Object-Relational Databases: An Area with Some Theoretical Promises and Few Practical Achievements

Marin Fotache, Al.I. Cuza University of Iaşi, Romania, fotache@uaic.ro
Cătălin Strîmbei, Al.I. Cuza University of Iaşi, Romania, linus@uaic.ro

Abstract

This paper tries to point out some of the promises, quarrels, achievements, and perspectives of the forced marriage between the relational and object-oriented data models, from both theoretical and implementation perspectives. As practical O-R achievements, the latest SQL standards (SQL:1999-2008) and Oracle implementation are discussed

Keywords: Databases, Object-Relational model, SQL, Oracle, Java

1. Introduction

The computing history has recorded many clichés and misunderstandings. One of them is Object-Relational (O-R). The major producers (Oracle, IBM) have stated that their DBMSs manage O-R databases. Most of the applications developed in the latest years combines Object Oriented (OO) and Relational technologies. So the idea that O-R model of data rules the database world today is quite logical. In fact, this is not quite true.

At the beginning of the 1990's the OO wave struck the database world [3][5][6]. Surrounded by an immense interest in OO technologies, relational model sinking seemed to be a matter of time. Relational database market was assumed to be eclipsed by the new OO database products before 2000. But something went wrong, and now OODBMSs cover just a tiny niche in the market.

Today, all the methodologies, tools, and platforms for application development are (more or less) OO, except the databases which remain "a relational island". The basic problem of "impedance mismatch" remains, and much effort is spent these days in mapping the application objects to database relational table rows.

2. Relational model, purists and traitors

Almost ten years passed between the publication of the E.F. Codd seminal paper [1] which marks the birth of relational data model and first commercial RDBMS. But within few years the market felt the expansion of the relational database products.

There are many advantages which can explain the tremendous success of the relational databases: clear, short and deep theoretical foundations (predicate logic), better and better DBMSs performance in storing and querying data, very active and enthusiastic communities of researchers and professionals, etc. SQL popularity was also decisive in relational model dominance, even if today SQL is seen by the relational "purists" [2] as a traitor of "relational laws".

At the end of 1980s many scholars and professionals felt uncomfortably with relational rigidity. Compared with OO, what relational database systems lacked most were abstract data types, complex integrity constraints and versioning [3]. So the OO data model was seen as a a better alternative to relational model because it could store persistent objects – data (properties) and code (behaviour). The idea of storing pieces of code within the database was not related, as expected, only with OO model. The network model (CODASYL) allowed database procedures, written in whatever programming language, to be invoked [4]. For relational databases, [4] were among the first to propose stored procedures. Actually, their idea was more generous even compared to current implementations – to have fields in tables which have as value a collection of commands in the query language supported by the DBMS.

3. Towards an Object-Oriented Data Model

According to Kim [5], OO concepts evolved in three different disciplines: first in programming languages (Simula-67, Smalltalk, C++), then in artificial intelligence (KEE, ART), and then in databases (semantic data models). The paradigm of OO programming is the encapsulation, within an object, of both the data and the programs which operates on data. Data is the state of the object (values of the attributes), and code is the behaviour of the object. The behaviour is invoked via messages through the interface.

Despite the accessibility and naturalness of OO, defining an OO data model has been a daunting task [3] [5] [6]. As for OODBMS, initially there were two main approaches [3] [5]: extending a relational DBMS with new data types, operators, and access methods; and extending an OO programming language with database functionality (persistence, authorization, concurrency).

A highly influential paper in establishing OO data model as academic and research fields is [6]. It claims a "pure" and orthogonal database adoption of object oriented model. The basic idea is to transparently handle the objects' pool and their relationships, produced by OO applications, between persistent (database) and transient (in-memory) contexts.

The most distinctive feature of the object oriented

database systems consists in the concept of "orthogonal persistence" [9] defined by three fundamental principles:
- *independency*: the application is not depending on the manipulated data lifetime (no distinction between transient and persistent objects);
- *data types orthogonality*: objects could become persistent no matter what is the type that they are defined on;
- *persistent identity*: the mechanisms to identify and supply persistent objects must be orthogonal regarding database system context.

These principles resemble to data independency ANSI/SPARC principle but generalized and applied over objects instead of relational data. Unfortunately this "object" independency principle has not surpassed the proposal level. OODBMS providers do not cross the boundary of a single OO programming language by offering cross-languages persistence mechanisms. Their mechanisms are much like extensions (as a specific API) of existing application platform, built around a programming language. The lack of compliance with the principle of data independency and the mathematical "weaknesses" of object oriented models' foundation, are the major reasons of OODBMS blame by relational theorists.

The authors of [6] define three categories of characteristics to be fulfilled by a DBMS in order to be declared OODS:
Mandatory: complex objects, object identity, encapsulation, types (classes), inheritance, overriding combined with late binding, extensibility, computational completeness, persistence, secondary storage management, concurrency, recovery, and ad hoc query facility;
Optional: multiple inheritance; type checking and inferencing, distribution, design transactions, and versions;
Open: programming paradigm, representation system, type system, and uniformity.

The major OODMBS producers gathered (www.odmg. org) and yielded three standards of OO database language: ODMG 1.0 in 1993, ODMG 2.0 in 1997 and ODMG 3.0 in 2000. The standards technical problems [10], the insignificant OO share in the database market and the OO advances in SQL:1999 leaded to group disbanding in 2001. The market share looks disastrous for OODBMS producers. The main technical drawbacks for OODBMS market failure have been, according to Leavitt [11], the strength of object-relational producers, bad performance and poor standardization.

4. Object-Relational Strands

The combination of the two data models, relational and object-oriented was proposed before coining the term object-relational. For example, Cattell and Rogers [12] announced a DBMS which combined OO and relational models of data. Ullman [7] anticipated the "synthesis of *object-* and *value-*oriented systems". More astonishing, Premerlani et al. [3] used the expression *OO Relational Database* to define a database system which combines a relational DBMS with an OO programming language. Their basic idea was to buffer the database with an OO layer that keeps relevant data in memory. The OO layer would hide the database from applications.

Nevertheless, the O-R does not deal with mapping application OO modules of code to a relational database, but how to store and manage natively object instances in table tuples.

The first major theoretical development of O-R data model is [13]. Thought as a reaction to [6], the article argues that the next generation (the third one) of database systems would be, in fact, extensions of relational model, proposing thirteen features to be incorporated, such as:
- rich data types - complex objects and abstract data types to model complex structures as meaning to overcome the relational model "flatness";
- inheritance (including multiple inheritance) - mechanism to get new from existing ones though derivation (no mater what it is about base or abstract types) taking into consideration the structure as well as object manipulation function inheritance;
- functions, including database procedures and methods
- unique identifiers (UIDs) should be assigned by the DBMS only when a user-defined primary key is not available;
- rules (triggers, constraints) - active rules to handle the events or actions materialized as queries as well as updates;
- non-procedural, high level access language;
- at least two ways to specify collections (by enumeration, and by query language);
- updatable views
- database accessibility from multiple high level programming languages;
- SQL support.

As a reaction to first manifesto [6] and the second one [13], Chris Date and Hugh Darwen started in 1995 a series of papers and books which circumscribe the so-called The Third Manifesto [2] [14]. If [6] argued for replacing relational model with an OO one, and [13] tried to conciliate the two models, the third manifesto is a frontal attack not only to OO data model (without blaming object orientation), but also to SQL standards. The basic idea is that objects are orthogonal, and the relational model can manage both standard (primitive) and user-defined (composite) types, no matter how complex they are. So the relational model "needs no extension, no correction, no subsumption, and, above all, no perversion" [2].

As consequence, two strange directives were

prescribed [14] to the database developers: *Relational Model Proscription 10: Not SQL* and *Relational Model Very Strong Suggestions 9: SQL migration*. Date and Darwen justify their "war declarations" against SQL by stating that SQL (especially SQL:1999) has undermined the traditional relational by extending the model in the wrong direction: the object oriented features are included in a completely isolated manner, defining object structures above relational ones and not on their basis. The final result: we can store objects in tables. Instead they had reconsidered the concept of domain (already defined in traditional relational theory as a set of values) and they treated it as real (abstract) data type thus applying it the OO principles like encapsulation and inheritance. The result: tables are variables, these variables contains sets of TUPLES (with their own types), and the value of each TUPLE contains a complex of values (attribute values) selected from a scalar type, TUPLE type or RELATION type.

Despite the quarrel, there is some degree of equivalence between SQL last standards (SQL:1999-SQL:2008) and [14] model, but the rigorous manner in which Date and Darwen propose the reformation of the relational model system type in conformance with the abstract data types theory make their approach more closed to conceptual model constructs. Also, Date and Darwen demand a clear distinction between specification and implementation of types. DBMS language (D language) must not have any reference to how a type (scalar or non-scalar) has to be implemented. This principle can be exploited in an interesting way: the implementation of types can be provided in a independent manner, so the DBMS might allow to construct the implementation in language like Java or C++, but the types to be exposed in the databases language (be it SQL or D) which is one of the O-R criteria proposed by Stonebreaker to the "new wave" of DBMSs [15].

Implementing the logical model proposed by the SQL:1999 standard and The Third Manifesto is a "mission impossible" because of inexistence of a real product (DBMS) to do this task. The problem is that the commercials DBMSs do not extend the relational model in the same way, but are closer to the SQL standards than [14] model.

5. Object-Relational Features in SQL Standards

Prior to SQL:1999 there was no support of user-defined types, collection, typed tables and other featured which today are included under object-relational label. As response to the pressure of application developers and threatening ODMG language OQL, soon after SQL-92 publication, the people involved in SQL standards started adding new options. According to Melton [16] SQL:1999 introduced SQL Object Model which has two distinct components, *user-defined types* and *typed tables*.

There are three kinds of user-defined types. First, *distinct types* are based on a single built-in data type (ex. INTEGER, VARCHAR), whose values cannot be directly mixed in operations with that built-in type or other distinct types; they can be used to define columns, just like any SQL built-in type [16]. Second, *structured user-defined types as values*, such as address type composed by street, number, city, county, zip code. As distinct types, structured types as values can be used for defining the type of the columns within tables. Third come *structured user-defined types as objects*. In a typed table, every row is an instance of a structured user-defined type as object. The table has one column for each property (attribute) of the user-defined type the table is defined on, and also a self-referencing column which is the unique object identifier (OID).

As for collection types, SQL is not complete or as rich as other OO languages. The only collection type supported by SQL:1999 is ARRAY. In SQL:2003 MULTISET was added. It is equally true, that many commercial implementations have their own collection types, more or less conformant with standards.

A major drawback of SQL standards is the lack of support for declaration of integrity constraints on types. It is amazing because some papers argued that OO model in better than relational in terms of integrity rules which can be implemented. The formal definition contains no definition in this regard. The only way of declaring constraints in SQL is CREATE/ALTER TABLE.

The "type" notion seems to be the key of the Date and Darwen theoretical foundation and also, the "abstract type" is the cornerstone of the Stonebreaker conception to extend existing relational systems. More specific, in D language [14] there are types everywhere, scalar and non-scalar (TUPLE and RELATION) and these categories of types support the principles of full-packaging (scalar type with no components, but with possible representations), user defined components (non-scalar types), inheritance and substitutability (either scalar or non-scalar), but without identity.

In SQL new standards there are several categories of types: basic data types, considered as atomic or intrinsic and extendible by users as distinct types, abstract data types (with their own components or attributes, identity, inheritance, polymorphism) and collection types (row type, list type, set type, multiset type). These categories of types are primary used in defining the types of attributes from tables' headings. Consequently, we can try to find the counterpart elements of scalar data types of The Third Manifesto in specifications of abstract data types:

- *Observer* functions can be considered an equivalent of read only operators;
- *Mutator* functions can be considered an equivalent of update operators;
- *Constructors* functions ca be used as selector operators;
- *Possible representation* components will be (unfortunately, only in a larger sense) as attributes of public interface associated with the abstract data type.

Date and Darwen had made a detailed analysis on SQL conformance with their prescriptions, proscriptions and suggestions; the main conclusions of them can be summarized as in Table 1.

Table 1. Date&Darwen Analysis of SQL Standards

RM Prescription 1 – Scalar types (possible representations, selector operators)	SQL conforms, with some observations about constructor function which in reality initialize an allocation storage and not select an arbitrary value from a domain
RM Prescription 3 – Scalar operators	SQL conforms with observers and mutators
RM Prescription 4 – Actual vs. possible representation	SQL require indeed a type representation based on individual attributes but does not explicitly differentiate internal and external representation
RM Prescription 5 – Expose possible representation	SQL conforms mostly because each attribute (equivalent with a component from representation) definition automatically causes definition of one observer method and one mutator method
RM Prescription 23 – Integrity constraints	SQL conforms with respect to attribute, relvar and database constraints, but fails completely with respect to type constraints
IM Prescriptions 1 – types are sets	SQL Conforms
IM Prescriptions 2 – subtypes are subsets	SQL Conforms
IM Prescriptions 8,9 – scalar values and scalar variables with inheritance	SQL Conforms
IM Prescription 10 – Specialization by constraint (subtypes values satisfies supertypes constraints, but shall exist at least one subtype value that satisfies an subtypes special constraint too)	SQL fail completely because of lack of type constraints
IM Prescription 14, 15 – TREAT DOWN operator and type testing	SQL conform thanks to TREAT AS and TYPE (X) [IS] OF (t) expressions
IM Prescription 16 – Read-only operator inheritance and value substitutability	SQL conforms
IM Prescription 17 – Operator signatures	SQL conforms
IM Prescription 18 – Read only parameters to update operators	SQL conforms
IM Prescription 19 – Update operator inheritance and variable substitutability	SQL fails, it require update operators to be inherited unconditionally (lack of type constraints)

So, even with the lack of type constraints and the consequences of this, SQL:1999, SQL:2003, and SQL:2008 standards acceptable support the scalar type as they are stated in The Third Manifesto.

6. Oracle Object-Relational Features

Oracle is one of the best database products, not only as SQL dialect and database logic application language (PL/SQL), but also as O-R features. Since Oracle 8i version, new options have been added in order to work more naturally with objects. Oracle does not use the term *class*, but *type*. Similarly to packages, every type has a (public) header and a (private) body. The next example creates the UDT type as value which is useful in managing addresses:

```
CREATE OR REPLACE TYPE all_addresses_type AS OBJECT (
  numb NUMBER(3),
  street VARCHAR2(50),
  zipcode NUMBER(6),
  city VARCHAR2(25),
  county VARCHAR2(25),
  MEMBER FUNCTION getNumb RETURN NUMBER,
  MEMBER FUNCTION getStreet RETURN VARCHAR2,
  MEMBER FUNCTION getZipCode RETURN NUMBER,
  MEMBER FUNCTION getCity RETURN VARCHAR2,
  MEMBER FUNCTION getCounty RETURN VARCHAR2,
  MAP MEMBER FUNCTION ordering RETURN VARCHAR2
        ) NOT FINAL NOT INSTANTIABLE
/
-------------------------------------------------------------------------
CREATE OR REPLACE TYPE BODY all_addresses_type AS
-------------------------------------------------------------------------
MEMBER FUNCTION getNumb RETURN NUMBER IS
BEGIN
  RETURN SELF.numb ;
END getNumb ;
-------------------------------------------------------------------------
...
-------------------------------------------------------------------------
MAP MEMBER FUNCTION ordering RETURN VARCHAR2 IS
```

```
BEGIN
  RETURN county || city || zipcode || street || numb ;
END ordering ;
-----------------------------------------------------------------

END ;
/
```

Most of the methods are member functions which can be applied to an instance (object) of *all_addresses_type* type (class). The type is not final, so it can have subtypes. It is also (by default) not instantiable.

Next we introduce a subtype of *all_addresses_type*, called *flat_address_type* with four new attributes (properties) and one overwritten method. This type is associated with flats (apartments) addresses.

```
CREATE OR REPLACE TYPE flat_address_type
            UNDER all_addresses_type (
  building_name VARCHAR2(30),
  floor VARCHAR2(10),
  apartment NUMBER(4),
  MEMBER FUNCTION getBuilding_name RETURN
VARCHAR2,
  MEMBER FUNCTION getFloor RETURN VARCHAR2,
  MEMBER FUNCTION getApartment RETURN NUMBER,
  OVERRIDING MAP MEMBER FUNCTION ordering
            RETURN VARCHAR2
) FINAL
/

CREATE OR REPLACE TYPE BODY flat_address_type AS
-------------------------------------------------------------------------
MEMBER FUNCTION getBuilding_name RETURN
VARCHAR2 IS
...

OVERRIDING MAP MEMBER FUNCTION ordering
            RETURN VARCHAR2 IS
BEGIN
  RETURN county || city || zipcode || street || numb ||
building_name ||
    entrance || floor || apartment ;
END ordering ;

END ;
```

Collections used to be one of the main weaknesses of the relational databases. Now it is not the case. In Oracle it is possible to store two types of collections, *nested tables* and *varrays* :

```
CREATE TYPE phones_type IS TABLE OF VARCHAR2 (15)

CREATE OR REPLACE TYPE person_type AS OBJECT (
  personid NUMBER(7),
  first_name VARCHAR2(30),
  last_name VARCHAR2(30),
  address all_addresses_type,
  phone_numbers phones_type,
  MEMBER FUNCTION getFirst_name RETURN VARCHAR2,
  MEMBER FUNCTION getLast_name RETURN VARCHAR2,
  MEMBER FUNCTION getAddress RETURN
all_addresses_type,
  MEMBER FUNCTION getPhone_numbers RETURN
phones_type,
  STATIC FUNCTION getWhoOwnsThePhoneNumber
            (phoneno_ VARCHAR2) RETURN person_type
        ) NOT FINAL
```

When some attributes of a typed table are nested tables, Oracle requires using NESTED TABLE clause in CREATE TABLE statement. As pointed out in previous section, in both SQL standards and Oracle constraints may be declared only at table creation (see PRIMARY KEY clause) and not at types definition.

```
CREATE TABLE people OF person_type (PRIMARY KEY
(personid))
        NESTED TABLE phone_numbers STORE AS
phone_nos_nt
```

Now, the table being created, the type body may be declared/compiled:

```
CREATE OR REPLACE TYPE BODY person_type AS
MEMBER FUNCTION getFirst_name RETURN VARCHAR2 IS
...

STATIC FUNCTION getWhoOwnsThePhoneNumber (phoneno_
VARCHAR2) RETURN person_type IS
  v_pers person_type ;
BEGIN
  SELECT p.object_value INTO v_pers FROM people p
  WHERE p.personid IN (SELECT x.personid FROM people x,
    TABLE (x.phone_numbers) t
    WHERE t.COLUMN_VALUE = phoneno_ ) ;
RETURN v_pers ;
END getWhoOwnsThePhoneNumber ;

END ;
```

Method *getWhoOwnsThePhoneNumber* is a search method, returning the instance (of *person_type*) which has a given phone number.
Inserting objects in PEOPLE typed table requires default constructors of every type involved:

```
INSERT INTO people VALUES
  (NEW person_type (1109, 'John', 'Doe',
    flat_address_type(22, 'Narowway', 700100, 'Smallcity',
'Smallcounty', 'GreenTower', '24', 546),
  phones_type ('00330232217000', '00330232217111')))
```

Querying tables of objects is similar to “regular” tables. Oracle violates OO encapsulation because attribute values are extracted not only through type method invocation, but also in classical SQL way. Invoking a method which is defined at a subtype in a hierarchy of types is possible using TREAT clause. IS OF [ONLY] clause is useful for selecting

instances of a certain (sub)type with or without its subtypes.

Querying nested tables values is possible based on TABLE clause which converts the collection (nested table) into table rows for which WHERE predicate could be applied.

```
SELECT p.first_name, p.getLast_Name() AS last_name,
  p.address.city,
   TREAT (p.address AS flat_address_type).getBuilding_name()
AS Build_Name,   t.*
FROM people p, TABLE (p.phone_numbers) t
WHERE p.address IS OF (ONLY flat_address_type)
```

We stop here now, even if there are many other Oracle O-R features which deserve deeper discussions: OIDs, referenced objects, collections updating, triggers on typed tables, etc.

7. Native OO platform (Java) implementation and SQL exposure in Oracle

Oracle has a solid foundation of object-relational features, based on notion of OBJECT TYPE. The approach presented here exploit a critical feature of Oracle object types: they are callable from SQL DML and DDL statements and can be implemented in independent programming language aside PL/SQL – the implicit procedural database programming environment, e.g. C, C++, Java (plus any other language that produce byte code runable on DBMS JRE).

The main drawbacks consist in that although CREATE TYPE and ALTER TYPE syntax is similar with that established by SQL3, there are some notable differences. One of the most painful shortcomings is the absence of public/private declaration of individual components. This minimal mechanism would allow to differentiate in some way between possible representations (components or attributes declared public) and internal representation (components or attributes declared private).

To make a practical demonstration of our approach of independently implement data types usable in database declaration as SQL statement, let's consider the Address type with the number, street, zipcode, city and county attributes and database representation. The SQL3 declaration of such a type will be:

```
CREATE TYPE all_addresses_type (
  numb NUMBER(3),
  street VARCHAR(50),
  zipcode NUMBER(6),
  city VARCHAR(25),
  county VARCHAR(25)
);
```

As we have already seen, the simplified Oracle syntax will be in first form:

```
CREATE OR REPLACE TYPE all_addresses_type AS OBJECT (
  numb NUMBER(3),
  street VARCHAR(50),
  zipcode NUMBER(6),
  city VARCHAR(25),
  county VARCHAR(25)
);
```

The generic data class to implement the type previously defined will be:

```
package javatypes;
public class CommonAddress {
  public Long numb;
    public String street;
    public Long zipcode;
    public String city;
    public String county;

    public CommonAddress(){}
    public CommonAddress(Long numb, String street, Long
zipcode, String city, String county) {
        this.numb = numb;
        this.street = street;
        this.zipcode = zipcode;
        this.city = city;
        this.county = county;
    }

    public Long getNumb() {
        return numb;
    }

    public void setNumb(Long numb) {
        this.numb = numb;
    }

    public String getStreet() {...}

    public void setStreet(String street) {...}

    public Long getZipcode() {...}

    public void setZipcode(Long zipcode) {...}

    public String getCity() {...}

    public void setCity(String city) {...}

    public String getCounty(){...}

    public void setCounty(String county) {...}

    public String toString(){
        return "Adress: " + this.numb + ", " + this.street + ", " +
                this.zipcode + ", " + this.city + ", " + this.county;
    }

}
```

Java class to be used as implementation of object types has to implement the SQLData interface, a

JDBC interface which is essential in creating Java instance of Oracle DDL object type. So, to conform to this demand we have to create a second class (or to modify the original class) that extends the first one and to implement the interface required:

```
package javatypes;
import java.sql.*;
public class CommonAddressType extends CommonAddress
implements SQLData {
   // original Oracle Object Type
   protected String sql_type;

   public CommonAddressType() {    }

   public CommonAddressType(Long numb, String street, Long
zipcode, String city, String county) {
      super(numb, street, zipcode, city, county);
   }

   public static CommonAddressType create() {
      return new CommonAddressType();
   }

   public static CommonAddressType create(Long numb, String
street, Long zipcode, String city, String county) {
      return new CommonAddressType(numb, street, zipcode,
city, county);
   }

   public String getSQLTypeName() throws SQLException {
      return sql_type;
   }

   public void readSQL(SQLInput stream, String typeName)
throws SQLException {
      sql_type = typeName;
      numb = Long.valueOf(stream.readString());
      street = stream.readString();
      zipcode = Long.valueOf(stream.readString());
      city = stream.readString();
      county = stream.readString();
   }

   public void writeSQL(SQLOutput stream) throws
SQLException {
      stream.writeString(numb.toString());
      stream.writeString(street);
      stream.writeString(zipcode.toString());
      stream.writeString(city);
      stream.writeString(county);
   }
   public static CommonAddressType
set_numb(CommonAddressType adr, Long numb) {
      adr.numb = numb;
      return adr;
   }
   public static CommonAddressType
set_street(CommonAddressType adr, String street) {
      adr.street = street;
      return adr;
   }
   public static CommonAddressType
set_zipcode(CommonAddressType adr, Long zipcode) {
      adr.zipcode = zipcode;
      return adr;
   }
   public static CommonAddressType
set_city(CommonAddressType adr, String city) {
      adr.city = city;
      return adr;
   }
   public static CommonAddressType
set_county(CommonAddressType adr, String county) {
      adr.county = county;
      return adr;
   }
}
```

The *loadjava* tool will be the vehicle to "port" these two classes in the database schema, and the DDL declaration of object type implemented by the CommonAddressType will be as follows:

```
CREATE OR REPLACE TYPE all_addresses_type AS OBJECT
  EXTERNAL NAME 'javatypes. CommonAddressType'
LANGUAGE JAVA
  USING SQLData(
 numb NUMBER(3) EXTERNAL NAME 'numb',
 street VARCHAR(50) EXTERNAL NAME 'street',
 zipcode NUMBER(6) EXTERNAL NAME 'zipcode',
 city VARCHAR(25) EXTERNAL NAME 'city',
 county VARCHAR(25) EXTERNAL NAME 'county'

STATIC FUNCTION construct(numb NUMBER, street
VARCHAR, zipcode NUMBER, city VARCHAR, county
VARCHAR) RETURN all_addresses_type
     EXTERNAL NAME 'create (java.lang.Long, java.lang.String,
java.lang.Long, java.lang.String, java.lang.String) return
javatypes. CommonAddressType ',

STATIC FUNCTION set_numb(adr all_addresses_type, numb
NUMBER) RETURN all_addresses_type
EXTERNAL NAME 'set_street(javatypes. CommonAddressType,
java.lang.Long)return javatypes. CommonAddressType ',

STATIC FUNCTION set_street ... ,

STATIC FUNCTION set_zipecode...,

STATIC FUNCTION set_city...,

STATIC FUNCTION set_county...,

MEMBER FUNCTION to_string RETURN VARCHAR2
    EXTERNAL NAME 'toString() return java.lang.String'
)
```

With EXTERNAL NAME LANGUAGE JAVA and USING SQLData declarations, DBMS engine will note that CommonAddressType class will be the back-end of ALL_ADDRESSES_TYPE object type. So what we can do with this type: we can associate a table column with it:

```
CREATE TABLE customers
(id NUMBER(4), name VARCHAR(30), residence
ALL_ADDRESSES_TYPE)

INSERT INTO customers
VALUES (1001, 'Alpha Inc.', ALL_ADDRESSES_TYPE (11,
'Carol 1', 6600, 'IASSY', 'IASSY'))
```

And finally we can try to expose the value type using and "accessor" function (to_string()) that can implement a convenient "possible representation":

```
SELECT c.name, c.residence.to_string() FROM customers c;
```

8. Conclusions and open issues

The topic of "post-relational" age in database systems is a generous one. From many directions that have been proposed, this paper deal with three strands of O-R data model, summarized as follows:

- OR Database Systems, asserting extension of the existing relational "infrastructure" by engrafting OO generic innovations promoted by semantic data modeling and application programming (Stonebreaker, SQL:1999-2008);
- Relational database systems with orthogonal OO features (Date&Darwen), claiming that all "traditional" relational theory could remain unaltered;
- "Pure" OO Database Systems, with no-reference to the existing relational theory, supplying "orthogonal persistence" to the application objects.

One of the major drawbacks of today O-R applications is the "impedance mismatch" between OO and the relational layers. Much effort has been purported to finding adequate mapping tools between OO classes and relational tables [8] [18] [19].

As proved in Sections 5, 6 and 7, both SQL Standard and major database servers have powerful options for dealing with all the major aspects of O-R model and applications.

It is sad that, instead of creating types and managing them within O-R tables, and then mapping directly classes in application logic layer to typed tables in database layer [8] [20], most of the application developers just map classes to relational tables, failing to exploit the strengths of O-R model.

References

[1] Codd, E.F., A Relational Model of Data for Large Shared Data Banks, *Communications of the ACM, 13*(6), 1970, 377-387.

[2] Darwen, H., Date, C.J., The Third Manifesto, *ACM SIGMOD Record, 24*(1), 1995, 39-49

[3] Premerlani, W.J., Blaha, M.R., Rumbaugh, J.E., Varwig, T.A., An Object-Oriented Relational Database, *Communications of the ACM, 33*(11), 1990, 99-109.

[4] Stonebraker, M., Anton, J., Hanson, E. Extending a Database System with Procedures, *ACM Transactions on Database Systems, 12*(3), 1987, 350-376.

[5] Kim, W., Research Directions in Object-Oriented Database Systems, *Proc. of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, Nashville, TE, 1990, 1-15.

[6] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonick, D., The Object-Oriented Database System Manifesto, *Proc. of the International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, 1989, 1-17.

[7] Ullman, J.D. Database Theory: Past and Future, *Proc. of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, San Diego, CA, 1987, 1-10.

[8] Lodhi, F, Ghazali, M.A., Design of a simple and effective object-to-relational mapping technique, *Proceedings of the 2007 ACM symposium on Applied computing*, Seoul, Korea, 2007, 1445-1449.

[9] Atkinson, M P., Morrison R. *Orthogonally Persistent Object Systems*, VLDB Journal 4 (3), 1995, 319-401

[10] Kim, W., Observations on the ODMG-93 Proposal for an Object-Oriented Database Language, *ACM SIGMOD Record, 23*(1), 1994, 4-9.

[11] Leavitt, N., Whatever Happened to Object-Oriented Databases?, *Computer, 33*(8), 2000, 16-19.

[12] Cattell, R.G.G, Rogers, T.R., Combining Object-Oriented and Relational Models of Data, *Proc. of the 1986 international conference on Management of data*, Portand, OR, 1986, 78-87.

[13] Stonebraker, M., Rowe, L.A., Lindsey, B., Gray, J., Carey, M., Brodie, M., Bernstein, P., Beech, D., Third-Generation Database Systems Manifesto, *ACM SIGMOD Record, 19*(3), 1990, 31-44

[14] Date, C.J., Darwen, H., *Databases, types, and the relational model. The third manifesto* (Reading, MA: Addison-Wesley, 2007).

[15] Stonebraker, M., Brown, P., Moore, D., *Object-relational DBMSs: tracking the next great wave* (San Francisco, CA: Morgan Kaufmann, 1999).

[16] Melton, J., *Advanced SQL:1999. Understanding Object-Relational and other advanced features* (San Francisco, CA, Morgan Kaufmann, 2003).

[17] Seshadri, P., Enhanced abstract data types in object-relational databases, *VLDB Journal, 7*(3), 1998, 130-140

[18] Agarwal, S., Architecting Object Applications for High Performance with Relational Databases, *OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, 1995, available at: http://www-db.stanford.edu/pub/keller/1995/high-perf.pdf

[19] O'Neil, E., Object/relational mapping 2008: hibernate and the entity data model, *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, Vancouver, Canada, 2008, 1351-1356.

[20] Vara, H.M., Vela, B., Cavero, J.M., Marcos, E., Model Transformation for Object-Relational Database Development, *Proceedings of the 2007 ACM symposium on Applied computing*, Seoul, Korea, 2007, 1012-1019.