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.

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.

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

APPRAOCH 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 straightforward, but suffers from the same fragility as the underlying table structures.

Figure 1

Table 1

<table>
<thead>
<tr>
<th>OO</th>
<th>Relational</th>
<th>RDBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Class</td>
<td>Entity</td>
<td>Table Definition</td>
</tr>
<tr>
<td>Concrete Class</td>
<td>Entity</td>
<td>Table Definition</td>
</tr>
<tr>
<td>Inheritance hierarchy</td>
<td>Relationships</td>
<td>Primary/foreign key constraints</td>
</tr>
<tr>
<td>Interface</td>
<td>N/A</td>
<td>Stored module</td>
</tr>
<tr>
<td>Instance</td>
<td>N/A</td>
<td>Data row</td>
</tr>
</tbody>
</table>

Copyright © 2003, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.
public boolean readBoss_Emp(String employeeID) {
    Statement statement;
    ResultSet resultSet;
    int i = 0;
    String query = "SELECT * FROM Emp 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 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("Name");
        empSalary = resultSet.getDouble("Salary");
        i++;
    }
    resultSet.close();
    statement.close();
}

public boolean addEmp(String employeeID, String newName, Double salary) {
    String query = "INSERT INTO BossEmp (" + "NewID, NewName, Salary" + ") VALUES (" + employeeID + ", " + newName + ", " + salary + ");"
    int result = statement.executeUpdate(query);
}

public boolean addEmp(String employeeID, String newName, Double salary) {
    String query = "INSERT INTO BossEmp (" + "NewID, NewName, Salary" + ") VALUES (" + employeeID + ", " + newName + ", " + salary + ");"
    int result = statement.executeUpdate(query);
}

public boolean addEmployee(String employeeID, String newName, Double salary) {
    String query = "INSERT INTO BossEmp (" + "NewID, NewName, Salary" + ") VALUES (" + employeeID + ", " + newName + ", " + salary + ");"
    int result = statement.executeUpdate(query);
}

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("Name");
        empSalary = resultSet.getDouble("Salary");
        i++;
    }
    resultSet.close();
    statement.close();
}

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("Name");
        empSalary = resultSet.getDouble("Salary");
        i++;
    }
    resultSet.close();
    statement.close();
}

public boolean addEmp(String employeeID, String newName, Double salary) {
    String query = "INSERT INTO BossEmp (" + "NewID, NewName, Salary" + ") VALUES (" + employeeID + ", " + newName + ", " + salary + ");"
    int result = statement.executeUpdate(query);
}

public boolean addEmployee(String employeeID, String newName, Double salary) {
    String query = "INSERT INTO BossEmp (" + "NewID, NewName, Salary" + ") VALUES (" + employeeID + ", " + newName + ", " + salary + ");"
    int result = statement.executeUpdate(query);
}

public boolean readBoss_Emp(String employeeID) {
    Statement statement;
    ResultSet resultSet;
    int i = 0;
    String query = "SELECT * FROM Emp 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 readBoss_Emp(String employeeID) {
    Statement statement;
    ResultSet resultSet;
    int i = 0;
    String query = "SELECT * FROM Emp 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 addEmp(String employeeID, String newName, Double salary) {
    String query = "INSERT INTO BossEmp (" + "NewID, NewName, Salary" + ") VALUES (" + employeeID + ", " + newName + ", " + salary + ");"
    int result = statement.executeUpdate(query);
}

public boolean addEmployee(String employeeID, String newName, Double salary) {
    String query = "INSERT INTO BossEmp (" + "NewID, NewName, Salary" + ") VALUES (" + employeeID + ", " + newName + ", " + salary + ");"
    int result = statement.executeUpdate(query);
}
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
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.

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
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)
Related Content

Computer Assisted Problem Solving Environment (PSE)
www.irma-international.org/chapter/computer-assisted-problem-solving-environment-pse/112522

Design of a Migrating Crawler Based on a Novel URL Scheduling Mechanism using AHP
www.irma-international.org/article/design-of-a-migrating-crawler-based-on-a-novel-url-scheduling-mechanism-using-ahp/169176

A Comparative Analysis of a Novel Anomaly Detection Algorithm with Neural Networks

Conducting Congruent, Ethical Qualitative Research on Internet-Mediated Research Environments
www.irma-international.org/chapter/conducting-congruent-ethical-qualitative-research/28293

Some Portions of Dooyeweerd’s Positive Philosophy
www.irma-international.org/chapter/some-portions-dooyeweerd-positive-philosophy/28081