



PERSISTENT

Persistence Model Patterns

A Practitioner's View

Ajay Deshpande

ajay_deshpande@persistent.co.in

Software Architect, Messaging Technologies

Pratik Soares

pratik_soares@persistent.co.in

Software Engineer, Performance Engineering Group

Sameer Thakur

sameer_thakur@persistent.co.in

Senior Technical Lead, Enterprise Software.

Persistent Systems Ltd,
402, Bhageerath, Senapati Bapat Road
Pune – 411016, India
<http://www.persistentsys.com>

Abstract

Over a period of time, there have been a lot of efforts spent in getting Relation Databases to be the backend persistent stores for object oriented environments like Java. This document discusses some of the findings from our experience with building complex systems using the Object Relational Model. We discuss some of the Best Practices that have worked for us in these projects.

Introduction

In the past decade there has been a fundamental shift in the way enterprise applications are built. More often than not, these applications have a web interface. In earlier times, the underlying data model (usually relational) was designed first and the application logic weaved around it. However in the last decade, developers have started designing the data model in the application domain. In other words if they are using the Java platform it would only be natural for them to do the data model in Java. In this case they need to find a convenient way to persist the physical data to a data store because relational databases are so common, the corollary is that the persistence layer is usually relational. This has led to the rise of Object Relational Modeling (ORM) Systems.

However even today, there are no elegant solutions to this disparity. The solutions that are available, address only the mapping of data members in the object oriented world to the containers (usually columns) in the relational world. Typically a first class object in Object Oriented Programming (OOP) paradigm is self-contained – that is it has data and methods (or behavior) to manipulate the data. The present days ORM do not address the problem of storing behavioral code and hence remain incomplete. Our experience in working in this field has led to some best practices. The goal of this document is depict what has worked well for us and what has not for these persistence models.

Problems we have seen

As a part of evaluating various projects at [Persistent Systems](#); using different technology stacks for performance and scalability, we have seen some good and some not so good practices. When using the J2EE stack using hibernate as the data access layer, following are the common problems observed:

- **Relationships are marked as an eager fetch:** Often it so happens that the developer is not able to correctly diagnose the lazy initialization error (that hibernate throws using the fail fast principle), that occurs when a database entity's join is traversed outside a Hibernate session. To avoid this error, the developers rush to mark the relationship earlier not realizing the performance penalty it may have.
- **Fetch more than the required attributes:** When an object is required to be displayed for the web, we tend to fetch the entire object, not just the attributes that are required with a tendency to use “select *” type of queries. But this may lead to performance problem especially when the table has a large number of columns, (say 50+) or the table has blobs or clobs (note that hibernate marks blobs, clobs as lazy fetch by default).
- **No batch processing:** In a situation when a large number of rows need to be processed, developers sometimes call the save action after each row is edited. This can lead to a very chatty connection to the database which is inefficient. When processing a large number of rows it is recommended to take advantage of the batch processing API of Hibernate.
- **Processing/filtering results in client:** Sometimes it is observed that a lots of rows are fetched from the database and then some kind of filtering is done 'in code.' this is inefficient because more than the required rows are fetched, more network time is required, and more CPU cycles are required to filter the results. Ideally this should be done using a where clause.
- **No cache / frequent re-querying:** Infrequently changing columns/tables are fetched again and again from the database. This can be especially avoided when the database is accessed by a single application.

- **Closing and reopening session multiple times within a web request:** Instead of using an open session in view' design pattern, sometimes people handle opening hibernate session in the action as required. This if done incorrectly results in opening and closing of the session multiple times for a single web request. Thus it invalidates the already fetched hibernate objects, which have to be re-fetched again when required with the connection churn being reduced. The open session in view pattern has its own set of advantages.
- **Unnecessary use of saveOrUpdate*:** Once an object is associated with a session, all changes made to the object are automatically updated. There is no need to call saveOrUpdate explicitly.

```
hibernateSession.getTransaction();
User user = new User();
hibernateSession.saveOrUpdate(user);
user.setLoginName("mj");
user.setPassword("abc123");
hibernateSession.saveOrUpdate(user); /*BAD!*/
hibernateSession.getTransaction().commit();
```

Best Practices

The following 'best practices' are the compilation of our learnings and experience of working with a large number of projects:

- **Analyze Your Data Access Patterns:** One of the key things detrimental to the performance of a system having an ORM is the data access patterns. Hence it is very important that the developer analyzes and defines the access patterns for the data upfront. It is better to have an incorrect pattern initially, rather than have nothing at all. Handling object navigation such that only the requisite amount of data is fetched is the key to good performance.
- **Design Object Model First:** For historical reasons and sometimes for comfort levels, developers tend to begin by designing the relational data model. Then they try to fit an object model in the OOP world to the underlying relational model. Typically this is an iterative process, but our experience shows that starting with Object Model is a better alternative.
- **Need Good Understanding between DBAs and Developers:** The schema of the data model is driven from either side, we start with the Object Model, then define an implementation Relational Model. This in turn gives feedback to change the Object Model again and has a better chance to succeed if the Developers and DBAs work in tandem from the start.
- **Run Performance Tests Early On:** Once the data access patterns are defined, the system designers should set up performance runs. The data for these runs could be auto-generated by tools. The designers should select a few access patterns that can cover at least half of the eventual data accesses for the performance run. The system should be provisioned with data according to the conservative and optimistic estimates of the numbers of data entities.
- **Plan Pagination from the start:** We have noticed over many situations that projects eventually need pagination of data access, although they do not plan for it upfront. A typical example is the number of plans that subscribers can choose from. Initially it looks like we do not need pagination for this data set – however experience has shown us otherwise.
- **Lazy Fetching Should be Primary Strategy:** Designers should analyze the data access patters in their applications. One needs to understand the performance implications of each of the strategies. More often than not it is a good practice to start with using Lazy Fetching as their primary strategy for getting data from the store. Then they should tune upwards to Eager Fetching as they go along their data access patterns. Check out and understand the batching facilities provided by your ORM tool.

- **Use SHOW_SQL:** A switch in the hibernate configuration file helps you to see the SQL's being fired by hibernate, identifying potential query relate inefficiencies.
- **Differentiate between get() and load():** The load method is intended to be used when you know an instance actually exists in the database. This method actually returns proxy objects that alleviate database hits until transaction commit time, making it a little more efficient. The get method is better used when you don't know for sure if the instance you are loading, or getting, actually exists in the database
- **Override equals() and hashCode():** This ensures that two instances with exactly the same state are compared (representing the same row), the actual properties the object contains will be compared, and the compiler will not simply look at the memory locations of objects when performing an equality comparison.

Case Study I:

This case pertains to building a Secure E-Mail System (SES for short). It is implemented using Java technologies like *JSF, Seam, EJB 3.0 & JPA*. The SES application provides a contact management module. The primary function of this module is to allow user to manage his contacts (CRUD actions like update email address, phone number).The module also allows user to import contacts from popular formats.

Inside the system the contacts are stored in the contacts table. The user has many contacts, contacts can belong to one or more groups, and the groups can contain one or more contacts.

So the schema in short is as follows:

UserProfile <1----m> contacts <m----m> groups

The way it was implemented in Hibernate is as follows:

```

@ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "userId")
    public UserProfile getUserprofile() {

        @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy =
"contact")
        public Set<Groupcontact> getGroupcontacts() {

---group contact
        @ManyToOne(fetch = FetchType.EAGER, cascade = CascadeType.REFRESH)
        @JoinColumn(name = "groupId", nullable = false, insertable = false,
updatable = false)
        @NotNull
        public Group getGroup() {

        @ManyToOne(fetch = FetchType.EAGER)
        @JoinColumn(name = "contactId", nullable = false, insertable = false,
updatable = false)
        @NotNull
        public Contact getContact() {

```

All the relationships (UserProfile to Contact, Contact to group) were marked as eager. Hence when creating contact; all the corresponding groups, their corresponding Contacts and UserProfiles were queried for, and fetched from the database. This resulted in a lot of queries being fired against the database; a lot of memory was consumed to hold the contact structure. This did not pose a problem for a very small number of contacts, but as the number of contacts increased the contacts operations started hogging resources. Hence operations such as importing more than 200 contacts would fail.

This problem was diagnosed after observing the Hibernate query debug logs. The problem was rectified by marking the relationships as lazy.

About Persistent Systems

Established in 1990, Persistent Systems is recognized as an award-winning technology company specializing in outsourced [software product development](#). With 4,200 employees, innovative business models, and reusable assets and frameworks, Persistent helps its customers increase revenues and margins, and enhance brand value. Persistent Systems has delivered over 2,000 software product releases to their 170+ customers in the last five years. It has developed proven processes for the entire product lifecycle which reduce time to market while delivering consistent quality and customer satisfaction – as evidenced by customer partnerships that span many years. www.persistentsys.com

References:

* Just How Does Hibernate Work???

<http://www.hiberbook.com/HiberBookWeb/learn.jsp?tutorial=07howhibernateworks>

The ORM "Problem"

http://homepage.mac.com/s_lott/iblog/architecture/C20070522153704/E20090325060144/index.html

Modeling to Avoid Hacking Your Model in ORM Mapping Format

<http://www.artima.com/weblogs/viewpost.jsp?thread=247671>

Best Practices for Object/Relational Mapping and Persistence APIs

http://www.developerdotstar.com/mag/articles/o-r_mapping_persistence.html

Patterns of persistence, Part 2: Increase code reuse and enhance performance

<http://www.ibm.com/developerworks/java/library/j-pop2/>

ORM and the misleading DAO pattern

http://www.theserverside.com/news/thread.tss?thread_id=40581