IDEA GROUP PUBLISHING
701 E. Chocolate Avenue, Suite 200, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4834

Basic Impedance Mismatch Problem Resolution

Erick D. Slazinski and Michael J. Payne
Computer Information Systems and Technology Department
Purdue University 1421 Knoy Hall West Lafayette, IN 47907-1421
Tel: (765) 496-7582, Tel: (765) 494-2566
Fax: (765) 496-1212, Fax: (765) 496-1212
edslazinski@tech.purdue.edu, mjpayne@tech.purdue.edu

ABSTRACT

Many object-oriented software developers are faced with the dilemma of utilizing a relational database for their persistent data store. The phrase "impedance mismatch" is often used to characterize the difficulties in sharing data between the relational and object-oriented models. By harnessing the power of modern Relational Database Management Systems (RDBMS), an easy-to-implement, easy-to-use, persistent storage solution for Java that does not compromise the data integrity of either model will be illustrated. The technologies used in this paper are Java and Oracle.

INTRODUCTION

Impedance mismatch arises from the inherent lack of affinity between the object and relational models. Problems associated with the impedance mismatch include class hierarchies binding to relational schemas (InterSystems Cache, 2002)

Why does the impedance mismatch exist? Object-Oriented Database Management Systems (OODBMS) have been around for several years and are now quite mature – yet they have failed to expand beyond a niche market. (Leavitt, 2000) Several theories have been proposed to explain this phenomenon and are beyond the scope of this paper. This is an acknowledged fact that forces object-oriented (OO) developers to store persistent data objects in a relational structure. A basic understanding of Relational Theory, OO theory and RDBMS implementation is assumed.

To overcome the impedance mismatch, some associations between the relational model and the OO model should be mentioned. As seen in Table 1, there are many similarities between the models, which can be exploited. For the OO model, there are classes (abstract and concrete), inheritance hierarchies, interfaces, and instances of classes. In the Relational model, there are entities and relationships. In the relational model, tables, constraints (including keys) and data exist.

For this paper, the focus will be on a basic, single inheritance, OO structures. Advanced structures such as multi-inheritance and wholly contained object lists will be addressed in a later article. A modern RDBMS, such as those developed by Oracle, IBM, MS and others is required for implementing this technique. Additional mappings, such as class methods to stored modules exist, are also beyond the scope of this work.

Table 1

OO	Relational	RDBMS
Abstract Class	Entity	Table definition
Concrete Class	Entity	Table Definition
Inheritance hierarchy	Relationships	Primary / foreign key constraints
Interface	N/A	Stored module
Instance	N/A	Data row

One of the problems demonstrated in the above table is that there is only a single construct in the relational and RDBMS paradigm for two concepts in the OO paradigm. This, the authors believe, is the heart of the impedance mismatch. While the OO developers develop abstract and concrete classes to solve specific application problems, the concept of a structure that contains no data does not exist in the relational world.

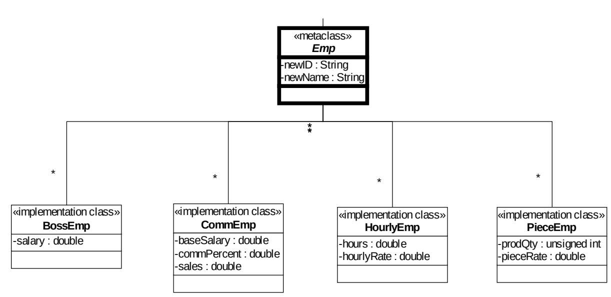
The techniques used for our examples will be written in Java and the RDBMS will be Oracle. Both technologies were chosen for their popularity in the marketplace. In order to minimize the impact of changes in either the database table structures or the java class codes, developers should have (where possible) a single point of entry (class method) to get a single item stored in the database. It is assumed that object ID's are not required to be consistent across application runs. The addition of object ID's (as surrogate keys) to the methods described here is a trivial task and left up to the reader.

Figure 1 illustrates the basic inheritance hierarchy that has a requirement for persistence of data.

APPROACH 1

An approach that is often employed when storing OO data in a RDBMS is the creation of database table for every concrete class in the system. This approach is labeled Approach 1a in Figure 2. The problem with this approach is that the rules of normalization are not satisfied. Normalization is the process, which produces an efficient data storage structure without data duplication. While the data is stored in a fashion, which is very close to the OO structure, changes to the database are expensive. Altering the EMP class would require the modification of four relational tables! The corresponding Java code is straight forward, but suffers from the same fragility as the underlying table structures.

Figure 1



```
public boolean readBoss_Emp(String employeeID) {

        Statement statement;
        ResultSet resultSet;
                int i = 0;

                String query = "SELECT * FROM Boss_Emp where
                NewID = " + "\'" + employeeID + "\'";
                statement = connection.createStatement();
                resultSet = statement.executeQuery(query);
                while(resultSet.next()) {
                        empID = resultSet.getString("NewID");
                        empName = resultSet.getString("NewName");
                        empSalary = resultSet.getDouble("Salaruy");
                i++;
        }
}

public boolean addEmployee(String employeeID, String newName,
Double salary) {
                {
                Statement statement = connection.createStatement();
                String query = "INSERT INTO BossEmp (" +
                        "NewID, NewName, Salary" + ") VALUES ('" +
                        employeeID + "', '" +
                newName + "' , " + salary + ")";

        int result = statement.executeUpdate(query);
}
```

A variation to Approach 1 has the development of 1 table, which contains all columns for all of the concrete classes. This approach is labeled Approach 1b in Figure 2. It is difficult to enforce data element requirements (such as for a given BossEmp, you must have a salary vs. an HourlyEmp requires hours and hourlyrate filled in). Another problem with this approach is that there will be wasted data space due to empty data cells for each row of data stored. Again maintenance is an issue, if another concrete class is added to the hierarchy; additional columns must be added to the database table. The problem extends into the Java code, not only does is the code required containing extraneous non-values, but maintenance is overly complex. Remember that these routines must be maintained for all concrete classes!

```
public boolean readBossEmp(String employeeID) {

        Statement statement;
        ResultSet resultSet;
        int i = 0;

        String query = "SELECT * FROM Emp where NewID = " + "\'"
                + employeeID + "\'" + " and EmpType = " + "\'" + "B"
                + "\'";
        statement = connection.createStatement();
        resultSet = statement.executeQuery(query);
        while(resultSet.next()) {
                empID = resultSet.getString("NewID");
                empName = resultSet.getString("Name");
                empSalary = resultSet.getDouble("Salary");
                i++;
        }
        resultSet.close();
        statement.close();
}

public boolean addEmp(String employeeID, String newName,
Double salary) {

        boolean OK = false;
        String empType = "B";
        Statement statement = connection.createStatement();
        String query = "INSERT INTO BossEmp (" +
                "NewID, NewName, EmpType, Salary, BaseSalary,
                CommPercent, Sales, " + "Hours, HourlyRate, ProdQty,
                PieceRate)" + "VALUES ('" + employeeID +"', '" +
                newName + "', '" + empType + "' ," + salary + ", " + 0
                + ", " + 0 + ", " + 0 + ", " + 0 + " ," + 0 + ", " + 0 + ",
                " + 0 + ")";

        int result = statement.executeUpdate(query2);     }
```

APPROACH 2

This approach is labeled Approach 2 in Figure 2 satisfies the rules of normalization. This structure is also very similar to the object model. This approach is not without its challenges. The storage of data is now spread across multiple tables – which could be seen as destroying the cohesiveness of the data. Secondly, data is stored in the relational manifestation of an abstract class – something that is not allowed in an OO language! This issue is an implementation (RDBMS) dependant issue and not something that can be addressed. In order to provide access to the fragmented data in a manner consistent with the hierarchical model presented, database views will be used

Database views, while a terrific mechanism for retrieving data from a database, they may encounter problems when the user tries to insert or update data through the view. These anomalies often restrict their usage. However, database vendors such as Oracle have given the database developer a mechanism to help the end-user get more mileage out of views – instead-of triggers. Like all other triggers, instead-of trigger can be defined to fire on any DML statement where the view is the target object. Thus the instead-of trigger will intercept the action and for our application, apply the DML statement on the correct underlying tables. (Slazinski, 2001)

To ensure data integrity, all permissions should be removed from the underlying tables and only allow users to access those database views that represent concrete classes in the system – thus giving access to whole instances, instead of partial instances that are spread across multiple database tables.

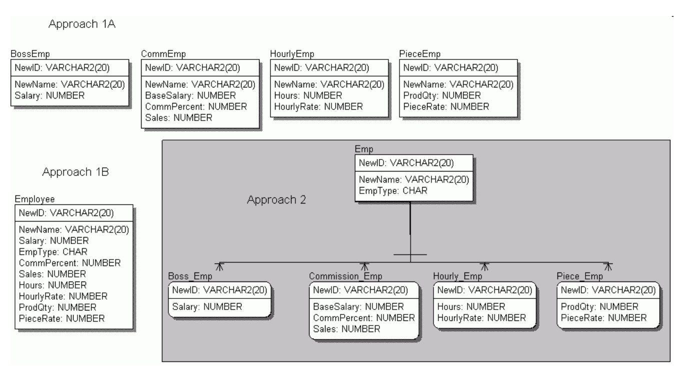
IMPLEMENTATION

With the theory is explained, a comprehensive example that builds upon the object model shown in Figure 1 follows.

Step 1

Convert your class diagram into relational tables. The resultant DDL is shown as Approach 2 in figure 2. It should be noted that multiple inheritance can also be addressed using these techniques – simply select the appropriate attributes (as opposed to all attributes) from the parent objects into the view definition and continue.

Figure 2



Step 2

Create database views that represent the concrete classes in our diagram. The resultant DDL will look like this.

```
create view BOSSEMP as
select EMP.NEWID as NEWID, EMP.NEWNAME as NEWNAME,
SALARY, PERCENT
from EMP, BOSS_EMP where
EMP.NEW_ID = BOSS_EMP.NEW_ID;

create view COMMEMP as
select EMP.NEWID as ID, EMP.NEWNAME as NAME,
BASESALARY, COMMPERCENT, SALES, PERCENT
from EMP, COMMISSION_EMP where
EMP.NEW_ID = COMMISSION_EMP.NEW_ID;
```

Step 3

Create instead-of triggers for all allowed operations (insert, update and delete). Even though our implementation used the instead-of trigger construct of Oracle, a set of stored procedures associated with each concrete class / view combination would have worked as well.

```
create trigger I_BOSSEMP_TRIG
i     instead of INSERT on BOSSEMP
for each row
declare
      EMP_ID number := EMP_ID.NEXTVAL;
begin
      insert into EMP values (EMP_ID, :new.NAME);

      insert into BOSS_EMP values (EMP_ID, :new.SALARY,
      :new.PERCENT);
end;
```

Step 4

Now the developers can code in a natural fashion as shown below.

```
public boolean readBossEmp(String employeeID) {

      Statement statement;
            ResultSet resultSet;
            int i = 0;
            String query = "SELECT * FROM BossEmp where NewID
                  = " + "\'" + employeeID + "\'";
            statement = connection.createStatement();
            resultSet = statement.executeQuery(query);
            while(resultSet.next()) {
                  empID  = resultSet.getString("NewID");
                  empName = resultSet.getString("Name");
                  empSalary = resultSet.getDouble("Salary");
                  i++;   }
            resultSet.close();
            statement.close();         }

public boolean addBossEmp(String employeeID, String newName,
      Double salary, Double percent) {

      boolean OK = false;
      Statement statement = connection.createStatement();
      String query = "INSERT INTO BossEmp (NewID, NewName,
      Salary) VALUES ('" + employeeID + "', '" + newName + "', "
      + salary + ")";

int result = statement.executeUpdate(query2);    }
```

CONCLUSIONS

Even though the examples used a basic class hierarchy, the theory holds. Additional, follow-on research will be conducted to explore advanced class definitions – including embedded classes, multi-inheritance, etc.

REFERENCES

Oscillating Between Objects and Relational: The Impedance Mismatch, InterSystems Cache. Available at http://www.e-dbms.com/cache/whitepapers/impedance.html (Date of access January 8, 2003)

Leavitt, Neal. Whatever Happened to Object-Oriented Databases?, Computer, August 2000. Available at http://www.leavcom.com/db_08_00.htm (Date of access January 8, 2003)

Slazinski, E. D. (2001). Views - the 'other' database object. Proceedings of the ISECON 2001 18th Annual Information Systems Education Conference, 33.