GLORP: Generic Lightweight Object-Relational Persistence

Alan Knight
Cincom Systems, Inc.
knight@acm.org

While object databases are technologically better adapted to storing object data, most of the world's information is still in relational databases. This makes it critical for object-oriented applications to be able to access data in relational databases. Unfortunately, this is a harder problem than it might initially seem, and one that is a major risk for projects.

Simple approaches can be used for basic mapping -- assigning a class for each table and an instance variable for each column, managing relationships manually -- but these do not scale well as complexity and performance requirements increase. A particular difficulty is that object applications must often deal with pre-existing relational schemas designed for other purposes. Mapping these schemas into objects may require a great deal of flexibility.

We will demonstrate GLORP (Generic Lightweight Object-Relational Persistence), a simple but powerful object-relational mapping layer. GLORP is an open-source project, implemented in Smalltalk, and adaptable to almost any relational data store. While still in the very early stages of development, it already offers many sophisticated features. GLORP is intended not just to serve as a mapping layer, but to illustrate the principles and patterns that can be applied to this problem space. We briefly outline what we feel are the important areas.

1 Mapping

We can divide the sophistication of storage strategies into various layers. The simplest relational storage strategy is simply to embed SQL into code whenever an object is read or written. This quickly leads to duplication and maintenance difficulties and so it is often generalized to a broker that stores SQL statements for reading and writing each class of object. This is workable, but tends to limit the operations that are possible, since each additional query or special form of write requires at least a new SQL statement, if not an extension to the broker protocol. Relationships between objects may not be modeled, forcing object developers to take foreign keys into account, e.g. `findOrdersWithCustomerID: aCustomer id`. Alternatively, relationships may `be` modeled with explicit queries (e.g. `findOrdersForCustomer:`) which must be added to the broker layer.

To go beyond these limitations requires a more explicit and declarative mapping, which is the scheme GLORP uses. Object developers do not define SQL statements. Instead, they define a mapping between object and database representations declaratively in terms of correspondences between fields and instance variables, or between relationships and foreign keys. This meta-data is represented as *mapping* objects, e.g.

```
OneToOneMapping new
   attributeName: #address;
   referenceClass: Address;
   mapingCriteria: (PrimaryKeyExpression
      from: (prsnTbl fieldNamed: 'ADDR_ID')
      to: (addressTable fieldNamed: 'ID').
```

This lets the mapping layer perform much more detailed analysis of the operations to be performed than is possible if explicit SQL statements are used. The framework can support complex and dynamic queries, dynamically modifying meta-data, transparent management of relationships, partial reads and writes of objects, and numerous optimizations.

2 Transactions

Rather than having explicit writes of individual objects, GLORP bases all writes on its transactional model, noting which objects were touched during the transaction. This is less intrusive into the application code, since it frees developers from having to remember which objects were modified. Since the transactional framework can also provide rollback on those objects, this can simplify the application programming model significantly, even without considering the database aspects.

Automatic writes on transaction boundaries is also important because it gives us the opportunity to re-order those writes. This can let us optimize the writes, minimize deadlock, and most importantly respect relational integrity constraints.

Given such a scheme, we need to determine which objects were modified in a transaction. There are several possible strategies. The least intrusive would be to inspect all objects in the cache, but this would not be likely to perform adequately. We could require the developer to explicitly register root objects, and examine only those roots and the objects reachable from them. This performs well, but requires some additional intrusion into the application code. Finally, we could create a write barrier so that any modification of an object could be detected and the transaction in which it occurred would be known. GLORP currently uses the second mechanism, but there has been design work done to enable transparent parallel transactions which would support a write barrier.

3 Non-Intrusiveness

A primary design goal of this work is not to intrude into either the relational model or the object model. This has several aspects. Firstly, we should not intrude by requiring changes to the relational schema. Since these schemas can vary widely, and may not correspond well to the desired object model this forces us to be very flexible in terms of how we can map this schema into objects. Secondly, we should not intrude into the object model. We should minimize the extent of any object model changes, and further we should not force the classes to inherit from a persistent superclass or to implement special database-related code. We should be able to work objects that were not designed to be persistent without difficulty. Finally, we should be non-intrusive in the development process, so we should not force developers to go through any extra steps during development (e.g. code-generation, pre-processing).

These goals cannot be achieved one hundred percent. It's not possible to map any arbitrary database schema onto any arbitrary object model. If we are to store objects in a relational database then we need to be able to somehow derive a primary key from those instances. If we are to store collections we must recognize that the ordering of an ordered collection cannot be automatically preserved in a relational database. Nevertheless, we try to come as close to the goal of complete non-intrusiveness as possible.

For example, GLORP objects can be stored across multiple tables. It is also possible to store multiple objects within a single row, and objects can be written to different tables in different circumstances (e.g. a money object might be stored directly with the thing that contains it in many different tables). GLORP does not require that objects in a one-to-many relationship retain a reference to their parent, and GLORP is currently entirely meta-data and reflection based so that no code is ever generated.

4 Queries

If we are to store objects in a relational database we must be able to retrieve them easily and efficiently. GLORP supports queries at the object level, the data level and mixtures of the two, as well as direct use of SQL. While most queries will be done purely in terms of objects, this flexibility means that all of GLORP's internal querying can be done with the same mechanisms that are available to the end user.

The most common query mechanism is to express them as Smalltalk blocks, using the object-level relationships to define attributes and joins. For example, to read a Person based on a property of the related Address object we can specify

```
aSession executeQuery:
   (Query
      forManyOf: Person
      where: [:person |
         person address street = 'Main'].
```

This will examine the query block, determine that it references the class Person and the one-to-one relationship to Address, and the attribute "street" from Address. Based on this information it consults the meta-data, determines which tables are necessary and the join criteria between them, then generates the corresponding SQL. Once the rows are retrieved, it examines each row, determines if the object is already in cache, and if not builds the new object.

Note that queries are addressed to a particular Session object, which also controls transactions. By talking explicitly to a session, rather than having an implicit session with queries as class methods or addressed to a general factory we can easily support multiple independent sessions within the same virtual machine.

5 Performance

Flexibility isn't useful if the application cannot perform adequately. GLORP already supports significant performance optimizations, and is architected to permit additional important optimizations.

In a database-centric application, in-memory performance is rarely the dominant factor. It would be possible to micro-optimize some of the GLORP features by imposing a code-generated write barrier, and using generated code rather than reflective access, but in practice we have not found these to be at all significant. Smalltalk supports very fast reflective access (especially compared to Java) and the most powerful optimizations have been in terms of database operations.

The most significant step is the introduction of an optimization phase during writes. When writing, we first compute the data representation of all changed or new objects, then examine the data to determine exactly what needs to be written, the write order, and any optimizations that can be applied. An example of such an optimization would be Oracle's array binding feature, which lets us perform many identical statements with different parameters as a single call with array-valued parameters. This is not yet implemented in GLORP, but it can easily be supported because of the optimization phase.

On read, queries can specify groups of objects to be read at the same time, and these can be broken down into groups which can be loaded as joins or composite reads. This can be specified declaratively, and can vary for different queries against the same objects. For example, when reading for an overview display we can read in minimal information, but when reading in for editing we can specify that related objects should be retrieved as well.

6 Links and Acknowledgements

For more information on GLORP, including source code, see http://glorp.sourceforge.net. While GLORP is intended to be portable between Smalltalk dialects and databases, most of the current development is in VisualWorks 3.0 and 5i against Oracle and PostgreSQL databases. GLORP is a Camp Smalltalk project (http://camp.smalltalk.org).

Many thanks to The Object People (now WebGain) which sponsored the initial development work on GLORP and whose TOPLink family of products has influenced its design. Thanks as well to everyone who has contributed to this effort, particularly Bruce Badger, Anthony Lander, John-Reed Maffeo, Mark Schwenk, and David Siegel.