Supporting Retrievals and Updates in an Object/Relational Mapping System

Jack Orenstein
Novera Software, Inc.
jack@novera.com

1 Overview

Novera Software, Inc. produces jBusiness, a pure-Java application server [NOVE]. Enterprise Business Objects (EBO) is an object/relational mapping system, which is part of jBusiness. This paper describes some of the transformations that EBO uses to provide transparent, high-performance database access to middle-tier Java applications.

2 Business Objects

An EBO application manipulates business objects. A *business object* is an object representing some concept relevant to the application, whose state is obtained from an underlying relational database. Much of the code of an application can be expressed in terms of business objects. Other code relies on the EBO application programming interface (API), and a collection library.

This section describes business objects, how they are mapped to Java objects, and how these objects are mapped to tables in a database. For more information, see [NOVE98].

2.1 Object Model

The EBO object model is extremely simple. Objects have identity. Objects have scalar properties, and may participate in one-to-one and one-to-many relationships.

A *simple scalar property* is mapped to a single column. A *computed* property is mapped to a SQL expression, e.g. QUANTITY * UNIT_PRICE. An *aggregate* property is mapped to a SQL expression containing an aggregation operator, e.g. SUM(QUANTITY * UNIT_PRICE). Each property gives rise to methods for getting and setting the property's value. "Set" methods are useful even for computed and aggregate properties as they allow the application to reflect changes due to actions by the application itself.

Foreign keys are mapped to one-to-one or one-to-many relationships. When a relationship is realized in Java, there are methods in the two participating classes. The methods generated depend on the cardinality of the relationship. For example, consider the one-to-many relationship between Employees and Departments. EBO will generate the methods Department.getEmployees, which returns a collection, and Employee.getDepartment,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

which returns a Department, for navigating the relationship in either direction. The methods Employee.insertEmployee, Employee.removeEmployee and Department.setDepartment modify the relationship.

Collections play two roles in EBO. First, collections are used to represent the "many" side of a relationship, as described above. Collections called *extents* are also used to represent the set of all instances of a business object class (in a database). The extent of a class can be obtained by calling the business objects extent method. For example, Employee.extent() returns the set of all Employee objects. Invoking extent() does *not* cause the retrieval of all the data in the database required to materialize all the Employee objects; it is merely a surrogate. Operations on extents result in the execution of SQL commands as described below. Because an extent is a collection, collection operations can be applied to it.

2.2 Defining Business Objects

The *schema mapper* is a GUI which assists the user in defining business objects. The schema mapper starts by connecting to a database and letting the user select tables of interest. Definitions of these tables are read in, and from these definitions, business object object definitions are created. The schema mapper generates default definitions, which can then be extended and customized. The schema mapper also lets the user customize the translation of business object definitions into Java object definitions.

In general, a business object combines one or more tables through joins, and projects selected columns. When defining a business object, one of the participating tables must be specified as the *root table*. Intuitively, the business object represents the same real-world entity that the root table does. Internally, the root tables primary key is used to determine the business objects identity.

For example, suppose the database has tables PRODUCT and CATEGORY, in which each PRODUCT has a CATEGORY. We might want to define a Product business object which is based on PRODUCT but includes some information from CATEGORY, e.g. CATEGORY_NAME. PRODUCT is the root table of Product. Product can have a CategoryName property based on CATEGORY.CATEGORY_NAME.

The schema mapper offers a simple way to turn a set of tables into a set of business objects. Each table is turned into a root table for some business object. Columns are turned into properties, and foreign keys are turned into relationships. These default definitions can be modified in a number of ways.

Once the business object and Java object definitions have been defined, the schema mapper can be directed to generate its output. The output comprises a set of Java sources, "class files" (resulting from compilation of the sources), and a *schema map*. The schema map represents all the information read and created by the schema mapper. The transformations carried out by the EBO runtime are guided by the schema map, as described in the following sections.

3 Simple Retrievals

For each business object, a method named *find* is generated to locate the object given its primary key, e.g. Customer.find(419). The implementation of find first checks the EBO cache to see if it contains the object, and if it is non-stale. A database query is executed if no such object exists. The SQL is generated by simply adding primary key selection clauses to the business objects view definition. *Navigation* describes the invocation of a business objects method which returns either another business object, or a set of business objects. These are the *get* methods associated with relationships. As with *find*, a database query is issued only if the required object(s) are not present in the cache.

4 View Expansion

4.1 View Table Queries and Base Table Queries

Consider the tables PRODUCT and CATEGORY, and the Product business object, (described in section 2.2). In the terminology of the EBO implementation, PRODUCT and CATEGORY are *base tables* - they are defined in the database. EBO also has the concept of a *view table* which corresponds to the view definition mapping Product to PRODUCT and CATEGORY. (When we use the term view we mean a definition stored in the EBO schema map; we never refer to a view definition in the database. A view in the database is just another (base) table.)

There is one subsystem, the *sql* subsystem, that has an API for creating SQL queries, adding restrictions, join terms, group by clauses, order by clauses, and so on. Those parts of EBO that translate retrieval operations into SQL use this API to build up SQL queries *in terms of view tables*. The sql subsystem is responsible for expanding the view definitions to come up with SQL queries involving base tables.

4.2 Outerjoin Translation

Now assume PRODUCT and CATEGORY tables are connected by a foreign key from PRODUCT.CATEGORY_ID to CATEGORY.CATEGORY_ID. Because the Product business object includes CATEGORY.CATEGORY_NAME, Product's view definition includes a join between the PRODUCT and CATEGORY, across the foreign key. To provide reasonable semantics to Java applications, the join must be an outer join. For example, iteration over the extent should locate all Products, including uncategorized ones, (i.e. PRODUCT rows with CATEGORY_ID = NULL). Such products are dropped by an inner join between PRODUCT and CATEGORY.

Outerjoin support by relational database systems is quite uneven. For this reason, we do not rely on native outerjoin capabilities. Instead, we simulate outerjoins using other techniques, e.g. a UNION of subqueries. This is quite expensive compared to an inner join, and in many situations the extra work is unnecessary maybe there really arent any uncategorized products, and the user knows this. To avoid (simulated) outerjoins when possible, the schema mapper allows the user to declare, for each foreign key, that certain joins can be inner joins for the purpose of materializing a business object.

4.3 Update Translation

Updates are handled differently due to the usual difficulties in updating through views. Whenever SQL update code is generated, (using the INSERT, DELETE or UPDATE commands), the updates are expressed directly in terms of base tables; i.e., the sql subsystem is not used for view expansion.

4.4 Query Language

EBO provides a Java-like query language resembling Orion [KIM89] and ObjectStore query languages [OREN92]. It was designed to be familiar to Java programmers rather than SQL programmers, and this is reflected in the syntax. It does, however, have capabilities similar to several SQL extensions (e.g. OQL [ODMG97]).

Example 1: Find Customers whose company name matches a given pattern.

```
Query company_name_query = DbBlend.createQuery
(``Customer customer; String pattern:  ''+
``customer.getCompanyName.like(pattern)'');
```

This query declares a Customer range variable, customer, and a free variable containing the pattern. The predicate locates those companies whose name matches the pattern. getCompanyName is a method for accessing the CompanyName property. (DbBlend is an earlier name for EBO. EBOs API still refers to this older name.)

Example 2: Find Customers still waiting for orders placed more than one year ago.

```
Query major_customers = DbBlend.createQuery
(``Customer c; Date order_date:  c.getOrders[''+
`` Order o:  o.getOrderDate ≥ order_date && ''+
`` o.getShipDate == null]'');
```

This query contains a subquery which looks for qualifying Orders. A Customer appears in the result if it has at least one qualifying Order.

5 Basic Query Translation

An EBO query is parsed, creating a parse tree in which leaves correspond to range variables, free variables, and literals; and internal nodes represent operators such as navigation (the "dot" operator), comparisons, boolean operators, and nested queries. Then a bottom-up scan of the parse tree is performed. At each node, some portion of a SQL query is created, using the API of the sql subsystem. The results of this generation are associated with the node. When this traversal is complete, the root of the tree contains a complete SQL query.

6 Prefetch

Consider an order entry GUI that deals with Customers, Orders, OrderDetails, and Products. Given a Customer's primary key, the GUI might need the Customer's Orders, each OrderDetail of each Order, and all Products referenced in any Order. All this information can be retrieved by navigation, using the methods Customer.getOrders, and Order.getOrderDetails, OrderDetail.getProduct. EBO provides an alternative. A description of a path can be supplied when the query is created. If this is done, then the SQL generated is modified to *prefetch* the related objects along the path specified. Without prefetch hints, the number of SQL queries generated by the navigation is proportional to the number of objects retrieved. With prefetch, different queries are issued, and the number of queries is proportional to the length of the prefetch hints path description. (In the example above, there will be exactly four queries, one for each class: Customer, Order, OrderDetail and Product.) For example, here is code that locates Customers whose company name contains the string "IBM", and prints descriptions of their Orders:

```
Query ibm_customer_query = dbblend.createQuery
(``Customer c:  c.getCompanyName.like(\''%IBM%\``)'');

for (ibm_scan = Customer.extent().elements(ibm_customer_query);
ibm_scan.hasMoreElements();) {
Customer customer = (Customer) ibm_scan.nextElement();
for (order_scan = customer.getOrders().elements();
order_scan.hasMoreElements();) {
Order order = (Order) order_scan.nextElement();
print_order(order);}
}
```

Assuming execution begins with an empty cache, there is one query to the database to fetch the qualifying Customers, and then another query for each Customer to get that Customer's Orders. The Order queries would look something like this:

```
SELECT o.order_id, o.order_date, o.ship_date
FROM ORDER o
WHERE o.customer_id = ?
```

The value of the free variable (denoted by ?) is the primary key of the current Customer.

Two modifications are required to use prefetch. First, the navigation path needs to be described:

1. Create a path description starting at Order
```
PathSet customer_to_order = dbblend.createPathSet(``Customer'');
```
2. Extend the path by navigation through getOrders to an Order.
```
customer_to_order.add(``getOrders(Order)'');
```

Then, add the path description as a second input to the createQuery method:
```
Query ibm_customer_query = dbblend.createQuery
(``Customer c:c.getCompanyName.like(\''%IBM%\``)'',
customer_to_order);
```

Now, when the query is executed, both Customers and Orders will be retrieved and placed in the cache. Instead of running multiple queries looking for the Orders rows with a given customer id, there will be one query to get the Orders for all qualifying Customers:
```
SELECT o.order_id, o.order_date, o.ship_date
FROM ORDER o, CUSTOMER c
WHERE c.COMPANY_NAME LIKE *%IBM%*
AND o.CUSTOMER_ID = c.CUSTOMER_ID
```
As a result, navigation from Customer to Order using customer.getOrders never needs to go to the database; the result is already in the cache.

PathSets provide a very easy way to tune performance. Application developers can develop their business logic using queries and navigation. When performance problems are detected, they can often be addressed by simply adding a PathSet to the queries, leaving the bulk of the business logic alone. If a query is created with a PathSet argument, then there is an additional phase in query translation. Following the creation of the initial SQL query a new query is generated for each step of each path represented by the PathSet.

7 Conclusions

EBO has proven successful in practice. Customers have quickly developed applications by mapping database schemas to business objects, writing applications in terms of business objects, and letting EBO take care of nearly all interaction with the database. Typical applications involve 100-150 tables, mapped to a smaller number of business objects.

8 References

[KIM89] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, Darrell Woelk: Features of the ORION Object-Oriented Database System. OO Concepts, Databases, and Applications 1989: 251-282

[NOVE] Novera web site, http://www.novera.com.

[NOVE98] Developing Enterprise Business Objects, Novera Software, Inc., December 1998.

[ODMG97] The ODMG 2.0 Specification, Morgan Kaufmann, (1997).

[OREN92] Jack A. Orenstein, Sam Haradhvala, Benson Margulies, Don Sakahara: Query Processing in the ObjectStore Database System. SIGMOD Conference 1992: 403-412.