A Survey of Object-Relational Transformation Patterns for High-performance UML-based Applications

Nemanja Kojić and Dragan Milićev
Faculty of Electrical Engineering, University of Belgrade, Bulevar Kralja Aleksandra 73, Belgrade, Serbia

Keywords: Object-relational Mapping, Relational Databases, Denormalization, UML.

Abstract: We outline a methodology for automatic and efficient object-relational mapping (ORM) in the context of model-driven development (MDD) of high-performance information systems specified with executable UML models. Although there are various approaches to performance tuning, we focus here on the persistence layer−the relational database. The relational data model is usually designed following the well-known normal forms. However, a fully normalized relational model often does not provide sufficient performance, and improper relational model design can easily lead to a slow and unusable relational database for particular operations. Our ORM approach is intended to exploit smart optimization techniques from the relational paradigm that abandon normalization and its positive effects, and trade them off for better performance. Our ORM approach hence combines the classical denormalization transformations, based on reducing or eliminating expensive database operations by the model restructuring, but applies them to a non-redundant conceptual UML model. In this paper, we also present the first step towards this goal: a catalogue of ORM transformation patterns.

1 INTRODUCTION

There are two broad classes of information systems: transactional (OLTP) and analytical (OLAP). OLTP information systems, apart from storing live and active data, are characterized by intensive short online transactions, fast query processing, and maintaining strong data consistency in a concurrent environment. Data in OLTP information systems are usually persisted in OLTP relational databases, since they are mature and reliable persistence technology. Performance optimization and efficient handling of data is tightly coupled with the data model in the relational database. In our context, a special UML profile, customized for information systems modeling, is used for capturing key data and operations. In the MDD approach, the data model (DDL schema) is automatically generated from the UML model by object-relational mapping (ORM).

In addition to (statically) generating a relational database schema, the runtime component of ORM has to provide operations of (dynamically) persisting data in a relational database during transaction processing. Conceptual UML models, that are the input into this process, are usually normalized, regarding the data aspect, which means there is no redundancy. The normal forms, in general, minimize effort for ensuring strong data consistency (Codd, 1971; Maier, 1983). Regarding ORM approaches, UML models are often in practice transformed to normalized relational data models. However, we have witnessed that a fully normalized relational data model cannot provide desired scalability and performance of a large-scale information system with intensive transactional processing. In addition, numerous researchers in the domain of relational databases argue that in practice, a relational data model must be denormalized to fit in a form that is handled most efficiently by a relational database. They also provide numerous denormalization techniques that increase performance of queries and reduce or even eliminate expensive database operations (Shin and Sanders, 2006; Sanders and Shin, 2001; Keller and Coldewey, 1997). These techniques have been traditionally associated with OLAP systems, which assume none or very little updates, and complex and intensive retrieval operations on high volumes of data. On the other hand, OLTP systems may still benefit from the denormalization techniques, although the penalty expressed through increased volume of update operations for the sake of consistency of redundant

DOI: 10.5220/0005242302800285
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 280-285
ISBN: 978-989-758-083-3

data has been traditionally considered as a detractor from applying these techniques to OLTP systems.

The paper is structured as follows. Section 2 describes the motivation for enhancing ORM to exploit database features in a more efficient way, according to the knowledge of experts in that area and proven optimization techniques. In section 3, we give a catalog of ORM transformations for mapping object-oriented models to optimized relational data models. Section 4 gives the main conclusions and addresses directions for future work.

2 MOTIVATION

Accessing a relational database in a most efficient way and maximizing usage of its most efficient features is the key approach for achieving good performance. Efficient access to a relational database is tightly related to the complexity of queries. The complexity of queries is directly dictated by the relational model itself. The more normalized model is, the more complex queries are in general, because of joins, repeating calculations of derived data, etc. In the relational database theory, it is well known that the normal forms are often considered and widely adopted as a principle of a good database design that promotes elimination of data redundancy, while minimizing effort of maintaining data consistency (Agarwal, Keene and Keller, 1995).

In the context of an OO information system that persists data in an OLTP relational database, the ORM approach has to be sophisticated enough to automatically create a relational model that is most efficient and optimized (even denormalized) for the particular database. Each denormalization technique brings both advantages and disadvantages. The choice of an appropriate denormalization technique highly depends on the nature of the system and data access patterns. For example, if a data value is derived (computed) from other data values that are very infrequently modified, or not modified at all after initialization, it will be an excellent candidate for storing as a redundant persistent value at all places (relational tables) where it is retrieved from with other data. Also, constraints in the logical UML models can sometimes limit the number of available denormalization options. Unlike the other denormalization approaches, our approach differs in one important detail. While the denormalization is considered as a process of restructuring an existing normalized relational data model (Shin and Sanders, 2006; Sanders and Shin, 2001; Keller and Coldewey, 1997), our approach exploits the knowledge from the denormalization techniques and applies them directly to create an initially denormalized relational model from a (non-redundant) conceptual UML model in the context of OLTP information systems.

Denormalization is a complex process in practice, done (or at least instructed and steered) exclusively by human experts, who are the only ones who understand the semantics of applications and the static and dynamic nature of the structural and behavioral model of the system. Static aspects of the model are based on recognition of structural class patterns that are good candidates for mapping to denormalized relations. If the class patterns are not isolated from other classes in the model, the static rules are not always applicable easily, since there are many combinations to examine while the model is being compiled to the relational model. For example, if a complex application does not contain any code for actions that access a structure of related data in a particular manner, it is of no use to optimize the piece of relational model for that particular access. This task is error-prone if done manually, without a systematic approach. That is one of the reasons why we investigate a fully automatic ORM approach, that should consider all available denormalization options in a systematic way. In spite of all positive effects of denormalization, it is worth repeating that it makes updates more complex (although they are done automatically by the ORM runtime, they still put additional workload to the database). It must be carried out in a controlled manner, balanced carefully between the achieved performance and the relational model maintainability. Uncontrolled denormalization can lead to an even more complicated relational model, derived as a result of the denormalization explosion. This can be solved by considering the dynamic aspects of the system's model, by online data access profiling and discovering the most dominant operations in the system. A hybrid combination of the static and dynamic aspects of the model would lead to a more scalable and efficient ORM approach that controls the denormalization explosion by focusing on the most frequently accessed data. All of these optimizations are hidden and transparent for the developers, since ORM is responsible for creating the appropriate relational model and mapping application's operations to the optimized relational data model. Finally, what we find extremely necessary regarding efficient application of denormalization is a comprehensive quantitative analysis of denormalization techniques that should result in a guide that will provide for each denormalization technique a context in which it is most effective.

3 ORM TRANSFORMATION PATTERNS

As the first prerequisite for the research path, we have described in the previous section, we establish and describe a catalogue and classification of ORM transformations in the described context.

There are four classes of denormalization strategies: (1) collapsing relations, (2) partitioning relations, (3) adding redundant properties, and (4) adding derived properties (Shin and Sanders, 2006).

In this chapter, we present transformations of generalization/specialization relationships and associations (aggregations and compositions, as special kinds of associations in UML, are not covered separately), relying on the denormalization techniques from the relational paradigm. We also outline some optimization techniques that are not directly related to the structure of the relational model, but rather represent optimization tricks.

3.1 Object Identifier

Object identifier generation should not be centralized in the relational database, as it may impose unnecessary database load. It should be rather decentralized and stateless. One way to accomplish this requirement is to generate an object identifier as a GUID. Yet another important aspect of object identifier is that may carry the object type identifier. That way, it is possible to dynamically infer the type of an object without querying the database, which leverages scalability of the persistence layer and makes the polymorphic queries more efficient. In addition, having the object type identifier encoded in object identifier, eliminates high load of tables near to the root inheritance table (Keller and Coldewey, 1997; Keller and Coldewey, 1998).

3.2 Mapping Inheritance

We do not consider multiple inheritance, but only single class inheritance. The authors in (Keler, 1995; Agarwal, Keene and Keller, 1995) presented a few relational model transformations for (efficient) mapping of inheritance. In this chapter we combine the existing denormalization approaches with requirements of ORM.

One Table per Class: Among probably many other places, this approach was presented in (Keller, 1997; Agarwal, Keene and Keller, 1995), as a vertical partitioning. The main idea of the approach is to map each class in the model to one table in the relational database. Abstract classes also have their own tables in the relational model. The tables in the database form a tree, with one root table that holds object identifiers. Records in each child table are linked to the corresponding records in the parent table with the object identifier as the foreign key. The advantage of such an approach is getting an easy-to-maintain and normalized relational model, optimized for updates, but not for reads. Queries that generalize objects (e.g., a query that searches for all instances of a base class, possibly abstract, that satisfy a criterion over properties of that base class) are straightforward and efficient. Other queries that fetch both inherited and specific properties of derived classes may be far from simple and efficient. The root table, and tables near to the root table, are thus often under heavy load of queries, because of frequent joins, which may affect the scalability of the system. The approach may not be usable in case of deep inheritance hierarchies, since multi-way joins are required for retrieving basic object information (Agarwal, Keene and Keller, 1995).

One Table per each Concrete Class: Usually, there is no need to create separate tables for abstract classes, but all their properties are copied to the tables of the inherited classes. The properties from the abstract class are thus duplicated in the schema, or repeated in each table that corresponds to a concrete inherited classes (note, however, that values are not duplicated, unless an object belongs to more than one derived class). The rule may be generalized to a sub-hierarchy of abstract classes related with generalization/specialization. Eliminating tables for abstract classes may improve reading performance, as some joins are eliminated.

One Table per One Inheritance Path: This approach is useful in situations when the previous two cannot provide sufficient reading performance, due to the mentioned heavy load of the root tables and multi-way join operations. This approach is characterized with producing one table per each inheritance path (assuming single inheritance). An inheritance path starts from the root class and ends at each concrete class, no matter if it is the leaf or not (abstract classes are not considered). All inherited properties on the inheritance path are collected into the table for that path. This approach introduces even more redundancy in the relational model (but still not on the data), but eliminates joins for retrieving basic object information that are already present in the table. As a consequence, the elimination of multi-way joins completely eliminates heavy load of the root tables. Although the bottleneck near the root table is eliminated, the relational model now complicates generalized polymorphic queries, since more tables must be combined/joined to retrieve the desired information (Keller, 1997).

One Table for One Inheritance Tree: This approach assumes mapping of a whole inheritance tree into one single table. This is also named as the typed partitioning, as mentioned in (Agarwal, Keene and Keller, 1995). The records in the table unify all the properties from all classes in the inheritance hierarchy, which eliminates expensive join operations and optimizes polymorphic queries, but creates a highly denormalized relational model. This approach may not show good results in case of deep hierarchies, since the table gets too big and cumbersome. Since all data are stored in only one table, the problem of bottleneck arises again, along with a large waste of storage, because of a lot of null values. Hence, this approach is recommended only in case of shallow inheritance hierarchies and low concurrency (Keller, 1997).

One Table per each Concrete Class with Controlled Redundancy of Properties: We propose this hybrid approach that leverages advantages of the presented approaches and refines the "one table per one concrete class" approach, by copying properties from a base class table to the tables of inherited classes, in order to speed up generalized queries. However, instead of copying all properties from a base class table to the tables of inherited classes, only those base class properties that are most often retrieved in combination with the inherited class properties in the queries, may be replicated. In particular, values of redundant properties are copied in several tables (for base class to which the property belongs and for derived classes for optimized retrieval). This way, the degree of denormalization is smaller than in the "one table for inheritance path" approach, but performance of reads is optimized. In addition, in the “one table for inheritance path” approach, producing records with great number of columns may also have some negative effects on performance of read queries. For example, if the database cannot store one record in one physical page, then the number of accessed pages may be increased. Hence, this selective copying of properties from the tables of base classes to the tables of inherited classes controls the explosion of columns in records and keeps the physical model under control. This approach must be supported by the online data access profiling. To the best of our knowledge, this approach has not been published or systematically implemented in an automated ORM except in our SOLoist (www.soloist4uml.com) framework for model-driven development (Milicev, 2009).

3.3 Mapping Associations

Efficient mapping of associations is another challenge for a sophisticated ORM. The multiplicity constraint on association ends is usually the main factor that influences the selection of a proper mapping transformation. In this section, we do not consider aggregations and compositions separately, since all that works for associations, works also for aggregations and compositions (composition has implications on the semantics of actions that are not relevant for our discussion).

There are three well known approaches for mapping associations, with respect to the multiplicity constraint: (1) distinct table approach, (2) embedded foreign key approach, and (3) embedded class approach (Agarwal, Keene and Keller, 1995).

It is necessary to mention that the transformations, presented in this section, are considered with one important assumption: we examine isolated classes, neglecting their relations with other classes in the model and other roles they may play in the model. Otherwise, the combinatorial complexity of available options and established constraints increases significantly. At this moment, this is beyond the scope of this paper.

3.3.1 Mapping Associations 1:1

Associations of type 1:1 usually relate one main class and one dependent class, or both classes may represent strong entities, but always related as 1:1. All of the three mentioned mappings can be applied for this type of associations.

The embedded Class Approach (1:1): If the properties of both classes are often combined and retrieved in queries, than this is the most efficient transformation. This mapping eliminates frequent joins, while keeping the updates still reasonably easy. It is important to mention that objects share the same record, no matter if one of them is existentially dependent or not. It is important to know the aspect of the association's semantics and respond appropriately on operations of deleting links.
It is worth mentioning that some ORM frameworks,

such as Hibernate, support this feature but on the level of class: a class can be annotated as one whose instances will be embedded into the instances of the other class (on the other side of compositions). However, it is important to understand that this is actually a property of an association, not of an entire class: in our practice, we have come across situations in which it was very useful (according to the usage of data) to have some instances of a certain class, which are aggregated into other objects over one composition, embedded into those other objects, while the other instances of that class should be stored in a separate table. Again, to the best of our knowledge, we are not aware of any ORM or publication that supports this feature.

The embedded Foreign Key and the Distinct Table Approach (1:1): If properties of one class are dominantly retrieved, while objects of the other class are rarely accessed, then the normalized models 1 and 2 may be a better choice, since negative effects of frequent joins are minimized, while the normalized relation model responds better to further changes in the model.

Mapping Associations 1:n

The Distinct Table Approach (1:n). This is the most flexible way of mapping, but it requires expensive joins if the relation is traversed frequently. This approach provides the implementation of the bidirectional navigability.

The embedded Foreign Key Approach with Unidirectional Navigability (1:n): The embedded key approach is the most convenient and efficient for this kind of associations. The embedded foreign key is incorporated into the table of the dependent class. This saves one level of joins, while still being flexible and keeps the relational model in a normalized form. The problem with this approach is that it does not provide the bidirectional navigability (on the level of foreign keys, which are stored in only table on the *n* side of the association).

The embedded Foreign Key Approach with Bidirectional Navigability (1:n): As mentioned in the case of the embedded key approach, it does not provide bidirectional navigability (foreign keys are stored in one table only). In a special situation, the bidirectional navigability can be implemented even without the distinct association table. We may use a technique called repeating groups (Shin and Sanders, 2006). Namely, if the multiplicity of the dependent class is low, and with specified maximum cardinality, IDs of the dependent objects can be incorporated as foreign keys in separate columns in the owner object's record (Zaker, Phon-Amnuaisuk and Haw, 2009). In our future work, we will be experimenting with an ORM approach of bidirectional navigability with unlimited maximal cardinality.

The embedded Class Approach (1:n): In the case of low multiplicity of the dependent class, it may be useful to embed it to the table of the owner class. Hence, joins are not needed for traversing the association, all property values are available directly in the owner object's record.

Mapping Associations m:n

The Distinct Table Approach (m:n): The distinct table approach is a natural solution for persisting links of this kind of associations. Each record of the table contains pairs of object identifiers, for objects from both sides of the association. Although flexible, this approach requires two joins to retrieve combined attribute values from the related objects. On the other hand, updates remain fast and easy (Agarwal, Keene and Keller, 1995).

Distinct Table Approach with Controlled Redundancy (m:n): In the previous paragraph, we mentioned that the distinct table approach requires two joins to retrieve properties of the related objects. Usually, applications need to retrieve only subsets of the properties from the related classes. If the property values are rarely changed, then it makes sense to copy those properties that are accessed most frequently to the association table. This way, similarly to the hybrid approach of mapping inheritance described before, we apply a controlled redundancy to keep copies of properties that are most often retrieved when the association is traversed in the association table. This approach eliminates join operations in these cases. As for the similar approach for mapping inheritance, we are not aware of an implementation or publication of this mapping, that we propose.

The embedded Foreign Key Approach (m:n): The embedded key approach may be considered in the situation when the multiplicity on at least one side is low, known and limited with the upper bound. That way, the table of the class at the opposite side may contain repeating groups of the finite number of object identifiers as the foreign keys (Shin and Sanders, 2006). Interesting to mention, such a mapping improves performance of queries that need to read direct links of an object. If both association ends have low and limited multiplicities, we may apply the same technique on the opposite table, too. That way we get bidirectional navigability, compared to the case when only one role has low and limited multiplicity.

The embedded Class Approach (m:n): The embedded class approach is not suitable for representing associations of this kind, so we do not examine it further.

3.4 Optimizing Transitive Associations

Transitive associations are often present in UML models and traversed. If the navigation from one object to its transitively related object goes through a sequence of links, such request is accomplished by using multiple joins. Again, if this operation is done frequently, it may impose significant slowdown and high load of the database. This issue is often solved by using pre-joint tables, as recommended in (Zaker, Phon-Amnuaisuk & Haw, 2009). That is, objects that are accessed together frequently are stored in the table for the direct navigation. In addition, this optimization can be justified only by the dynamic profiling of data access in the relational database. In fact, this technique is just a special case of storing derived properties (associations in this case).

3.5 Storing Derived Values Instead of Frequent Recalculation

This transformation provides optimized access to derived values. If derived values are calculated frequently, and if the basic values change rarely, then it is highly recommended to store the derived values in redundant columns and retrieve them on demand (Shin and Sanders, 2006). This is a large category of particular techniques that cover attributes as well as associations, including functional and recursive ones. Due to the lack of space, we will not investigate this category any further in this paper.

4 CONCLUSIONS

In this paper, we presented a survey of ORM transformations aimed for creating optimized relational models, specifically structured to eliminate expensive database operations in queries. Based on the given survey, we plan to implement a systematic approach for automatic mapping UML models to the optimized relational model, although some of the techniques are already present in our SOLoist framework as particular solutions. The approach is intended to provide static, as well as dynamic application profiling that will feed ORM with information needed for adapting the relational model to support the most dominant data access patterns. Finally, one of the most important contributions of this research is an initial framework for a detailed analysis and comparison of the presented ORM approaches based on denormalization of the relational model. As a result of the analysis, we expect to produce a methodology for using the denormalization techniques in a most efficient way.

REFERENCES

Batini, C., Stefano C., Navathe S., 1989. Conceptual Database Design. Entity Relationship Approach, *Elsevier Science Publishers BV* (North Holland).

Shin, S. K., Sanders, G. L., 2006. Denormalization strategies for data retrieval from data warehouses. *Decision Support Systems*. 42 (1). p. 267-282.

Sanders, G., Shin. S. K., 2001. Denormalization effects on performance of RDBMS. System Sciences, 2001." *Proceedings of the 34th Annual Hawaii International Conference on*.

Maier, D. (1983). *The theory of relational databases.* Rockville: Computer science press. Vol. 11.

Keller, W. 1997. Mapping Objects to Tables: A Pattern Language. *Proceedings of the 1997 European Pattern Languages of Programming Conference*. Irrsee. Germany.

Keller, W., Coldewey, J. 1997. Relational Database Access Layers: A Pattern Language. *Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences*. Washington University, Department of Computer Science, Technical Report WUCS 97-07.

Keller, W., Coldewey, J. 1998. Accessing Relational Databases: A Pattern Language. *Pattern Languages of Program Design 3*. Addison-Wesley.

Milicev, D. (2009). *Model-driven development with executable UML.* Wrox.

Codd, E.F. 1971. Normalized data base structure: A brief tutorial. *ACM SIG- FIDET Workshop on Data Description, Access, and Control.* San Diego, California.

Agarwal, S., Keene, C., Keller, A. M. 1995. Architecting object applications for high performance with relational databases. *OOPSLA Workshop on Object Database Behaviour, Benchmarks, and Performance*, Austin (Vol. 196).

Zaker, M., Phon-Amnuaisuk, S., & Haw, S. C. 2009. Hierarchical Denormalizing: A Possibility to Optimize the Data Warehouse Design. *International Journal of Computers*. (1).