Is an object layer on a relational database more attractive than an object database ?

Franck Lebastard
CERMICS/INRIA - BP 93
F-06902 Sophia-Antipolis Cedex, France
Voice +33 93 65 77 42
E-mail : Franck.Lebastard@sophia.inria.fr

1 Introduction

As researchers in Artificial Intelligence, our first aim was to allow our expert system shell Smeci [Sme90] the access to relational databases during reasoning. We also needed to save in a database complex objects that seemed interesting for further utilization, in particular the knowledge bases and the results of reasoning.

To this end, we have defined generic correspondences [Leb93] between relational concepts and some of the object concepts that are common to most object models. These correspondences allow to translate relational data into complex objects and conversely. They generalize the mapping proposals that we found in the literature [Lee90, WBL+91, KJA93].

An implementation of these definitions has been realized. This is the Driver system [Leb92] whose specificity is to define an object oriented DBMS (OODBMS) on a relational DBMS (RDBMS).

In Driver, a correspondence scheme must describe how to use a particular relational database that is, the object representation and the relational representation to bring together and the concrete mapping between them. It can be given by the user or automatically generated. The database is then available as an object oriented database.

Driver can be used with many object models. The system performs all operations on the objects through a functional interface that must be instanciated for the chosen model. In particular, it creates objects in memory and reads and writes their slots through this interface. This way to handle objects ensures that persistency is a property effectively orthogonal to the model. Of course, only selected object concepts can become persistent. The other properties are simply ignored.

Driver is operational and is used by several industrial partners.

2 Our generic correspondences

Let us see now the generic correspondences we have defined. They make possible to manage relational data in the form of complex objects and to express objects as relational data.

2.1 Classes and relational tables

We have associated the concept of relational table with the concept of class. More precisely, we have associated tables with class hierarchies because we enforce all subclasses of a class to be mapped on the same table. The most general class mapped to a table is called a *main class*. Its associated table is its *main table*.

We also allow to associate more than one table to a class. The extra tables are called the *secondary tables* of the class. The main table and the secondary tables are the *elementary tables* of the class. They must imperatively be linked all together with joins which are also called elementary, that means in our definition that each one binds one tuple of a table with one tuple of another. The elementary tables of a class are utilized to map its fields. In a hierarchy, any class may use one or several additional elementary tables to store specific data.

The figure 1 shows a possible mapping for the `Employee` class. `emp` is its main table and `person` is a secondary table, both are its elementary tables.

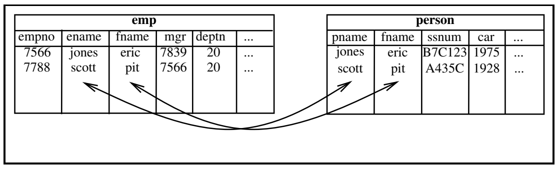

Figure 1: The elementary tables of the Employee class

We can notice that there is always a path, a join *chain* linking any elementary table to the main table.

2.2 Objects and tuples

As we have brought together both notions of relational table and class we also associate the concept of relational tuple with the concept of object. Both are data, occurrences of their own structures. In DRIVER, the correspondence of an object is a set of tuples, one for each elementary table of the class the object is instance of. We have chosen to compose its unique reference, its "oid" using the name of its class main table and its tuple key in this table. This way, every tuple in the main table is candidate to be the correspondence of an object of the associated main class.

The object candidate is an object when :

- all the elementary tables of the main class contain a tuple for it. These tuples are found by joining the main table.
- its value is compatible with the constraints that any instance of this class must check.

These constraints can be set on the atomic fields of the class or on the attributes of the associated elementary tables. They define a kind of filter that tells which tuples correspond to objects. Those which are not selected are simply ignored and everything works at the object level as if they don't exist. The potential objects that corresponds to the selected tuples are calles the *relational objects*.

In figure 1, both tuples "`jones`" and "`scott`" of the `emp` table are object candidates for the class `Employee`. Since the attribute `emp.empno` is the key of the table `emp`, their "oids" are for example `emp/7566` and `emp/7788`. Let us assume that a constraint "`emp.empno > 7000`" is enforced to any tuple of `emp` to be considered as a relational object of the class `Employee`. Since our both tuples comply this constraint and since it exists for each of them in the table `person` a tuple found back by the elementary join, they are considered as `Employee` relational objects, liable to be filtered by an object request involving `Employees`.

When an elementary table set is associated with a class hierarchy and when a main table tuple has been selected as an object of the associated main class, one infers the precise class it is instance of from its implementation in the elementary tables and from the constraints defined for each class it complies or not. Indeed, a class is different from its superclass, over and above its possible own fields that complete those it inherits from the superclass :

- by the possible use of new elementary tables in its correspondence. If the object is at least instance of this class, there must be a tuple for it in each of the elementary tables of the class.
- by stronger constraints enforced on its instances. These constraints must also define an object set that is disjoint from the sets defined by the other subclasses of the same superclass. Since the constraint sets are organized in a tree, to be instance of a class depends on their satisfaction along the considered hierarchy.

While a filtering in the database, DRIVER automatically classifies the chosen relational objects and gives them the most precise class depending on their values and implementation in the base.

Before ending the description of our correspondences at the class and object level, let us point out that the term "table" we have used up to here actually represents more a logical table than the "table as a structure in the database". In other words, in DRIVER, for our correspondences, we can define as many logical tables as we need on a same user table of the base. These logical tables allow us for example to map an object in different tuples of the same user table. They also allow to map independent class hierarchies on logically different main tables that actually represent the same user table in the base. There are indeed applications where one wants to supply independent classes with persistency in a unique table.

file				
tnum	objnum	fnum	fval1	fval2
824	7566			employee
825	7566	1		jones
826	7566	2		eric
827	7566	3	7839	
828	7566	4	20	
...	...	...	...	...
841	7788			employee
842	7788	1		scott
843	7788	2		pit
844	7788	3	7566	
844	7788	4	20	
...	...	...	...	

Figure 2: The file table

For example, let us consider the table `file` presented figure 2. In DRIVER, we can associate our class `Employee` with this table `file` as easily as we did with `emp` et `person`. To have access to the different tuples that make up the relational object, one only has to define elementary joins between for example `mainfile(file)`, the main table of `Employee`, and `file1(file)`, `file2(file)`, etc, defined as secondary tables. Here is an example of an elementary join that allows to have access to the names of the `Employees` :

```
mainfile.objnum=file1.objnum
      and file1.fnum=1
```

We must also set a constraint that precises which tuples of `mainfile` correspond to objects. In our example, this constraint can be :

```
mainfile.fnum is null
and mainfile.fval2='employee'
```

With this correspondence of the class `Employee`, the "oids" of our two relational objects `jones` and `scott` are this time `mainfile/824` and `mainfile/841`.

This way to store all the objects in a unique table is not so odd since it is effectively used, for example in the OODBMS *MATISSE* [Int92].

2.3 Relational correspondences of the field types

Correspondence of the atomic types

We have associated atomic type fields with attributes. This way, any attribute of a class elementary table can be used as the correspondence of any of the atomic fields of this class and vice versa.

If we consider our association `Employee - (emp, person)` again, and if the class has an atomic field `social-security-number`, its mapping can be `person.ssnum`.

Correspondence of the object type

An object field represents an oriented link from an object to another. We associate this link between classes with a relational link, more precisely with a join between one of the elementary tables of the first class and the main table of the second class. This join that we call an *object join* must be an equijoin that compares elementary attributes of the first —attributes called *referential attributes*— and the attributes composing the key of the second.

This restriction allows to know the value of an object field just by knowing the values of the corresponding referential attributes. An object field is empty if any of the referential attributes is containing a `NULL`.

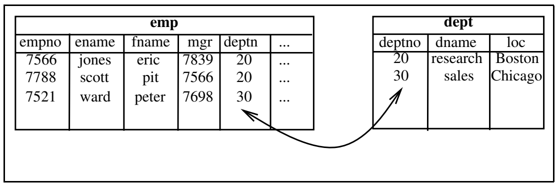

Figure 3: The mapping of the field `dpt`

Let us consider an object field `dpt` of the class `Employee`. It makes each `Employee` referring the `Department` it belongs to. This field can be mapped on the join shown figure 3 that links `emp` to the main table of the class `Department`. Then the referential attribute associated with the field is `emp.deptn`.

We can point out that this object field correspondence offers a way to modelize more or less strong links between objects : if any referential attribute is constrained by a clause `unique`, `not null` or both, the possible values for the corresponding object field are restricted. The stronger link is set when the referential attributes are also the key of their table. In that case :

- they must be valued : the associated object field cannot be empty.
- once the containing object or the contained object is persistent, its key value is fixed. As the key value of the other is settled at the same time (because of the join), and both objects are linked together for their life (!).

Correspondence of the set type

The correspondence of this field type is a *set join*. A set join is an equijoin between one of the elementary tables of the class the field belongs to and a *set table*. For a given object, the set table contains as many tuples as there are members in its set field value.

When the set is an atom set, the set join is completed with an attribute (of the set table) which contains the set members in the base.

When the set is an object set, the set join is generally completed with another join, this time between the set table and the main table of the referenced objects class. This second join must be an an equijoin that compares referential attributes and the key attributes of the joined main table.

Table **emproj**	
projno	**empno**
101	7566
103	7566
101	7788
...	...

Table **project**		
projno	**pname**	**budget**
101	alpha	250000.
102	beta	175000.
103	gamma	95000.
...	...	...

Figure 4: Tables `emproj` and `project`

For example, let us consider the table `emproj` shown figure 4. It is the representation in the base of the participation of every `Employees` to some `Projects`. If our class `Employee` owns a field `projects` (object set type), its mapping can be the join sequence (J[`emp, emproj`] J[`emproj, project`]) where the join expressions are respectively `emp.empno=emproj.empno` and `emproj.projno=project.projno`. In this example, the set table is of course the table `emproj`.

We also make possible to define the correspondence of an object set field in the form of a unique join between an elementary table of the class it owns to and the main table of the referenced objects class. In that case, the set table and the joined main table may be the same table. It happens when the join represents a N:1 relation. Then adding or removing members (objects) in a set finds expression in the database in updating the corresponding main tuples whereas it usually causes insertions or deletions of tuples in the set table.

An example of such a correspondence can be proposed for the field `employees` of the class `Department`. Indeed we can associate it with the join shown figure 3. Then, the new assignment of an `Employee` to a `Department` causes an update of the object tuple in the table `emp` : in this database, an `Employee` cannot work for more than one `Department`.

Correspondence of the list type

The correspondence of this type is quite similar to the one of the set type. It is made up by one or several attributes of the set table which values allow to arrange the list members and to differenciate the tuples corresponding to doubles. The first attribute defines the primary order, the second the secondary order, etc. For each of them, the sorting out can be in an ascending order or in a descending order.

Table **emproj2**		
projno	**empno**	**order**
101	7566	237
103	7566	121
101	7788	310
...	...	...

Figure 5: The emproj2 table

We show an example of list correspondence with the `emproj2` table of figure 5. Here the arrangement of members is determined by the attribute `emproj2.order`.

In the case of the object list, doubles are allowed only if the set table is not the main table of the referred objects and if the order attributes are not chosen in their elementary tables.

3 Benefits of the DRIVER approach

Let us now present the benefits of the DRIVER approach.

Firstly, the user chooses the object model to be handled. Thus, the accessed databases are directly viewed in his own object model, even if it is really a very own ad hoc model. More, he can easily supply volatile objects with persistency, at his convenience. Here, the OODBMS (i.e. DRIVER) doesn't impose which object model the application must work with. Thereby the OODBMS is not the main piece of the system any more. The DBMS is just a partner that simply offers a service to the application. Indeed that should be the only role of a persistency service for many applications.

Secondly, DRIVER gives an object access to very big amounts of data since the relational model is by now the most used DBMS standard. All relational databases are immediately available as object databases and conversely, all object databases built with DRIVER are of course immediately available and accessible to the numerous RDBMS users.

Industry has invested a lot in relational databases owing to the maturity of this technology. A lot of people aspire now to pass on to the object technology without giving up existing applications soon. To propose an OODBMS on top of a RDBMS lives up to this expectation. This solution does not upset the usual RDBMS users and allows the new users who need the object technology to access the same databases in the suitable form. Generally speaking, the DRIVER philosophy is a good solution to share data with many users. Data are expressed in a relational form –the simplest– in the database but they are used by everyone in another model, his own model with his own representation, an optimal choice of classes, relevant to his application.

Lastly, we believe that it is necessary to completely separate the object level, i.e. the knowledge representation level, from the physical level, i.e. the file manager level, to be able to make easily evolve the persistent object model. This separation is not complete in "classical" OODBMS. DRIVER uses RDBMS as intelligent file managers and proposes object models on top of them. Here the level separation is actual and the models should easily evolve with time.

References

[KJA93] A.M. Keller, R. Jensen, and S. Agarwal. Persistence software : Bridging object-oriented programming and relational databases. In *Proceedings of International Conference on Management of Data*. ACM SIGACT-SIGMOD, May 1993.

[Leb92] F. Lebastard. DRIVER v1.34, Reference manual. Technical Report 92-7 (in french), CERMICS-INRIA, Sophia-Antipolis (France), October 1992. 119 pages.

[Leb93] F. Lebastard. *DRIVER : A persistent virtual object layer for reasoning on relational databases.* Ph.D.Thesis (in french), CERMICS-INRIA Sophia-Antipolis (France), March 1993. 380 pages.

[Lee90] B.S. Lee. *Efficiency in Instanciating Objects from Relational Databases through views.* PhD thesis, Stanford University, Stanford (California), 1990. STAN-CS-90-1346.

[Int92] Intellitic International. MATISSE : Open semantic database - product overview. Technical report, Saint-Quentin-en-Yvelines (France), 1992. 16 pages.

[Sme90] Ilog, Gentilly (France). *SMECI Version 1.65, Reference manual*, May 1990. 470 pages.

[WBL+91] G. Wiederhold, T. Barsalou, B.S. Lee, N. Siambela, and W. Sujansky. Use of relational storage and a semantic model to generate objects : the PENGUIN project.

In *Database'91 : Merging policy, standards and technology.* The armed forces communications and electronics association, Fairfax (VA), June 1991.