Object Identifiers for Relational Legacy Data

Prepared for discussion on the Internet

Wolfgang Keller

sd&m München Februar 1995

sd&m GmbH & Co. KG
Finanz-Informationssysteme
Thomas-Dehler-Straße 18
81737 München
Telefon (089) 6 27 02 - 0
Telefax (089) 6 27 02 50

Table of Contents

1 Abstract

This paper describes problems that you will be confronted with, when trying to build an Object Oriented Data Access Layer for relational legacy data without any alterations of old tables - the relational data reengineering problem[1]. This results in very expensive solutions for OID (Object Identifier) handling.

We present our current solution, the solutions we have abandoned in a first discussion and we invite others to join our discussion with the purpose to find an elegant solution for the handling of relational legacy data in OODBMS without breaking ODMG'93 user code patterns.

This paper does not discuss the overal architecture of an access layer - it only discusses the special problems around object identity.

If you have comments or ideas, please send email to `wk@sdm.de` or `100655.566@compuserve.com`

2 The Problem

Our goal is to offer an Object Oriented Access Layer for relational data. The environment we develop our solution for is a large bank with thousands of legacy applications and thousands of relational legacy tables. We have decided to use an access layer instead of an OODBMS because we have to support about 80% old relational legacy data and only about 20% new data in the new applications, we are going to develop. We also have to support access to data in systems like IMS and TABEX.

Our special situation (DB2, IMS, OS/2, MVS, 3 Layer C/S) does not allow us to use Persistence, Objectivity or ONTOS as frontends for various reasons:

- Persistence does not yet support DB2.
- ONTOS and Objectivity have hooks for replacing their storage management by custom routines but no way of dealing with transaction monitors like IMS/DC - we have to access our data through IMS/DC.

---

[1] see also [Pre 94]

- We don’t know if anyone of them has a reengineering concept that works without changing tables and allows coexistence of old and new applications.

A central concept in OODBMS and also in OO access layers is **strong Object Identity**. It is typically represented by OIDs (Object Identifiers). These are typically dumb numbers that will accompany an object from birth to grave and will be used to constitute links (relations) between objects.

Normal solutions for reengineering legacy relational data with OO systems will therefore insert an OID into tables and work with it.

We are trying to find a solution without alteration of any tables, because:

- Legacy applications are still allowed to update the data we are using - they do not know anything about an OID.

- Legacy applications must not be changed - therefore only solutions that need no change in any old application code are allowed.

- Tables should not be altered - inserting a timestamp is the most we are allowed to do.

- Relational data, posessed by new object oriented applications, should still be accessible by old-style applications. We are confronted with a longer process of transforming a large software development organisation.

We have a solution for this but we don’t like it for several reasons:

- It is complex and not easy to understand by application programmers.

- It is expensive in terms of code needed.

- It conflicts with ODMG [ODMG 93] conforming user code - which we would like to imitate as good as possible. The solution brings application know how to the surface that we would rather hide.

This is why we would like to discuss our state of the art in public - maybe we can start an exchange who might have a more elegant solution than our’s.

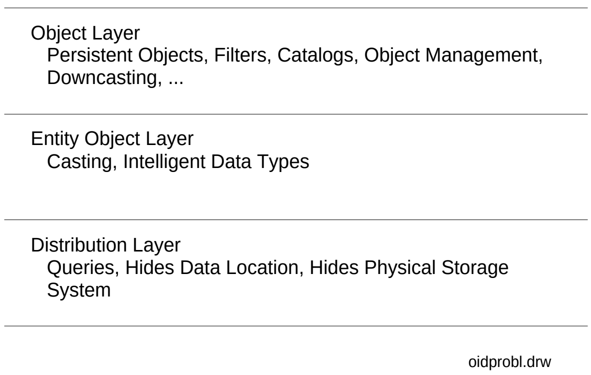
3 Architecture

To facilitate the following discussion, we will briefly describe our overall architecture for an OO Data Access layer (see also Picture 1):

- The Object Layer contains persistent objects from a common protocol class PObject. It also contains an object manager, transaction objects, the database object and some additional classes like filters and catalogs.

- The Entity Object Layer contains Objects that reflect the physical relations but use intelligent data type classes for base types and structures.

- The Distribution Layer is a set of abstract queries that are used to hide the physical location of data an the database system used (IMS, DB2, TABEX). Queries are static (with respect to DB2 and others) and are selected by a query manager at runtime.

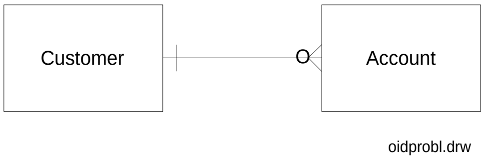


Picture 1: Architecture

4 Our Solution

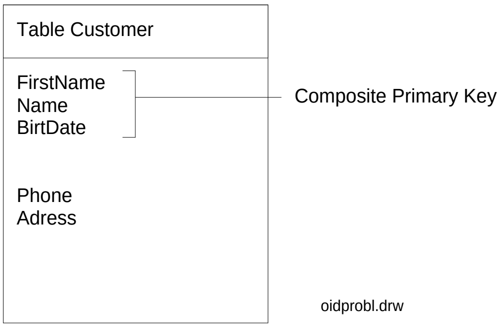
4.1 Running Example

As a running example we will use a customer -[1:n]->> account relationship. Both objects have legacy tables that are accessed by legacy application that cannot be changed.



Picture 2: Customer/Account Relationship

We use a relationship in our example, as problems will not arise before you have at least a 1:n or n:m relationship somewhere.



Picture 3: Customer Relation

In our example we use a composite primary key (see Picture 3) instead of an artificial key to make the problem a bit harder.

4.2 The ODMG Solution

ODMG'93 uses hidden object identifiers plus names. Code to add a new customer and to give an account to that customer could look as follows:

```
(1)    Ref<Customer> MyCustomer;
(2)    MyCustomer = new Customer('John','Doe','69/6/13',
                                 '089-8714228','Hillstreet 15')
(3)    Ref<Account> MyAccount;
(4)    MyAccount = new Account(BranchOffice, AccountMode);

(5)    MyCustomer.AddAccount(MyAccount);
```

Statement (1) declares an undefined smart pointer. Statement (2) constructs a new customer object - an oid is assigned automatically. The Customer constructor does not need to know anything about primary key attributes.

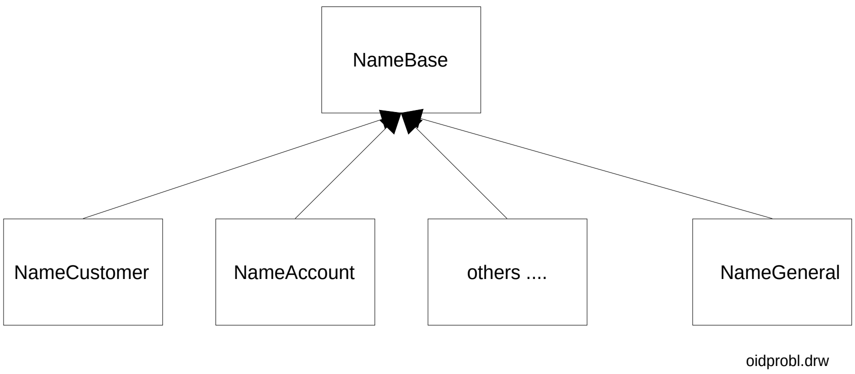
Statement (4) constructs a new Account - in a relational system it would be normal to assign it some kind of artificial account number - in an object oriented system this is not necessary.

Finally statement (5) inserts the new account into the set of accounts a Customer holds.

With ODMG’93 you can also assign names to objects and retrieve those objects by name (See [ODMG’93 pp 112 - 119]).

4.3 Our Current Solution

As stated above we are not allowed to insert a new dumb OID into existing tables. Instead we have to live with what we find in the tables. We will build a hyrarchy of different OID Classes (we call them Name Classes - all derived from a NameBase Class (see Picture 4). There is one NameGeneral Class that is used for the forward engineering cases. In case of forward engineering we can insert a dumb OID into the tables we define ourselves.



Picture 4: OID Class Hyrarchy

Names are the access layers equivalent to primary keys. The derived name classes for legacy data will typically contain the underlying objects primary key attributes.

4.3.1 User Code Example

The code creating objects looks like the ODMG code above. :

```
(1)    Ref<Customer> MyCustomer;
(2)    MyCustomer = new Customer(‘John’,’Doe’,’69/6/13’,
                                 ’089-8714228’,’Hillstreet 15’)
```

```
(3)    Ref<Account> MyAccount;
(4)    MyAccount = new Account(BranchOffice, AccountMode);

(5)    MyCustomer.AddAccount(MyAccount);
```

The name classes will not appear here - they are constructed by the constructors of Customer and Account. The name classes will appear, if we want to lookup an object :

```
(6)    Ref<Customer> =
          Customer::lookup(NameCustomer('John','Doe','69/6/13'));
```

or when a constructor wants to check the existence of an object

```
(7)    Customer::exists(NameCustomer('John','Doe','69/6/13'));
```

4.3.2 Need for Number Servers

If you create e.g. an account in the relational world you have business rules that tell you, how the account number has to be formed.

```
(4)    MyAccount = new Account(BranchOffice, AccountMode);
```

In our example, the account number is formed using the branch office and some kind of account mode. It has to be reserved in the database by a number server. As you have different primary keys that might all contain application logic you might end up with as many number servers as you have application classes.

4.4 Why we do not like the above solution

4.4.1 Complexity

For each application class you have

- o the aplication class itself
- o a name class

- a number server that gets you numbers from the relational data model
- a filter and a catalog class (not described in this paper - surrogate for OQL)

The user of the access layer can generate many things but has to write lots of the code for the reengineering szenario by hand - that’s not the clean and simple solution we would like to have.

4.4.2 Application Know How not Separated from Object Database

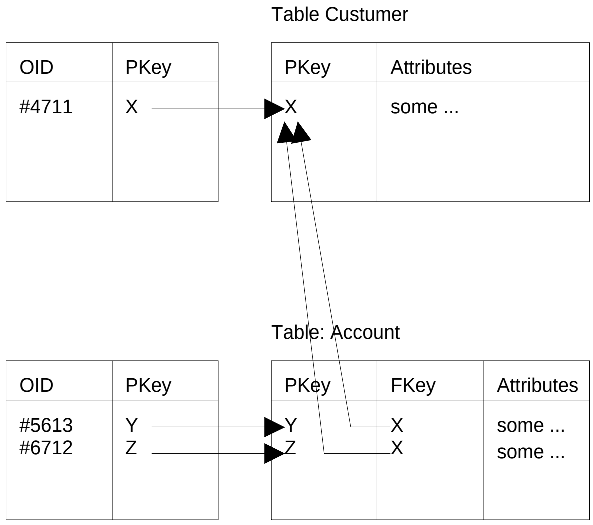
In our examples the constructor of an application class has to care for the correct instantiation and creation of the name class - which is the relational equivalent of the oid. Thus the relational term of a „primary key“ makes his way thorugh all the layers of the object oriented access layer. Thats a serious design flaw.

5 Solutions that will not work

This chapter contains some solutions we have discussed so far but considered not feasible.

5.1 Separate OID Tables

One spontaneous idea is to add a second table for each legacy table, containing an OID to primary key mapping. This works fine if data are no longer updated in legacy aplications, it still works, if you are working with one table only (which is both unrealisitic)



Picture 5: Separate OID Tables

First it doesn't work when you have concurrent online updates from old and new applications requiring good performance.

Picture 5 shows the principle of added OID-Tables. One OID table for all relations is equivalent to this model. Update and insert triggers cannot handle the OID because it has to be consistent over several tables due to relations. Therefore this solutions doesn't work without changing old application code. You could then as well insert an OID into the table - this is also equivalent to this solution.

5.2 OODBMS as Proxy

Another idea is to have an OODBMS to store new objects and to use the storage manager hooks for old, reengineered relational data. This solution has several drawbacks:

- new data are no longer accessible for systems that are developed the old way.
- It is not very likely the storage manager will cooperate correctly with IMS/DC resulting in proper transactions.
- we are still left alone with the oid reengineering problem .
- we have no idea whether the whole method works without table alterations.

All in all the solution does not look attractive.

5.3 Inserting OIDs into Tables

As said before - we cannot insert OIDs into tables as we are not allowed to change any legacy application code.

6 References

[ODMG 93] Rick G. G. Cattell (Ed.) et. al.: *Object Database Standard (ODMG 93);* Morgan Kaufmann Publishers, 1993.

[Pre 94] William Premerlani, Michal R. Blaha: *An approach for reverse engineering of relational databases;* Communications of the ACM, May 1994, p42(9).