Object-Oriented Query Language

A Technical White Paper

Zhongling Li

February 5, 2006

Introduction

Object-Oriented Query Language (OQL) is designed to provide an object-oriented query interface for traditional relation database systems (RDBMS). The goal is to bridge the gap between object-oriented programming language (specifically Java) and set-oriented Standard Query Language (SQL), and make the persistence layer fit better in an OO system design.

Background

Though object-oriented programming languages such as Java, Smalltalk, C++ and C# have dominated the server-side business application world for many years, unfortunately object-oriented database did not take off due to numerous reasons. Relational database still is the majority of persistence mechanism.

The standard programming interface to database is SQL, which was originally designed for human interaction purposes. As a set oriented language, with very limited control flow and loose grammar, it slowly becomes a barrier for a pure object-oriented system design. Within the data access layer, too often we find developers throw away the beautiful object model, but start concatenating SQL strings (either directly or indirectly) and create so many messy codes. Even worse, because of the limitable of SQL, many developers use data access layer, or the data model itself as the starting point when designing a system. This ground-up approach tends to make the final system more service-oriented or more procedural.

OQL wraps the basic SQL language into generic object-oriented APIs, and hide the underline complexities such as porting to difference databases. It acts as a mapping tool between objects to rows in database tables, but also has the capability of manipulating data with control flow. Since the output of OQL compiler is Java classes, developers can easily integrate OQL into their applications.

Design Goals

We want to make OQL a simple language and very easy to learn and use for Java developers. The syntax and grammar is very close to Java instead of SQL. Actually all the logic of generating SQL strings are completely hidden from OQL developer.

Object-Oriented

Everything is an object in OQL; there is no primitive type. All operations are defined at class level, such as Create, Retrieve, Update and Delete (standard CRUD in SQL). Those very complicated query semantics are nicely wrapped in Criteria, Projection and Join interfaces. From developers' perspective, they are simply dealing with a special set of objects, which happened to be stored in a relational database transparently to their design model.

Database Generic

OQL will be compiled into Java classes, which use ANSI standard SQL and can run against any relational databases without modification. This makes application porting easy, but it also implies a limitation of OQL, we cannot support any database specific features or improvements. In future releases, we will add database flavor option to allow optimization for specific databases.

Less Overhead

The code generated by OQL compiler should not be much slower than hand written native SQL blocks. Since compiler generates all SQL statements (with optimization in mind, such as using prepared statements, or table join ordering) instead of concatenating them at runtime, OQL can actually give better performance in some cases.

Simple

OQL has similar simple syntax and grammar like Java. It has less keyword, and does not support primitive data type. Since it serves a single purpose of presenting SQL query, the API is simple and clean.

Easy to Integrate

Since OQL compiler generates Java classes (or other OO language output in future releases), it is very easy to integrate it into existing systems as data access layer.

Portable

Java is a portable language runs on most operating systems. With its database generic feature, OQL is highly portable to many platforms.

Implementation Overview

Entity Definition

Entity is defined in a format similar to a Java class. Actually it is a special type of Java class internally which maps to a database table. Each entity contains an “id” field which is the primary key (it is a good design to have surrogate key for every table in the database).

```
Entity Address {
        Integer id,         // implicit primary key
        String street,
        String city,
        String state,
        String zipCode
}
```

Standard CRUD operations are defined at entity level.

```
    // create a new instance of Address
    Address a = Address.create(new Integer(100), "1234 Main Street", "New York", "New York",
"100020");

    a.street = "345 A Street";
    Address.update(a);                // update instance a

    Address.delete(a);                // delete instance a
    Address.delete(new Integer(100));    // delete instance by primary key

    Address b = Address.get(new Integer(101));    // query instance by primary key
```

Query Definition

Query interface is the most important API of OQL. Criteria and OrderBy are two fundamental objects in the query structure.

```
    OrderBy o = new OrderBy("zipCode", true);            // order by zipCode, ascending
    Criteria c1 = Criterion.eq("city", "New York");      // city='New York'
    Criteria c2 = Criterion.ne("zipCode", "10020");      // zipCode != '10020'
    Criteria c3 = Criterion.or(c1, c2);                  // c1 OR c2
    Address[] list = Address.query(c3, o);               // query
```

Aggregation

OQL supports basic aggregation functions like count(), sum(), avg(), min() and max(), with criteria restrictions.

```
    Integer count = Address.count(c3);          // count by criteria
    Integer sum = Address.sum("price", c1);     // sum(price) by criteria
```

Entity Join

Basic table inner joins are supported by OQL.

```
Set C { A join B on A.fk = B.id };              // C is a result set of join of entity A and B
```

Internally Set is a super class of Entity, which defines all the CRUD, query and aggregation functions.

Related Projects

There are numerous existing projects dealing with Object-Relation mapping issues, such as Hibernate, iBatis, JDO and infamous Entity Bean. OQL actually borrows many ideas from these mature projects. The difference lies in:

- Existing projects are released as libraries instead of its own language.
- Some projects tend to do more than simple things, such as entity state management, caching and complicated transaction management. Hibernate and Entity Bean is the best example.
- Though Hibernate has its own HQL language, the idea still is a mock of SQL string, just replacing the table with Java class, and column with object property.
- iBatis goes to another extreme which simply provides a Object to SQL directly mapping layer.

Roadmap

There are many features that did not make it to the original design of OQL due to time and resource limitations. In the near future, we plan to support the following list of things:

- Outer joins
- Support operations cross multiple databases
- Declarative transaction support
- Use existing Java classes as entity definition
- Optimization for specific databases, like SQL hints

Summary

OQL provides a powerful alternative to the Object-Relation access layer. It is pure Object-Oriented, simple to learn, database and platform generic. In addition, since OQL abstracted many good designs from existing OR projects, which makes it a good candidate for any new Java project which requires relational database as persistence layer.