is important in modern multi-tiered applications, where applications and databases are located in different servers/tiers. High memory consumption has indirect impact on the performance as it usually leads to more frequent garbage collection or even page swaps.

The presented antipatterns show that the usage of Hibernate is not as simple as it looks at first glance. Even plain use cases can significantly decrease the performance of an application. The antipatterns explain how to use Hibernate efficiently and what policies should be improved in Hibernate in order to shorten the execution time.

References

[1] Hibernate. [Online]. Available: http://www.hibernate.org

[2] C. U. Smith and L. G. Williams, "Software performance antipatterns; common performance problems and their solutions," in *Int. CMG Conference*, 2001, pp. 797–806.

[3] ——, "New software performance antipatterns: More ways to shoot yourself in the foot," in *Int. CMG Conference*. Computer Measurement Group, 2002, pp. 667–674.

[4] T. Parsons and J. Murphy, "A framework for automatically detecting and assessing performance antipatterns in component based systems using run-time analysis," in *The 9th International Workshop on Component Oriented Programming, part of ECOOP*, 2004.

[5] C. Trubiani and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, ser. ICPE '11. New York, NY, USA: ACM, 2011, pp. 19–30. [Online]. Available: http://doi.acm.org/10.1145/1958746.1958755

[6] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, and C. Trubiani, "Digging into uml models to remove performance antipatterns," in *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, ser. QUOVADIS '10. New York, NY, USA: ACM, 2010, pp. 9–16. [Online]. Available: http://doi.acm.org/10.1145/1808877.1808880

[7] V. Cortellessa, A. Di Marco, and C. Trubiani, "Software performance antipatterns: modeling and analysis," in *Proceedings of the 12th international conference on Formal Methods for the Design of Computer, Communication, and Software Systems: formal methods for model-driven engineering*, ser. SFM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 290–335. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30982-3_9

[8] Eclipselink. [Online]. Available: http://www.eclipse.org/eclipselink/

[9] Openjpa. [Online]. Available: http://openjpa.apache.org/

[10] Datanucleus. [Online]. Available: http://www.datanucleus.org/

[11] P. van Zyl, D. G. Kourie, L. Coetzee, and A. Boake, "The influence of optimisations on the performance of an object relational mapping tool," in *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, ser. SAICSIT '09. New York, NY, USA: ACM, 2009, pp. 150–159. [Online]. Available: http://doi.acm.org/10.1145/1632149.1632169

[12] P. van Zyl, D. G. Kourie, and A. Boake, "Comparing the performance of object databases and orm tools," in *Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, ser. SAICSIT '06. Republic of South Africa: South African Institute for Computer Scientists and Information Technologists, 2006, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1145/1216262.1216263

[13] S. Cvetković and D. Janković, "A comparative study of the features and performance of orm tools in a .net environment," in *Proceedings of the Third international conference on Objects and databases*, ser. ICOODB'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 147–158. [Online]. Available: http://dl.acm.org/citation.cfm?id=1926241.1926257

[14] A. Szumowska, M. Burza$#324;ska, P. Wi$#347;niewski, and K. Stencel, "Efficient implementation of recursive queries in major object relational mapping systems," in *Proceedings of the Third international conference on Future Generation Information Technology*, ser. FGIT'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 78–89. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27142-7_10

[15] P. Wiśniewski, M. Burzańska, and K. Stencel, "The impedance mismatch in light of the unified state model," *Fundam. Inform.*, vol. 120, no. 3-4, pp. 359–374, 2012.