ODBMS for RDBMS Users

By Rick Grehan

The goal of this article is to provide an introduction to ODBMS concepts for users of RDBMS systems.

Luckily, this is easier now than it might have been, say, a decade ago. Object-oriented programming languages have become so ubiquitous that it is unlikely any reader isn't using some object-oriented programming language dialect. And, if you are one of the few readers who are not (or have not), this introduction will be of little benefit to you. After all, what's the point of using an object-oriented database if you don't have any objects to put into it? (For those few non-OO programmers, we suggest you select one of the many object- oriented programming languages, acquaint yourself with it by experiencing the ups and downs of writing some OO programs, then return here.)

We also assume that you are familiar with classical relational databases. If not please refer to the www.odbms.org portal's free downloads or book suggestions on "Databases in General".

A Thought Experiment

Suppose you've written an OO program; we'll pretend it's in Java. In that program, you have a number of objects whose data you would like to store into a database. More accurately, you want the *objects themselves* stored into a database ... you don't want the database to consist of rows of data values, you want it to consist of objects.

Furthermore, you want to operate on those objects the same way you operate on, say, a row in a database table. So, you want to be able to add objects to the database; you want to be able to search for objects in the database; you want to be able to update objects; delete objects, and so on. (In database-speak, we would say that you want to make the objects *persistent*. On the one hand, you want them to behave like things stored in a database, in that you can put them in the database, close the application, turn your computer off, and -- the next day -- turn your computer back on, open the database, and fetch those objects out of the database. On the other hand, you want those objects to act like ... well ... objects.)

How do you go about doing this?

First, let's assume that you decide define one table per class. This seems resonable, because in one sense a class describes -- among other things -- a relation. (We should point out that this is not the only strategy for *mapping* classes to database tables. But we'll leave the discussion of those other strategies for later.)

Next, you examine the class' data elements, and from that you derive a format for each record (row) in the table. So, assuming that you have an `Employee` class defined as follows (showing only the data elements):

```
public class Employee
{
  string lastName;    // Last name
  string firstName;   // First name
  int IDNumber;       // Employee ID
  Money salary;       // Annual salary
```

```
  ...
}
```

(Note: In the above example, we assume that there is a `Money` class defined.) Given the above, you create a database table using the following SQL code:

```
CREATE TABLE Employee
(  lastName  VARCHAR(50) NOT NULL,
   firstName VARCHAR(50) NOT NULL,
   IDNumber  INTEGER NOT NULL,
   salary    DECIMAL(10,2) NOT NULL
);
```

Graphically, we could represent the table as shown below (where we've substituted more human-friendly names for the table columns, and added some sample data):

Last Name	First Name	ID Number	Salary
Jones	Jeff	0001	30000.00
Smith	Jane	0002	40000.00

Figure 1. *Graphical representation of the Employee table.*

So far, so good. But now, you'll have to write code that moves data back and forth between the relational database and objects. That is, you need to create code that can take the data items fetched by an SQL `SELECT` statement, instantiate an object, and move those data items into the instantiated object.

In Java, code to do that would look something like this:

```
...
ResultSet rSet = statement.executeQuery(
  "SELECT lastName, firstName, IDNumber,
  salary " + " FROM Employee");

while(rSet.next())
{
  Employee emp = new Employee();
  emp.lastName = rSet.getString("lastName");
  emp.firstName = rSet.getString("firstName");
  emp.IDNumber = rSet.getInt("IDNumber");
  emp.salary = Money.fromDouble(rSet.getDouble(
   "salary"));
  ...Do something with the emp object...
}
...
```

...where we assume that our `Money` class includes a `fromDouble()` method that converts from a `double` datatype to a `Money` datatype. The above code executes an SQL `SELECT` statement on the `Employee` table, and places the results into the `ResultSet` object `rSet`. You can iterate through `rSet` to retrieve individual rows and columns.

Still not quite there yet. The above code snippet gets data out of the database. You'll want to get data *into* the database, too. There are a couple of ways to do that. The first way is to just put everything into a string, and call the `executeUpdate()` method. So, you can add a new employee with:

```
...
int result = 0;
Statement statement =
 connection.createStatement();

...instantiate Employee object emp...
...and initialize it's data members...

result =
   statement.executeUpdate(
   "INSERT INTO Employee " +
   "(lastName, firstName, IDNumber, salary) " +
   "VALUES('" + emp.lastName + "', " +
   "'" + emp.firstName + "', " +
   emp.IDNumber.toString() + ", " +
   emp.salary.toString() + ")" );
...
```

(In the above code, we assume the presence of a `connection` object to the RDBMS.)

Or, you can use the `PreparedStatement` interface to create a pre-compiled statement, and use *placeholders* to transfer data values into the SQL code. For example, if you wanted to update employee #1's salary to $30,000.00, you could use code that looks like this:

```
...
int result = 0;
PreparedStatement pstmt =
 connection.preparedStatement(
 "UPDATE Employee SET salary =
 ? WHERE IDNumber = ?");
ptmt.setDouble(1,(double)30000);
psmt.setInt(2,1);
reslt = pstmt.executeUpdate();
...
```

This bit of code has just about lost all semblance of object-oriented programming. There is no `Employee` object. We've simply updated data values directly into the table. (Of course, we could instantiate an `Employee` object, and initialize its data values ... only to pull them out again to get them into the `PreparedStatement`. It almost seems like it's more trouble than its worth.)

You can employ similar code to remove objects from the database using an SQL `DELETE` statement. In the interest of brevity, we won't show that here.

Let's recap. To work with objects in an RDBMS, you've had to:

- 'Map' the object structure (Java) to a table structure (SQL)
- Create the table using SQL
- Write SQL code 'inside' Java code to create, read, update, and delete objects
- Explicitly instantat objects when reading them from the database, and...
- ...Explicitly manage the translation between SQL datatypes and Java datatypes

In short, you've had to dance between two worlds: the Java world, and the SQL world. (The same would have happened had you written the code in, say, C#.) And this was a simple object. Imagine what things would look like if you were dealing with a complex 'tree' of interrelated objects. You would have to come up with some way to model object references in the RDBMS world.

Thought Experiment, Part 2: Object/Relational Databases

Object/Relational databases can mitigate some of the tedium described in the preceding section. They manage the job of translating between objects and relational tables for you. The details of moving data back-and-forth between object data members (in the program) and table rows (in the database) are handled invisibly for you by code provided by the O/R mapper component of the O/R database.

Note that we said that O/R databases can mitigate *some* of the tedium. Not all. You still have to tell the O/R database which objects are to be made persistent, and how their contents are mapped to the table. How this is done depends on the O/R database you use.

Many O/R databases require that you derive those classes in your application that you want to store in the database from a *base persistent* class. Among other things, the base persistent class typically defines a field that the O/R engine uses to hold a unique identifier for the object.

Virtually all O/R databases require that you create a *descriptor* file. This file tells the O/R mapping engine which classes map to which table, and which insance variables map to which columns. An example descriptor file for our `Employee` class might look like this:

```
<class-descriptor class="Employee"
 table="Employee">
  <field-descriptor id="1"
       name="lastName"
       column="lastName"
       jdbc-type="VARCHAR" />
  <field-descriptor id="2"
       name="firstName"
       column="firstName"
       jdbc-type="VARCHAR" />
  <field-descriptor id="3"
       name="IDNumber"
       column="IDNumber"
       jdbc-type="INTEGER"
       primarykey="true" />
  <field-descriptor id="4"
       name="salary"
       column="salary"
       jdbc-type="DECIMAL" />
</class-descriptor>
```

(The above descriptor file is based on the format used by the Apache Jakarta OJB object-relational mapping tool.)

Often, a *deployment descriptor* file is also required, which includes information such as the database driver class, the alias to the database, username, password, and so on. Typically, this deployment descriptor file is also an XML file.

Because an O/R database appears to the programmer as an Object-Oriented database, querying the database can look much the same as querying a "pure" OO database. Query languages and APIs vary. For example, the OJB relational mapping tool (described above) supports several query APIs, including JDO and ODMG (both standards-based APIs). Continuing with our `Employee` class example, we could use the following code to store a new `Employee` object in the database (using the ODMG API):

```
...
// Instantiate an ODMG implementation,
//  create a new database, and open it
Implmentation odmg = OJB.getInstance();
Database db = odmg.newDatabase();
db.open("default",Database.OPEN_READ_WRITE);

// Start a transaction
Transaction trans = odmg.newTransaction();
trans.begin();

// Instantiate a new Employee object
Employee emp = new Emmployee();
... populate emp object's fields here ...

// Put the object in the database
db.makePersistent(emp);

// Commit the transaction
trans.commit();

...
```

In the above example, we've assumed that the mapping descriptor has already been built and associated with the application, and that the `IDNumber` field was identified as the primary key (as shown in the descriptor file above).

We'll have more to say about interacting with an OO database shortly, but for now notice that a single call to `db.makePersistent()` is all that is required to "store" an `Employee` object into the database. Furthermore -- and this is important -- it would still be a single method call regardless of the size of the object (i.e., how many data members the object had).

Thought Experiment, Part 3: Pure Object Databases

An object database stores objects. That might sound simple ... and, in most cases, it is. This is as compared to a relational database which, as we've said already, stores relations. In practical terms, storing relations means storing data items (in a table) that are related or logically "connected" in some way. So, for example, a person's name is logically connected to that person's address ... so a customer table would store (among other things) the relation between the person's name and address. It would do that by (again, in practical terms) putting those items together in the same row of the same table. We've already shown this.

Meanwhile, in an object database, the relations are inherent in the structure of the objects themselves. And the structure of a given object is, of course, defined by that object's class. Furthermore,

relationships between objects -- which in the RDBMS world would be handled by `JOIN` operations -- are modeled via references.

Let's extend our Employee example a bit to show this. Suppose we added a `Department` class to our application:

```
public class Department
{
  int IDnumber;     // Department ID
  string name;      // Department name
  string location; // Department location
  ...other data members...

  ...Department class methods...
```

Furthermore, we want to be able to determine which employees belong to a given department, as well as which department a given employee is a member of. There is, then, a bidirectional relationship between `Employee` objects and `Department` objects. Were we to model this in a properly normalized RDBMS, we would create 3 tables, as illustrated below:

Employee Table

Last Name	First Name	Empl ID	Salary
Jones	Jeff	0001	30000.00
Smith	Jane	0002	40000.00

Department Table

Dept ID	Name	Location
0010	Sales	New York

Dept_Empl Table

Dept ID	Empl ID
0010	0001
0010	0002

Figure 2. *The database now includes a Department Table. The database also models the bi-directional*

relationship betweenn employee and department using the Dept_Empl Table. Queries into this latter table can quickly determine which employees are in a given department, as well as which department a given employee belongs to. Basically, this table connects a foreign key from the Department Table with a foreign key from the Employee Table.

This is nice, but notice how we're multiplying entities in order to "help" the RDBMS world model relationships that are expressed succintly in the OO world. The connection between employee and department can be handled by adding a `Department` object reference into the `Employee` class, and adding an array of `Employee` object references into the `Department` class. We can represent this graphically as follows:

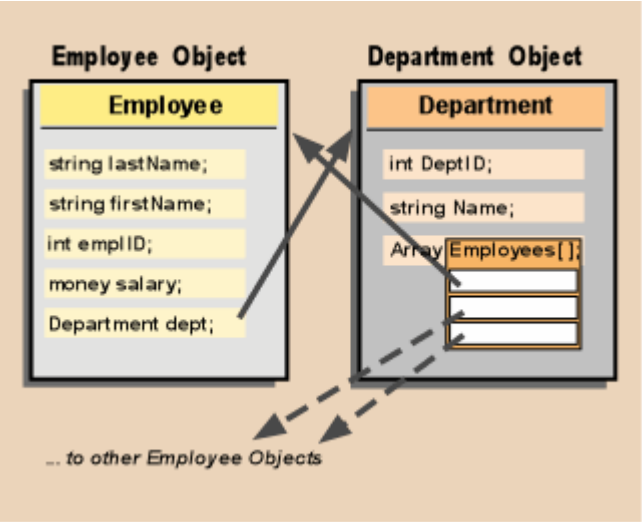

Figure 3. *In an OO database, relationships can be "expressed directly." There is no need to model them with (as in the preceding example) an intermediate table.*

Now, this scenario sounds appealing ... *provided* that manipulating objects with an OO database allows us to (a) store and retrieve whole objects, and (b) preserves object relationships. Happily, with virtually all professional OO databases, both are true. This leads to a sort of principle of OO databases:

The class structure is the schema.

In other words, associations and relationships that have to be "manuall constructed" for a relational database, are already built into the class architecture. Given that most programming these days is done in one OO language or another, then, we can see that using an ODBMS sidesteps a lot of work that has to be done to support an RDBMS. Specifically, with an RDBMS, you have to build a schema that mimics the class structure. With an ODBMS, this is unnecessary. (And that work is NOT including the extra conversion code that we illustrated earlier. Additionally, in many cases, you don't have to build descriptor files, either.)

Persistence

OO database systems provide two general persistence techniques: *explicit* persistence and *transparent* persistence. With explicit persistence, operations on the database -- storing objects, retrieving objects, deleting objects, etc. -- are explicitly expressed in code. This is the sort employed by the popular open-

source OO database, db4o.

For example, to put an `Employee` object into a db4o database, the following Java code suffices:

```
// Open the database
ObjectContainer employeeDB = db4o.openFile(
 "database path");

Employee emp = new Employee();
...populate emp object's data fields...

// Store the employee in the database
employeeDB.set(emp);

// A transaction is automatically started when
//  the ObjectContainer (database) is opened.
//  Before closing the database, commit the
//  transaction.
employeeDB.commit();
employeeDB.close();
...
```

We can fetch an `Employee` object from the database with equal ease. db4o uses a query-by-example query technique, which requires that we simply build a template object with the data elements we want matched filled with target values. Data elements that are not to participate in the match, are filled with either `null` or zero.

So, assuming we have an `Employee` object called `empTemplate`, and that the `IDNumber` field is set to 1 ... while all the string fields are set to `null`, then we can fetch employee #001 from the database with:

```
ObjectSet result = employeeDB.get(empTemplate);

// Get the retrieved employee object
if(result.hasNext())
{    Employee emp = (Employee)result.next();
     ...do something with the emp object...
}
...
```

And, if we wanted to delete the employee #1, in the code snipped above where you see the line "*...do something with the emp object...*", simply substitute:

```
employeeDB.delete(emp);
```

In short, you work with objects as whole, unified, entities ... rather than rows of values.

With **transparent persistence**, objects are moved to and from the database invisibly. Reference an object, and object data is read from the database, a new object is instantiated, and its contents are populated with the data. Modify the object's content, and the object is marked as "dirty". At some point (usually marked by a call to a `commit()` method), the database is updated with the modified object's contents.

For example, using the JDO API for object persistence, assume we want to update employee #1's salary. We could use the following:

```
...
pmanager = factory.getPersistenceManager();
pmanager.currentTransaction().begin;

// Fetch an employee object from the database
Query query = pmanager.newQuery(
     Employee.class, "IDNumber == 1");
Collection result = (Collection)
 query.execute();
Employee emp =
 (Employee)result.iterator().next();

emp.salary = Money.fromInt(30000);

pmanager.currentTransaction().commit();
...
```

(The above code assumes that we have a `fromInt()` method that converts an `int` value to a `money` value.) The persistence is transparent in that we need not tell the database engine when to store the modified object. Any changes to any persistent objects within the context of the transaction are written to the database when `commit()` is called. In fact, we could have modified all sorts of persistent objects within that transaction ... it would not have mattered. Any updates would be recorded to the database at the conclusion of the transaction.

Conclusion

We have compared ODBMS and RDBMS systems at the fundamental levels. Hopefully, we've provided a framework that you can use to compare the two ... in those areas where comparisons make any sense.

Of course, there are characteristics and behaviors of ODBMSes and RDBMSes that we have either glossed over or left out entirely: things such as indexes, in-depth queries, optimizations, transactions, and more. Additional materials on this website will, we hope, cover all those issues and more. As those materials become available, we will be careful to provide links to them, so that this page on "basics" will serve as an evolving "jump-off" point.

Rick Grehan is a QA Engineer at Compuware/Numega labs, where he has worked on Java and .NET projects.

He is also a contributing editor for InfoWorld Magazine. His work has appeared in Embedded Systems Programming, EDN, The Microprocessor Report, and Computer Design. Before coming to Compuware, Rick was on the Discover DSP Project at Metrowerks, Inc.

Earlier, Rick was a Senior Editor at BYTE Magazine, where he was the Lab Director, and authored BYTE's JavaTalk column.