A Case Study of Applying Object-Relational Persistence in Astronomy Data Archiving

Shilong Stanley Yao, Rafael Hiriart, Irene Barg, Phillip Warner, Dave Gasson

Data Products Program, NOAO/AURA, 950 North Cherry Ave., Tucson, AZ 85719

Abstract.

NOAO Science Archive (NSA) team is developing a comprehensive domain model to capture the science data in the archive. Java and an object model derived from the domain model well address the application layer of the archive system. However, since RDBMS is the best proved technology for data management, the challenge is the paradigm mismatch between the object and the relational models.

Transparent object-relational mapping (ORM) persistence is a successful solution to this challenge. In the data modeling and persistence implementation of NSA, we are using Hibernate, a well-accepted ORM tool, to bridge the object model in the business tier and the relational model in the database tier. Thus, the database is isolated from the Java application. The application queries directly on objects using a DBMS-independent object-oriented query API, which frees the application developers from the low level JDBC and SQL so that they can focus on the domain logic. We present the detailed design of the NSA R3 (Release 3) data model and object-relational persistence, including mapping, retrieving and caching. Persistence layer optimization and performance tuning will be analyzed. The system is being built on J2EE, so the integration of Hibernate into the EJB container and the transaction management are also explored.

1. INTRODUCTION

Astronomy data archiving manipulates large volumes of semantic-rich data. Meta-data, as the portal to the massive data products, is of special interests, because it contains numerous science parameters describing a large number of domain objects with complex structures and interactions. An effective modeling of meta-data is essential to the archive design. The NSA (NOAO Science Archive) is developing a comprehensive domain model to better capture the science data. Therefore a powerful server side development platform is desired to realize this complex model.

1NOAO/CTIO, Casilla 603, La Serena, Chile
The NSA release 2 was successfully developed majorly based on MySQL and PHP. As the domain model evolves, this platform is no longer effective enough due to its limit on inter-operation, integration and scalability. Application servers emerged recently to provide standardized, scalable and flexible platforms to develop and deploy enterprise systems. J2EE is among the popular and proved standards for the application servers. It is backed with Java technology and provides cross-platform, easy integration across multiple tiers, and high scalability. As a standard, it also guarantees the compatibility across J2EE compliant application servers. Due to the various benefits, for the next generation NSA R3, we will migrate onto the J2EE technology and Java (e.g., Servlet, JSP, EJB) would be the major server-side programming language.

J2EE enterprise applications normally consist of 4 tiers, namely client, presentation, business and EIS (enterprise information systems, or database). RDBMS (Relational Database Management System) is the best proved technology for EIS. However, there is a major challenge in this architecture, which is the paradigm mis-match between the Java object oriented model and the RDBMS relational model. Many mapping solutions have been developed to sew these two pieces together, such as Hibernate, Entity Beans, JDO. After a fully comparison in terms of robustness, maturity, features and performance, we chose Hibernate to be our ORM solution.

2. ARCHITECTURE

Figure 1 shows how the Hibernate fits in the overall architecture of NSA. Hibernate locates between the business and database tiers of the application and abstracts the database details to provides the business tier with a common interface to persistent objects, connection pool, transactions, and so on, no matter which specific underlying DBMS it is and if the system is running in a managed container on some application server or directly through JDBC.

3. RELATIONAL MODEL

According to the object-oriented development process, the NSA team produced an object model derived from the domain model. This is usually a fine-grained model. It is trivial to do a one-to-one mapping between classes in the object model and tables in the RDBMS, but it will bring poor performance. For a
better performance, we did normalization and de-normalization on the relational model. Generally, we try to keep the tables at 3NF (3\textsuperscript{rd} Normal Form). However, when business keys are not obvious, we use auto-generated surrogate keys; some fine-grained per-object tables are merged. These downgrade some tables from 3NF, but improve the performance.

4. OBJECT-RELATIONAL MAPPING

4.1. Mapping

Mapping between objects and tables is the core part of the O/R persistence. We will address a few most important issues here. Inheritance hierarchy mapping is important to support efficient polymorphic query on the objects. We have two major mapping strategies depending on different situations. When subclasses tend to have more common attributes, we use table-per-hierarchy mapping, where the whole class hierarchy is mapped onto a single table, with a special column to discriminate different classes in the hierarchy. With this mapping, no join is required during a polymorphic query. When subclasses tend to have more uncommon attributes, table-per-hierarchy won’t work well due to too many null values in the table. Instead, we use table-per-subclass mapping, in which each subclass has a separate table storing attributes only owned by itself. Common attributes inherited by subclasses, are mapped to the super class’s table. This mapping requires table joins during queries, but also reduces the number of null values, which actually improves the overall performance.

Hibernate provides mapping of most conventional Java types, like primitive types, wrapper classes (e.g., Long), etc. However, astronomy data has many flexible types like Dimension, vectors, arrays. So mapping non-conventional java types efficiently is critical. A simple default solution is to map those types to binary data type in RDBMS (e.g., “Blob”). But this will create difficulty to the query performance, because WHERE clause involving any part of the blob require reading the whole binary block. We use Hibernate API for customized persistent user types instead, which can map each part of a complex data type to a separate column in the table. This greatly optimizes the database operations.

In the object model, some small objects could form a logical group, for example, astronomer’s home address, contact (e.g., email, phone), and NSA account (e.g., ID, role). Having one table for each of the 3 objects and joining them during the query is inefficient. In the reality, there will be very few astronomers sharing the same home address, phone number, or account ID. So mapping all three objects into a single table would reduce the number of tables and the cost of joins. We do this by mapping objects as components.

Other issues like mapping of class associations, auto-generated object ID, etc. are implemented in standard ways with hints from the object and relational model, which we won’t discuss in detail due to space limit.

4.2. Retrieving

The object model really presents us a “graph”, in which classes are nodes and the associations are edges. Querying is done on this graph, while the database tables are joined and rows are transformed into resulting objects. Hibernate does
this by issuing SQL statements to the database. So the key issue of database
performance is to minimize the SQL round-trips and the data bandwidth for
each query. And this is the challenge of object retrieving.

Hibernate provides "lazy" retrieving, which means that objects are not
retrieved until the application actually visit them. For example, an astronomy
observation object has a long list of facility objects as its members describing
the details of the telescope, camera, filter and so on, with which the observation
was taken. If the application is answering a user’s question regarding to the date
of the observation, it doesn’t make sense to load all the member facility objects.
With “lazy” retrieving, this overhead is eliminated.

But in some cases, we need “eager”, the opposite of “lazy”. Continuing the
above example, if a user want to see the full details of the facility configuration
of the observation, it’s better to join the tables of all facility objects and the
observation table and finish the query with a single SQL statement. So in the
NSA, we configure the mapping to use “lazy” retrieving as the default and
explicitly override it with “eager join” on some queries, which we are sure all
members will get loaded anyway.

4.3. Caching and concurrency control

Hot-spot data is frequently accessed. For example, some telescopes are exten-
sively used by many recent observations. It is best to cache them in the memory
rather than loading them every time they are referenced. Hibernate provides
two levels of such cache with different scopes, namely level-1 per-session cache
and level-2 per-process cache. Level-1 cache is required, while level-2 is optional
for caching process-wide frequently read but less frequently modified objects.
We specify objects, like Institute, Permission, to be level-2 cached.

Hibernate caching feature provides more flexible concurrency control mech-
anism. Hibernate provides optimistic locking (based on versioning and level-1
caches) to support “repeatable reads” isolation. This improves performance
in most cases in the NSA, where concurrent read-write and write-write conflicts
tend to be rare. Versioning info (a sequence number) is a special attribute added
to the object. In this case, more expensive DBMS level pessimistic locks are not
acquired. Instead, the version number is incremented after each modification
and during committing the version number of the object is checked against the
old version number when we read it in. If no other transaction modified this
object on our back, the two version numbers should match and we could safely
commit. Otherwise, we abort.

Although optimistic locking works well for us most of the time, there are
cases where conflicts are highly possible, and we still want to use the DBMS pes-
simistic locking. Multiple instances of running ingestion services is an example.
In this case, we explicitly specify, during the queries, that Hibernate locking is
bypassed and a certain type of pessimistic lock is acquired at the DBMS level.

4.4. Transaction and Session

In concept, we have two kinds of transactions. Application transaction is a
coarse-grained unit of work from the point of view of the users. Database trans-
action is the transaction from the RDBMS’s point of view. Nested transactions
are not well supported by EJB, Hibernate or JTA, so we use flat CMT (Con-
tainer Managed Transaction). Hibernate supports CMT, because its transaction API is only an abstraction of the underlying JTA. In case of session bean calling session beans in the same transaction, the callee joins the caller’s transaction instead of starting a new one.

Hibernate session dictates the level-1 cache and object identity scope. For a long application transaction (e.g., astronomers read some data, modify it, and store it), we use session-per-request with detached objects. This reduces resources tied up during the modification and minimize the stale data.

5. IMPLEMENTATION

We build the NSA on JBoss application server. Hibernate is used at the ORM layer. With the aid of ANT build tool and xDoclet code generation engine, most code and script generation is automated. Hibernate mapping files, which specify the details of ORM, are generated from the tags we embed in the Java source code. Although, Hibernate API is already very concise, we built utility wrappers around it to further simplify the persistence code in the business tier. Hibernate resources, including library files and mapping files, are automatically packaged into a service deployment unit for the JBoss server, which is utilized by the application through JNDI.

6. PERFORMANCE TUNNING

For retrieving, we will adjust the lazy fetching and the global maximum depth of eager fetching. Both consulting with the astronomers and collecting production system performance hints will be used to study the data access pattern, such as which ones among the many observation parameters are most interesting. The less frequent accessed data will be tagged for lazy fetching.

In a polymorphic query, the number of SQL interactions or the number of tables joined is roughly a linear function of the hierarchy depth. So the shallower hierarchy depth is more desired. For a good balance of the performance and soundness of the object model, an iterative approach should help, which suggests a revisit to the object modeling with the hints from Hibernate implementation.

The level-1 cache of the Hibernate provides opportunities for optimizations. Besides the optimistic locking, immutable objects is another example. Immutable objects will never be flushed. Facility objects (e.g., filter, CCD) are good candidates of immutable objects. From the data access pattern, we will adjust the list of immutable objects. However, another tradeoff related to this is the performance improvement and the portability of immutable objects. We learned that not all ORM middle-ware provide this facility.

References

Christian Bauer, Gavin King, Hibernate in Action, Manning, Aug 2004