Object Relational Databases

Index

- Object-based models: revolutionary approach (OO) and evolutionary (OR)
- Historical examples of ORDBMS and evolution of SQL
- Evolutions of the relational model:
  - Removing the constraint of the first normal form
  - Object with identity
- The object type and object relational model
  - Equality and methods
- The data model in SQL-3
  - Types
  - Methods
  - Instances Ordering
  - Inheritance of types and tables

Logical design of an object-relational databases

  - Starting from ER diagrams
  - Starting from UML class diagrams

Objects-based models: approaches

We can recognize two approaches to the introduction of objects in databases.

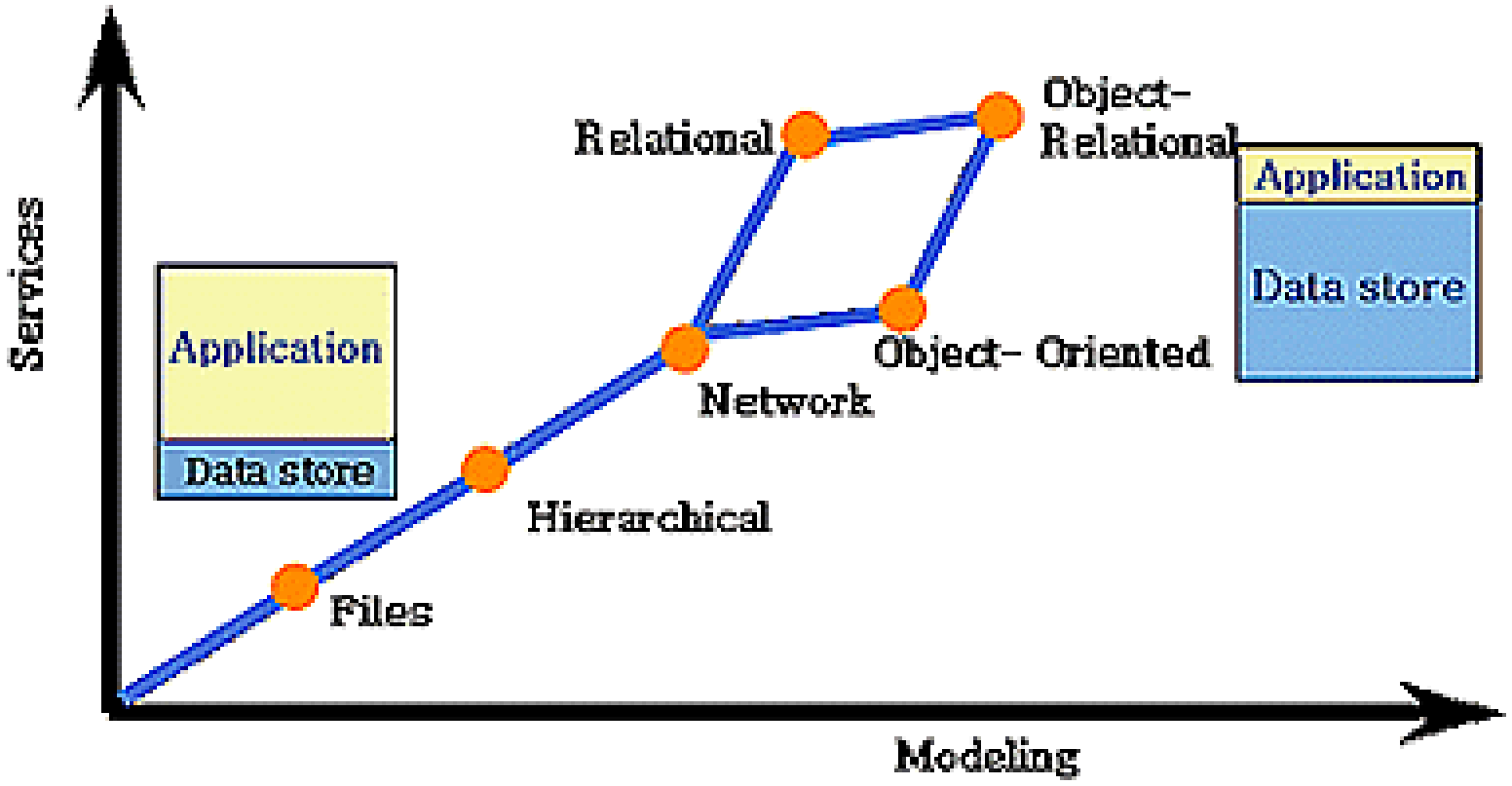
The object oriented databases (*Object-Oriented Database Management Systems, OODBMS*) They have followed a more revolutionary approach, extending the DBMS from the features of object-oriented programming languages.

The object-relational databases (*Object-Relational Database Management Systems, ORDBMS*) They have followed a contrary evolutive approach, integrating the concept of the object in the relational model.

The two approaches, very confrontational in the early '90s, were later proven to be quite convergent.

Objects-based models: approaches

The evolution of the models



The object-relational model

Examples of ORDBMS (historical) are:

- Illustra (Version marketed by Illustra Information Technologies Ltd. POSTGRES, designed by M. Stonebraker as successor of Ingres, the first example of RDBMS).
- Omniscience (Omniscience Object Technology, Inc.)
- UniSQL (UniSQL Inc, a company founded by Won Kim, who had designed the system ORION objects).

Starting from the standard SQL: 1999 (Also known as SQL-3), the SQL has been extended with object-relational characteristics. However, the major relational DBMS manufacturers (for example, Oracle, and IBM) have extended their products with object-relational characteristics, often wider than those provided by SQL-3 and not always adhering to it.

The object-relational model

The same situation occurred with open source systems such as PostgreSQL.

The fifth revision of the SQL standard, SQL: 2003, supports the following new object-relational features:

- Clearer specification of the *object relational*
- The nested collection type (*nested)* MULTISET

It is expected that the ORDBMS now adapt quickly to the new standards in the market.

The sixth revision of the SQL standard, SQL: 2008, introduces important developments in the treatment of data in XML, while the last revision SQL: 2023 adds JSON data type.

Extensions of the relational model: removing first normal form

A first proposal for an extension to the relational model was the elimination of the constraint that attributes are primitive types.

In particular, the relation type is thus redefined:

- int, real, bool and string are *primitive types*;
- if T_1, ..., T_n are primitive types or *set types* and A_1, ..., A_n are distinct labels, said *attributes*, then $(A_1:T_1, ..., A_n:T_n)$ is a *tuple type*;
- if T is a tuple type, then {T} is a *set type*, and the type of relations (tables) is a set type.

So relationships can be nested.

Extensions of the relational model: removing the first normal form

From the relational model in first normal form, it remains the fact that the associations between data from different relationships can be expressed with the mechanism of foreign keys.

From the object-based model, it is just taken the opportunity to define relations with tuples which contain non-elementary components, but not the ability to share tuples, to define hierarchies or procedural aspects.

On nested relationships, new relational algebra operators are defined. They are a generalization of the classical relational algebra operators.

Extensions of the relational model: object identity

Another extension of the relational model replaces the notion of tuple with the notion of ***object identity***.

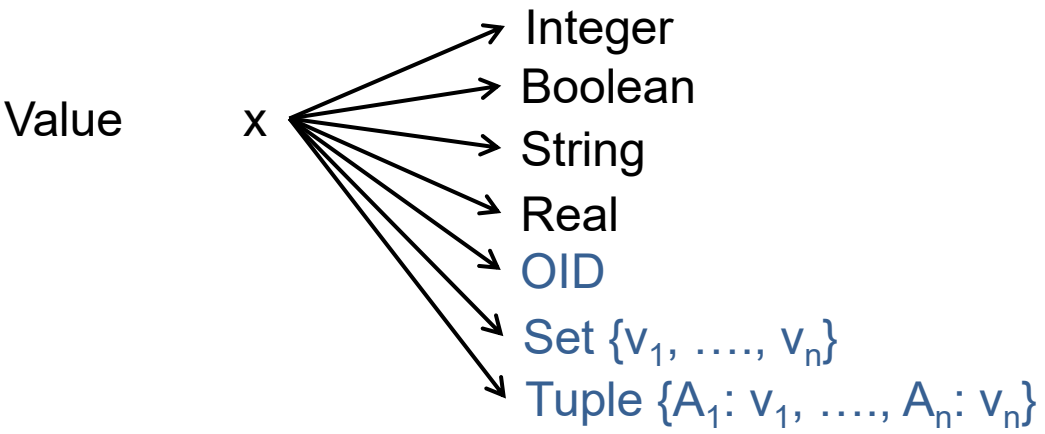
A ***object*** is a pair

(Identifier, value)

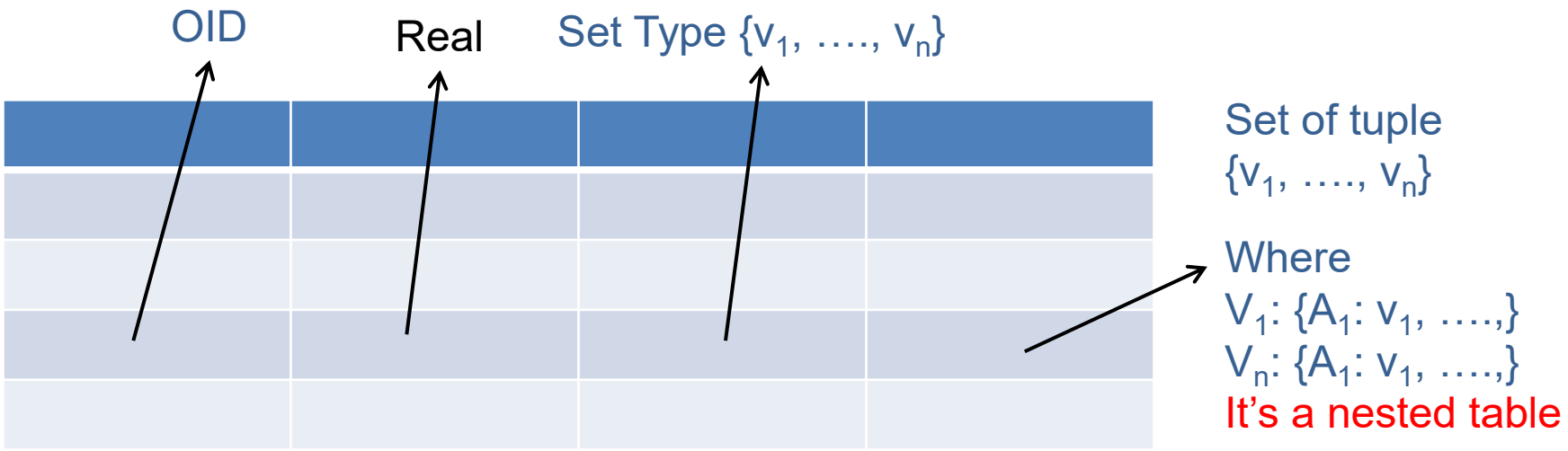
where the identifier (**OID**) is unique and unchangeable, while the value is a tuple defined as follows:

- integer, real, boolean, string, and ***OID*** are ***primitive values*** (Undef denoted by *nil*)
- if v_1, ..., v_n are values of the same type, then $\{v_1, ..., v_n\}$ is a ***value of set type***;
- if v_1, ..., v_n are values, and A_1, ..., A_n are distinct labels, said *attributes*, then (A_1: = v_1, ..., A_n: =v_n) is a ***tuple value***.

Extensions of the relational model: object identity

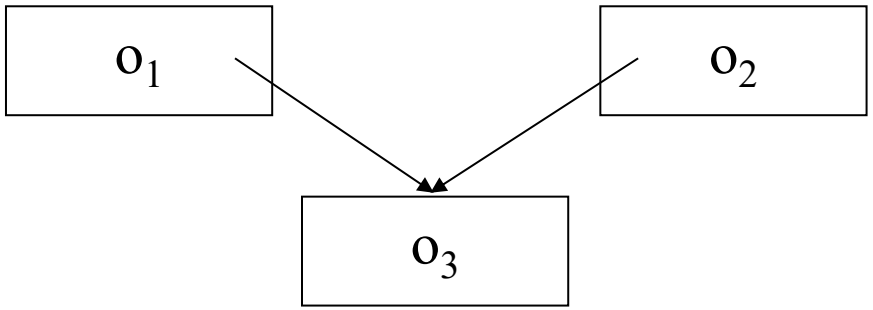


For example:



Extensions relational model: object identity

The use of objects with identity lets us build objects o_1 and o_2 which have as a component the identifier of the same object o_3. Since the identifier of an object is immutable, changes to the state of o_3 are automatically reflected in o_1 and o_2.



The object- relational model

A formalization of the object relational model passes through that of the ***object type***, which can be defined as follows:

- int, real, bool and string are ***primitive types***;
- if T_1, ..., T_n are types and A_1, ..., A_n are distinct labels, said *attributes,* then (A_1:T_1, ..., A_n:T_n) is a ***tuple type***;
- if T is a type, then {T} is a ***set type***;
- if T is a tuple type, then objecttype IDE:= T defines an ***object type*** named IDE, whose values have type T.
- if T is an object type, then objecttype T2: = is T plus (B_1:T_1, ..., B_n:T_n) and defines a new ***object type*** by ***inheritance***.

Defined the object type, we can define the following:

- If T is an object type, ref (T) is the ***type of OIDs*** of objects of type T.

The object-relational model

On the set of types, we can also define a relationship "***subtype***". One type T is a subtype of type T' ($T \subseteq T'$) when all the operators and functions defined on values of type T' can be applied to the values of type T.

A ***object-relational schema*** is a collection of
- Definitions of object types
- Schemes of relations (table), each associated with an object type.

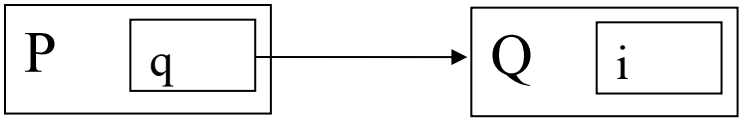
In an object-relational schema associations between tables of objects can be modeled by the mechanism of ***aggregation***, typical of the object oriented paradigm, that is, by inserting in the objects of a table the OID of the object associated in the other, or the set of OIDs if the association is multivalued.

The object- relational model

The aggregation may be represented in two ways, leading to two different data models.

The function ref is applied to an object in order to extract the OID, while the function deref is applied to an OID to obtain the related object.

In the data model specified earlier we explicitly distinguish the type of an object from the type of OIDs. This is the approach used in SQL: 2003 and Oracle, and is called ***relational model with object identity***.

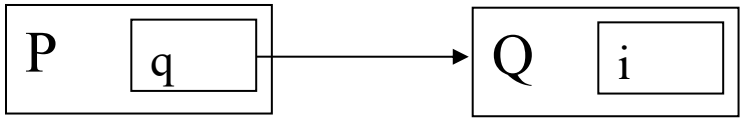


deref(P.q).i

Q==deref(P.q)

The object- relational model

Another possible approach is to hide the existence of the type of OIDs. In this case, an object type in a field of a tuple type is interpreted by the system as a type OID. Now the system automatically inserts the operations deref and ref when necessary.



P.q.i

This model is said ***object relational*** and it is the one adopted in UniSQL and PostgreSQL.

Equality between objects

Two objects o_1 and o_2 are equal if they have the same OID, or if they are the same object.

The equality of two objects o_1 and o_2 is checked writing $o_1 = o_2$ in object-relational model, while in the relational model with object identity it is necessary to explicitly compare the OID, writing $ref(o_1) = ref(o_2)$.

The operator of ***superficial equality*** (**==**) checks if two objects have the same value.

Definition of methods

An object is characterized not only by structural elements, but also by its behavior, defined by the **methods**.

A method has three components: the name, the signature and implementation (or body). Usually in the relational models with objects we take the following alternatives:

- In the object type we specify only the name and the signature of a method. The implementation is specified separately, even in a language different from that used to define the objects, if it does not allow to treat the procedural aspects.
  SQL3
  Oracle

Definition of methods

```
CREATE TYPE MyType {
 ------
 ------
 ------
  Method1()
}

CREATE TYPE BODY MyType {
   Method1(){
        <implementation of method1>
   }
}
```

Definition of methods

The object type contains no information on methods. The name, signature and the implementation of a method are defined separately as a function having between its arguments the type of the object to which it refers. To enable the redefinition of the methods in the types defined for inheritance, a function can be redefined with the same name specializing the type of the object used as an argument (***overloaded function***).

PostgreSQL

Definition of methods

```
CREATE TYPE MyType {
 ------
 ------
 ------
}

Method1(MyType, …){
        <implementation of method1>
   }
```

Data Model for SQL-3

The SQL-3 standard before and SQL: 2003 standard then, have introduced different characteristics of the object in the relational model. The major extensions in this sense are:

1. The extension of the system of the types of SQL in order to provide the user with the possibility of define new data types relevant to the domain of interest. The new data types allow you to model
   - Tuples,
   - Collections of values,
   - Structured values.

   Such types can then be used in the definition of the table schema.
2. The ability to define methods.

Data Model for SQL-3

3. The introduction of the
   - object identifiers
   - reference types

so as to allow the representation of complex objects between their associated (association based on references, different from the one based on value, typical of the relational model).

4. The introduction of the inheritance of types and tables, to allow the reuse of user-defined types.

Following these extensions will be illustrated in detail.

Implicit row type

Consistent with the previous version, in SQL-3 is possible to define tables with primitive type fields:

```
CREATE TABLE Person (
   Name          VARCHAR (30) NOT NULL,
   Residence     VARCHAR (30),
   Tax_code            CHAR (16) PRIMARY KEY)
```

Each table consists of a set of tuples and the type of each tuple of a table is said row type. In the Person table, the row type is implicitly defined.

Explicit row type (ROW)

The row type in SQL: 1999 is an extension of the anonymous row type that SQL always allowed. It offers designers of databases the opportunity to define structured values in single columns of the database:

```
CREATE TABLE employee (
  emp_id    INTEGER,
  name ROW (
                          given VARCHAR (30),
                          family VARCHAR (30)),
  salary REAL )
```

Structured type

The suggested approach in SQL-3 is that of first explicitly defining a type (eg, the type PersType), said structured type, and then making it reusable.

```
CREATE TYPE PersType AS ROW (
Name                    VARCHAR (30) NOT NULL,
Residence    VARCHAR (30),
Tax_code                CHAR (16) PRIMARY KEY);
CREATE TABLE Person OF TYPE PersType;
```

ROW is optional

The type PersType can also be used for other typed tables.

```
CREATE TABLE Industrial OF TYPE  PersType;
CREATE TABLE Pilot OF TYPE PersType;
```

Simple types

In addition to the definition of row types, the CREATE TYPE construct is used to define new primitive types, called simple types, starting from the standard primitive types, associating them user-defined operations.

Example:

CREATE TYPE Dollar AS DECIMAL(8, 2)

CREATE TYPE Euro AS DECIMAL (8,2)

The simple type values are not directly comparable with the values of the type on which they are based.

For this reason simple types are also known as distinct types.

Casting of simple types

To compare, we must use the casting functions that convert instances of a simple type in instances of primitive type and vice versa.

SQL requires that an ORDBMS automatically creates two casting functions when we create a new simple type:

1. simple type → primitive type on which it is based
2. and viceversa

These functions may, however, be subsequently modified, by defining a new casting with the command function CREATE CAST.

Casting of simple types

Example: Creation of a casting function

```
CREATE CAST ( Euro AS Dollar)
WITH Euro2Dollar(Euro)
AS IMPLICIT;
```

This definition says that, to convert Euro values in Dollar values we must use the function Euro2Dollar already present in the database.

```
CREATE FUNCTION Euro2Dollaro( e Euro) RETURNS Dollar
BEGIN
DECLARE g DECIMAL (8,2);
SET g = e; ← use the cast function between primitive and simple types
RETURN g * 1.17; ← idem
END;
```

Casting of simple types

The AS ASSIGNMENT clause used in the CREATE CAST command ensures that, in case of comparison between a value of type Euro and a value of type Dollar, the casting function is implicitly invoked.

Example: If Payment has two fields, amountEurope of type Euro and amountUSA in Dollar, querying

```
SELECT *
FROM Payments
WHERE amountEurope> amountUSA;
```

involves the invocation of the conversion of amountEurope in Dollar.

References

In SQL-3 we can use references from one structured type to another structured type, thus creating the conditions for sharing objects in the DB

```
CREATE TYPE t_residence AS (
Street                  VARCHAR (20)
No                              VARCHAR (10)
Postal_Code     INTEGER DEFAULT '70100',
city                            VARCHAR (20) DEFAULT 'Bari');

CREATE TYPE t_customer AS (
Code            DECIMAL(4)
Name            VARCHAR (20)
surname         VARCHAR (20),
residence       t_residence);
CREATE TABLE Customer OF TYPE t_customer;
```

References

What if we have:

```
CREATE TYPE t_residence AS (
Street                  VARCHAR (20)
No                              VARCHAR (10)
Postal_Code     INTEGER DEFAULT '70100',
city                            VARCHAR (20) DEFAULT 'Bari');

CREATE TYPE t_customer AS (
Code            DECIMAL(4)
Name            VARCHAR (20)
surname         VARCHAR (20),
residence       ref(t_residence));
CREATE TABLE Customer OF TYPE t_customer;
CREATE TABLE Residence OF TYPE t_residence;
```

References

We can also use the type t_residence directly in the definition of a customer table:

```
CREATE TABLE Customer (
    Code            DECIMAL(4)
    Name            VARCHAR (20)
    surname     VARCHAR (20),
    residence   t_residence);
```

The insertion of a tuple in the customer table uses the constructor automatically added to the default type t_residence:

```
INSERT INTO Customer
VALUES (6650, 'Mario', 'Rossi', NEW t_residence());
```

Each sub-column of the residence will assume NULL column value for the new tuple, with the exception of city and Postal_Code, which will take the default values.

Dot notation

The values of the sub-columns can then be initialized using the dot notation to refer to components of tuples attributes and row types.

Example:

```
UPDATE customer
SET residence.street = 'via Re di Roma', residence.no = '7'
WHERE code = 6650
```

The *dot notation* can also be used in queries:

```
SELECT residence.city
FROM customer;
```

The created types can be deleted with the command

```
DROP TYPE <type name> {RESTRICT | CASCADE};
```

CASCADE also deletes all schema elements that refer to the deleted type.

Reference types

REF is a type constructor that must be parameterized by the name of a row type to produce a specific type whose values are references (OIDs) to instances of the type row type.

Example:

```
CREATE TABLE Video (
    colloc          DECIMAL (4),
    movie                   REF(t_film) NOT NULL,
    type                    CHAR NOT NULL DEFAULT 'd');
```

where t_film is the name of a row type previously defined.

In this example, instances referenced by the values of the column movie of the table Video, may belong to any typed table of t_film type. In this case the reference is ***unconstrained***.

Reference types

To ensure referential integrity, and then force that valid values for a type REF (<type name>) column are formed by OIDs of tuples contained in a certain typed table of <type name>, with name <table name>, we can use the SCOPE clause.

This is used in the REF syntax: REF(<type name>) SCOPE <table name>. This clause can be used both in the definition of a type and in the definition of a table schema (Typed or not).

Example:

```
CREATE TABLE Film OF TYPE t_film;
CREATE TABLE Video (
    colloc          DECIMAL (4),
    movie                   REF (t_film) SCOPE Film NOT NULL,
    type                    CHAR NOT NULL DEFAULT 'd');
```

where Film is the name of a table with rows of type t_film.

Reference types

We could also define the type:

```
CREATE TYPE t_video AS (
    colloc      DECIMAL (4),
    movie               REF (t_film) SCOPE Film,
    type        CHAR DEFAULT 'd');
```

and then the table:

```
CREATE TABLE Video1 OF t_video
(   movie WITH OPTIONS SCOPE shortFilms NOT NULL
    type WITH OPTIONS NOT NULL);
```

The SCOPE clause contained in the definition of t_video can still be redefined in typed tables having t_video as a row type.

Reference Operator

The navigation along the references between types is obtained by the ***reference operator*** -> that allows access from a source object x to an attribute A of an object y referenced in x, in the following way:

x -> A

Example:

If film points to objects on a table with a column title then we can write:

```
SELECT movie->title
FROM Video
WHERE colloc = 1311
```

Reference Operator

Using the reference operator, this query allows to navigate from the table to the tuples of table Video to the tuples of table Film. This navigation is a kind of implicit join based on equality between the OID in the film column and the OID of the tuples contained in the Film table.

In general, the presence of the reference types allows to perform a navigational access to the data, as opposed to the typical associated access of relational languages, done through the primary and foreign keys mechanism.

Values generated by the system

The object identifiers declared in a row type may also refer to the same row type.

Example:

```
CREATE TYPE AS t_emp (
    Name            VARCHAR (15),
    address    Tipo_Ind,
    age             INTEGER,
    id_emp          REF (t_emp));
```

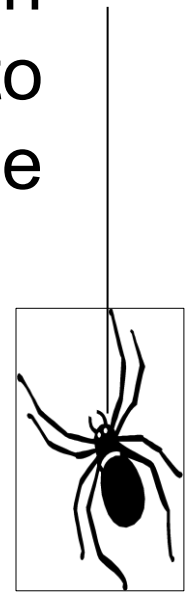
In this case, we can specify that values of id_imp are system generated

```
CREATE TABLE Employee OF t_emp
VALUES  FOR id_emp ARE SYSTEM GENERATED;
```

Values generated by the system

The id_emp values of object identifiers, created by the system, but can be used in queries like other attributes.

NB: The direct use of identifiers can result in the presence of tuples that remain orphan (***dangling references***), when the references to them are explicitly deleted or modified by the query.



Collection types

SQL:2003 provides two types of collections: ARRAY and MULTISET.

The type ARRAY, already present in SQL:1999 is defined as follows:

<Component Type> ARRAY [<size>]

where <size> is an integer value that represents the size of the array and <component type> is the type of the array elements.

Example:

Suppose we want to assign to each client a list of at most three phone numbers. The schema of the customer table can be defined as follows:

Collection types

```
CREATE TABLE customer
(codCus          DECIMAL (4) PRIMARY KEY,
 name            VARCHAR (20),
 surname         VARCHAR (20),
 phone           CHAR(15) ARRAY [3],
 DOB             DATE,
 residence       t_residence);
```

The number of elements in phone is between 1 and 3.

ARRAY [] represents an empty array.

Collection types

```
UPDATE customer
SET phone = ARRAY [ '0805443196', '080533555']
WHERE codCus = 6635;
```

In this example, the value of array type has been created by applying the constructor ARRAY and a set of values with type <component type>. This is a structured value.

The following query returns the first stored telephone number for all tuples in the customer:

```
SELECT phone[1]
FROM customer;
```

Collection types

The type MULTISET was introduced in SQL:2003 and was not present in SQL:1999. It is defined as follows:

<Component Type> MULTISET

where <component type> is the type of the components of the multi-set.

Unlike the type ARRAY, the type MULTISET does not specify an order between the elements of the collection and does not require the specification of the maximum cardinality of the collection.

Duplicates are allowed.

Collection types

Example:

Suppose we want to assign each client a certain set of phone numbers, without specifying an order or an upper bound on the cardinality of this set. The customer table of the schema can be defined as follows:

```
CREATE TABLE customer
(codCus          DECIMAL (4) PRIMARY KEY,
 name            VARCHAR (20),
 surname         VARCHAR (20),
 phone           CHAR (15) MULTISET,
 DOB             DATE,
 residence       t_residence);
```

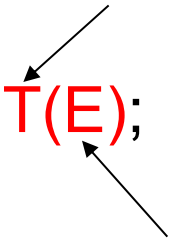
Collection types

A multi-set type value can be created by applying the corresponding constructor to a set of values of <component type>.

```
UPDATE customer
SET phone = MULTISET [ '0805443196', '080533555']
WHERE codCus = 6635;
```

The access to the elements of a multi-set can take place using suitable functions that transform a MULTISET type value in a table, which can then be accessed with the usual mechanisms.

```
SELECT c.name, c.surname, T.E
FROM customer c, UNNEST(c.phone) T(E);
```



Collection types

The alias c represents a generic client, and therefore corresponds to phone numbers of c.phone. UNNEST (c.phone) returns a table whose schema is set equal to T(E) through the use of an alias. The expression T.E allows access to the column E of T.

Finally, the following instruction updates the customer's phone numbers with 6635 code, adding the number 3334445678.

```
UPDATE customer
SET phone= phone MULTISET UNION DISTINCT MULTISET [
    '3334445678']
WHERE codCus = 6635;
```

Collection types

On multi-sets, the following operations are permitted: merge (MULTISET UNION), Difference (MULTISET EXCEPT), Intersection (MULTISET INTERSECT).

These operations can be followed by the qualifier DISTINCT if duplicates must be eliminated by the result.

Nesting

The nesting operation is the opposite of unnesting, and is used to create an attribute with values that are of type collection. Nesting is not provided by the SQL: 1999, but has been introduced in SQL 2003 using the aggregate function COLLECT(). The function takes a column of the element type as input and creates a multiset from selected rows.

Example:

Consider the following relation *flat-books* in first normal form:

title	*author*	*pub-name*	*pub-branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Nesting

Example (Cont.)

The following query creates a nested relationship based on attribute *keyword*:

```
SELECT title, author, Publisher(PUB_NAME, pub_branch )
      AS publisher, COLLECT (keyword) AS keyword_set
FROM flat-books
GROUP BY title, author, publisher
```

while the following query aggregates on both *authors* and *keyword*:

```
SELECT title, COLLECT (author ) AS author_set,
       Publisher (PUB_NAME, pub_branch) AS publisher,
       COLLECT (keyword ) AS keyword_set
FROM flat-books
GROUP BY title, publisher
```

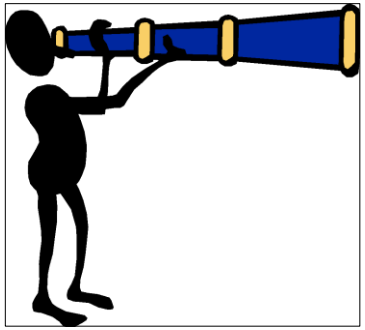
SQL-3 data model

Example:

```
CREATE TYPE PlantType AS (
    Name            VARCHAR (25),
    City                VARCHAR (7),
    NoEmpl          INTEGER
)

CREATE TYPE CostrType AS ROW(
    CostrId             REF(CostrType)
    Name                    VARCHAR (25),
    President           REF(PersType)
    Plants      PlantType MULTISET
)
```

Model for SQL-3 data

```
CREATE TYPE SparePartsCarType AS (
    Engine                  CHAR (10),
    Shock       CHAR (5))

CREATE TYPE CarType AS ROW (
    Plate                           CHAR (10) PRIMARY KEY,
    Model                   VARCHAR (30),
    Builder             REF(CostrType)
    MechanicalParts SparePartsCarType)
```



The types PlantType and SparePartsCarType are used in the context of the types CostrType and CarType without, however, introducing the construct REF, then without being introduced as independent objects.

Methods in SQL-3

We can associate ***methods*** at each instance of simple or structured type.

One method has a

- signature (Method name + parameters definition)
- implementation.

The SQL standard provides SQL procedural extension in SQL/PSM (Persistent Stored Module) that can be used to define the implementation of functions and procedures.

Alternatively, the implementation of the methods can be described in other programming languages.

The instance on which the method is invoked is a implicit parameter (which is added to those explicit) and is indicated with SELF.

Methods in SQL-3

For each type T defined by the user, a default constructor T() is implicitly defined. It returns a new object of that type, initializing each attribute to its default value.

In addition, for each attribute A of a structured type T, SQL provides the following:

- one *observer function* (*getter*) named A, which returns the value of A for the instance on which it is invoked;
- one *mutator function* (*setter*) named A, having as a parameter a value v for the attribute A; this method returns a new instance for T, obtained from that on which the method is invoked, setting the value of A equal to v.

The state of an instance of type T can be modified thanks to these methods (but not only).

Methods in SQL-3

Example:

Consider the following table:

```
CREATE TABLE Customer (
    Code            DECIMAL(4)
    Name            VARCHAR (20)
    Surname   VARCHAR (20),
    Residence t_residence);
```

The following command uses the mutator method t_residence to update the street number of residence of all customers:

```
UPDATE Customer
SET residence = residence.no('15')
```

Methods in SQL-3

Example (cont):

While the following will use the observer method to access the content of a component of the residence column:

```
SELECT residence.city()
FROM Customer
```

Because the observer method has no explicit parameters it can be invoked without the use of parentheses:

```
SELECT residence.city
FROM Customer
```

The *dot notation* then corresponds to a sequence of calls of observer methods.

Methods in SQL-3

In addition to the default methods offered by ORDBMS, we may define others that are specific for the application domain.

They should be reported at the end of the list of attributes and their types, according to the following syntax:

CREATE TYPE < type name > AS
<Representation>
attributes and their types
< list of specifications of methods >

Methods in SQL-3

The methods can be classified as:

- **Instance methods**. Methods are invoked on every instance of the type, using the dot notation.
- **Static methods**. They are methods which can be called on the same type, preceded by '::'.
- **Constructor Methods**. They can be invoked directly using the keyword NEW, they must have the same name of the type and allow us to redefine the default constructor.

In SQL: 2003, the signature of a method is presented according to the following syntax:

Methods in SQL-3

[INSTANCE | STATIC | CONSTRUCTOR] ← default INSTANCE
METHOD <method name> <parameters list>
[RETURNS <result type>]
[SELF AS RESULT]
[<Method characteristics>]
where:

<parameters list> is a list of SQL variable declarations, separated by commas. Each parameter can be preceded by the keyword IN if the parameter is read-only (default), OUT, If the parameter is write-only, or INOUT if the parameter can be either read or written from the procedure. If the method is a constructor or an instance method, this list implicitly contains a parameter SELF, which represents the instance on which the method is invoked.

Methods in SQL-3

- <result type> is the type of the value returned by the method, if defined as a function; in the case where the method is of CONSTRUCTOR type,<Result type> must coincide with the type for which the method is defined and the clause SELF AS RESULT must be specified to indicate that the result will replace the instance on which the method is invoked.
- <method characteristics> it allows us to specify additional information, including the language used for the method body.

The body of the method must then be defined with a separate command, and can be written in SQL/PSM or other programming languages.

Methods in SQL-3

In SQL / PSM, the syntax for the definition of the body of a method is the following:

CREATE [INSTANCE | STATIC | CONSTRUCTOR]
METHOD <method name> <parameters list>
[RETURNS <result type>]
FOR <type name>
<method body>

where <type name> is the name of the type, simple or structured, to which the method is attached.

Methods in SQL-3

Example:

```
CREATE TYPE  t_film AS
    (Title VARCHAR (30),
    Director VARCHAR (20),
    year DECIMAL (4),
    type CHAR (15),
    evaluation NUMERIC (3,2))

CONSTRUCTOR METHOD t_film
    (title VARCHAR (30),
    director VARCHAR (20),
    year DECIMAL (4),
    type CHAR (15),
    evaluation NUMERIC (3,2))
    RETURNS t_film
    SELF AS RESULT,
INSTANCE METHOD changeVal ()
    RETURNS t_film;
```

```
CREATE CONSTRUCTOR METHOD t_film
    (title VARCHAR (30),
    director VARCHAR (20),
    year DECIMAL (4),
    type CHAR (15),
    evaluation NUMERIC (3,2))
    RETURNS t_film
    FOR t_film
BEGIN
    SELF.Title = title;
    SELF.Director = director;
    SELF.Year = year;
    SELF.Type = type;
    SELF.Evaluation = evaluation;
    RETURN      SELF;
END;
```

Methods in SQL-3

Example:

```
CREATE TYPE t_film AS
    (Title VARCHAR (30),
    Director VARCHAR (20),
    year DECIMAL (4),
    type CHAR (15),
    evaluation NUMERIC (3,2))

CONSTRUCTOR METHOD t_film
    (Title VARCHAR (30),
    Director VARCHAR (20),
    year DECIMAL (4),
    type CHAR (15),
    evaluation NUMERIC (3,2))
    RETURNS t_film
    SELF AS RESULT,
INSTANCE METHOD changeVal ()
    RETURNS t_film;
```

```
CREATE INSTANCE METHOD changeVal ()
    RETURNS t_film
    FOR t_film
BEGIN
    IF SELF.year = 2006 THEN
        SET SELF.evaluation = 4.00;
    END IF;
    RETURN SELF;
END;
```

Methods in SQL-3

Example:

The following command shows an example of use of the new constructor during the insertion of a film in the table Video of which the definition is used:

```
CREATE TABLE Video (
    colloc          DECIMAL (4),
    movie           REF(t_film) NOT NULL,
    type            CHAR NOT NULL DEFAULT 'd');
```

```
INSERT INTO Video
VALUES (1130, NEW t_film ( 'Charlie and the Chocolate Factory', 'tim burton', 2005
    'fantastic', 3:00), 'D');
```

while the following command updates the evaluation of the film just entered:

```
UPDATE Video
SET film = film->changeVal()
WHERE colloc = 1130;
```

Abstract Types in SQL-3

Here an example of a method defined in Java:

```
CREATE TYPE AS type_address (
    Street       VARCHAR (45),
    city VARCHAR (25),
    PostalCode         CHAR (5)
)
METHOD nr() RETURNS CHAR (8);

CREATE METHOD FUNCTION nr() RETURNS CHAR (8)
FOR type_address AS
EXTERNAL NAME 'x/y/nr_civ.class' LANGUAGE 'java';
```

The code (.class) is stored in the file specified in the path.

Inheritance in SQL-3

In SQL-3 it is possible specify ***inheritance*** both among ***types and*** among ***tables***.

The inheritance among types allows a (sub-) type to be defined starting from the definition of another (super-)type. The sub-type inherits the attributes and methods of the super-type and, additionally, can define their own.

The super-type is unique. Inheritance is single.

The syntax for defining a sub-type is as follows:

CREATE TYPE <type name> UNDER <super-type name> AS (<representation>)

- [INSTANTIABLE | NOT INSTANTIABLE]
- [FINAL | NOT FINAL]

<List of methods specific>

Type inheritance

- [NOT] INSTANTIABLE indicates that the type that is created can (not) be instantiated. The default is instantiable.
- [NOT] FINAL indicates that the type that is created can not (or can) be used as a supertype for defining new types. The default is NOT FINAL.

The NOT FINAL clause is mandatory if the statement does not specify a superclass. So in many of the examples given above it had to be specified.

```
CREATE TYPE AS t_film
    (title VARCHAR (30),
    director VARCHAR (20),
    year DECIMAL (4),
    type CHAR (15),
    evaluation NUMERIC (3,2))
NOT FINAL
```

Overriding

The sub-type can also redefine (overriding) instance methods inherited. In this case the definition of the method must be preceded by the keyword OVERRIDING in the sub-type.

Example:

```
CREATE TYPE t_customer AS  (CodCus DECIMAL (4),
    name VARCHAR (20),
    surname VARCHAR (20),
    phone          CHAR (15),
    DOB            DATE,
    Residence t_residence )
INSTANTIABLE
NOT FINAL
INSTANCE METHOD score() RETURNS NUMERIC (5,2);
```

Overriding

Example:

```
CREATE TYPE t_standard UNDER t_customer AS
        (remainingPoints INTEGER)
NOT FINAL
OVERRIDING INSTANCE METHOD score() RETURNS
   NUMERIC (5,2);


CREATE TYPE t_VIP UNDER t_customer AS
   (Bonus NUMERIC (5,2))
NOT FINAL
OVERRIDING INSTANCE METHOD score () RETURNS
   NUMERIC (5,2);
```

Overriding

Example:

```
CREATE INSTANCE METHOD score() RETURNS NUMERIC (5,2)
FOR t_customer
BEGIN
    RETURN (SELECT COUNT (*)
                        FROM renting
                        WHERE codCus = SELF.codCus);
END;
```

In the case in which the customer has not yet been classified, the score corresponds to the number of rentals made; for standard customers, is the number of remaining points, and in case of VIP clients corresponds to their bonuses.

Overriding

Example:

```
CREATE INSTANCE METHOD score() RETURNS NUMERIC (5,2)
FOR t_standard
BEGIN
    RETURN SELF.remainingPoints();
END;

CREATE INSTANCE METHOD score() RETURNS NUMERIC (5,2)
FOR t_VIP
BEGIN
    RETURN SELF.bonus;
END;
```

Substitution Principle

In SQL, the principle of substitution is valid. An instance of one type can be used wherever one might expect an instance of its super-type.

This principle is not valid for typed tables.

In SQL: 2003, you can not place an instance of a sub-type in a typed table based on the super-type.

Example:

```
CREATE TYPE T2 UNDER T1
CREATE TABLE Tab1 OF T1
```

Instances of T2 can not be inserted in Tab1.

This is a difference with respect to OODBMSs, in which the substitution principle applies (polymorphism by inclusion).

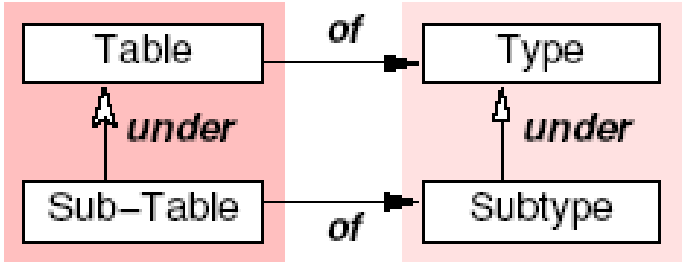
Inheritance of tables

Why this limitation? Because the schema of a typed table always coincides with the format of their row type.

To manage instances of T1 and T2 is therefore necessary to create two typed tables. In this way, however, the contents of the two tables is independent, although the types are connected by an inheritance relationship.

it is possible to solve this problem in SQL-2003 by defining inheritance between typed tables.

Condition: If Tab2 is a sub-table of Tab1, then the type T2 of Tab2 must be a sub-type of the type T1, on which Tab1 is defined.



Inheritance of tables

The inheritance between tables is defined with the command:

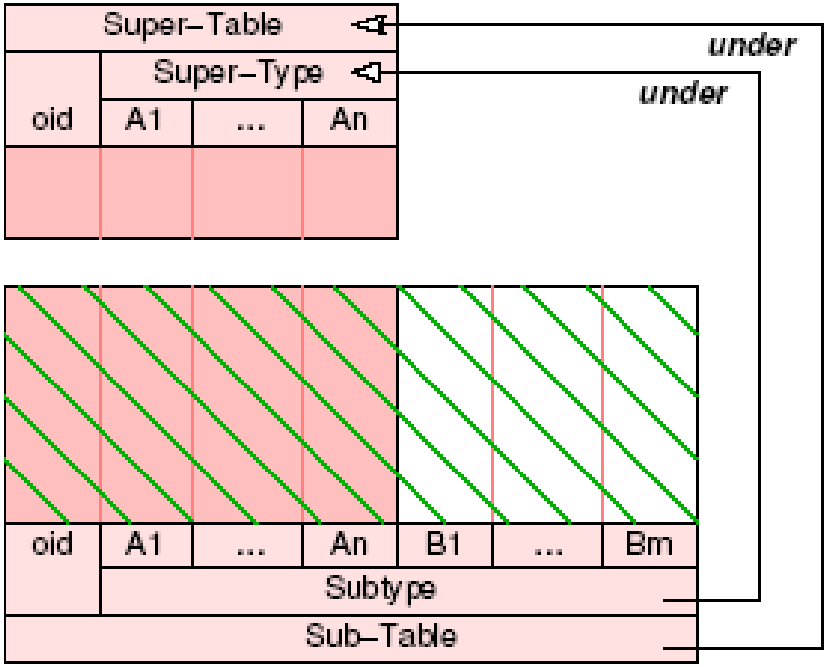
CREATE TABLE <table name> OF <type name>
UNDER <Super-table name>

Example:

CREATE TABLE Customers OF t_customer;
CREATE TABLE Standard OF t_standard UNDER Customers;
CREATE TABLE VIP OF t_VIP UNDER Customers;

In the Customers table we can not store instances of types t_standard and t_VIP. However, as a hierarchy has been defined between the tables, tuples contained in the Standard and VIP tables will be visible also in the Customers table.

Inheritance of tables



This affects the results of queries. A query made on a table automatically propagates to subtables. The same applies to the cancellation and modification operations, while an insertion operation involves only a specific table

Inheritance of tables

Example:

```
SELECT c.phone, c.score()
FROM Customers c;
```

Which implementation of c.score()?

Inheritance of tables

Example:

SELECT c.phone, c.score()
FROM Customers c;

The query is executed on the union of tuples of all three tables.

For each customer c, implementation to be used for the scoring method depends on the specific type of c. The late (or dynamic) binding is therefore used.

If we want to restrict the operation to the instances of a certain table we must use the ONLY qualifier in the FROM clause.

SELECT c.phone, c.score()
FROM ONLY Customers c;

Inheritance of tables

Example:
SELECT c.phone, c.score() FROM Customers c;

is equivalent to:

```
SELECT c.phone, c.score()
FROM ONLY Customers c;
UNION
SELECT c.phone, c.score()
FROM ONLY Standard c;
UNION
SELECT c.phone, c.score()
FROM ONLY VIP c;
```

Inheritance of tables

Example:

nome	id	data_di_nascita	indirizzo
Smith	74	16/8/68	
John	86	3/2/48	

Persone

nome	id	data_di_nascita	indirizzo	stipendio ….
Allen	82	9/7/67		30ml
Mark	81	3/5/58		60ml

Insegnanti

SELECT name
FROM Persons
WHERE DOB> 01/01/1967;

Inheritance of tables

Example:

nome	id	data_di_nascita	indirizzo
Smith	74	16/8/68	
John	86	3/2/48	

Persone

nome	id	data_di_nascita	indirizzo	stipendio ….
Allen	82	9/7/67		30ml
Mark	81	3/5/58		60ml

Insegnanti

SELECT name
FROM Persons
WHERE DOB> 01/01/1967;

The result will be: Smith and Allen

Inheritance of tables

Example:

nome	id	data_di_nascita	indirizzo
Smith	74	16/8/68	
John	86	3/2/48	

Persone

nome	id	data_di_nascita	indirizzo	stipendio ….
Allen	82	9/7/67		30ml
Mark	81	3/5/58		60ml

Insegnanti

```
SELECT name
FROM Persons
WHERE DOB> 01/01/1967;
```

The result will be: Smith and Allen

```
SELECT name
FROM ONLY Persons
WHEREDOB > 01/01/1967;
```

Inheritance of tables

Example:

nome	id	data_di_nascita	indirizzo
Smith	74	16/8/68	
John	86	3/2/48	

Persone

nome	id	data_di_nascita	indirizzo	stipendio ….
Allen	82	9/7/67		30ml
Mark	81	3/5/58		60ml

Insegnanti

```
SELECT name
FROM Persons
WHERE DOB> 01/01/1967;
```

The result will be: Smith and Allen

```
SELECT name
FROM ONLY Persons
WHEREDOB > 01/01/1967;
```

The result will be: Smith

Inheritance of tables

Example:

nome	id	data_di_nascita	indirizzo
Smith	74	16/8/68	
John	86	3/2/48	

Persone

nome	id	data_di_nascita	indirizzo	stipendio ….
Allen	82	9/7/67		30ml
Mark	81	3/5/58		60ml

Insegnanti

```
DELETE FROM Persons
WHERE id> 80;
```

Inheritance of tables

Example:

nome	id	data_di_nascita	indirizzo
Smith	74	16/8/68	
John	86	3/2/48	

Persone

nome	id	data_di_nascita	indirizzo	stipendio ….
Allen	82	9/7/67		30ml
Mark	81	3/5/58		60ml

Insegnanti

DELETE FROM Persons
WHERE id> 80;

Will delete John from the table Persons and Allen and Mark from the table Teachers.

Unstructured Complex objects

SQL-3 has new data types for large objects (LOB, large Objects).

There are two variants:

- For binary large objects (BLOBs *binary large object*)
- For large character object (CLOB, *character large object*)

SQL proposes the use of LOB within the DBMS without the need to use external files.

On the LOB not all the operations are permitted (for example, arithmetic comparisons, group by and order by are not permitted), but the LIKE comparison, the concatenation, the extraction of a substring, etc. are permitted

SQL / MM is the separate standard proposed for the management of multimedia data.

Inheritance: SQL: 2003 vs. Oracle 9i

SQL 2003

- single inheritance
- In terms of types and tables
- FINAL / NOT FINAL only for types
- Overriding of methods, not attributes
- No substitution pronciple at the level of row-types

Oracle since version 9i

- single inheritance
- A level types (no tables)
- FINAL / NOT FINAL for types and methods
- Overriding of methods, not attributes
- Substitution principle for row-types and at the attribute-level

The implementations of the Oracle objects

In Oracle, since version 8, the following concepts have been introduced:

- Abstract data types;
- Collections (such as variable length arrays and nested tables);
- Row objects;
- Object views;
- Inheritance (of types).

The abstract data types

The abstract data types (***ADT Abstract Data Type***) are types of user-defined data.

Example: To store information concerning addresses, instead of separately define the columns, we can create an abstract data type that contains these columns:

```
CREATE TYPE AddressTY AS OBJECT
(  Street          VARCHAR (100),
   City            VARCHAR (40),
   Province  CHAR (2))
```

The abstract data types

You can define an abstract data type that makes reference to another:

```
CREATE TYPE PersonTY AS OBJECT
(  Name          VARCHAR (50),
   Address       AddressTY)
```

The abstract data types builders

When we create AddressTY, no method is assigned to it (at least explicitly), but the system still defines the method *constructor* which is used to to *instantiate* an object of type AddressTY. The constructor takes the same name of the abstract data type.

The abstract data types builders

Example: To enter data in the table

```
CREATE TABLE Students
(  EnrollmentNo  CHAR (4),
   Person        PersonTY)
```

Constructors of PersonTY and AddressTY should now be used:

```
INSERT INTO Students VALUES
( '2345', PersonTY( 'Rossi Mario',
        AddressTY( 'Via Morgagni 57', 'Vaglia', 'FI')
        )
)
```

Navigation between objects

To select an attribute of an instance of an ADT it is necessary to use *dot notation*, indicating first the name of the abstract data, followed by the dot operator and attribute name.

```
UPDATE Student
SET Person.Address.Province = 'TO'
WHERE Person.Address.Province = 'MI'
```

The methods in Oracle

In the definition of an abstract data type, we can include methods operating on its attributes.

```
CREATE TYPE StudentTY AS OBJECT
(  Name                    VARCHAR (50),
   NoExams        NUMBER,
   MEMBER FUNCTION
   NoExamsToGive(NoExams IN NUMBER)
   RETURN NUMBER,
   PRAGMA RESTRICT_REFERENCES (WNDS)
)
```

The methods in Oracle

In method declaration part.

MEMBER FUNCTION gives the name NoExamsToGive to the member method of the type StudentTY.

- It is written MEMBER FUNCTION because the method returns as output a value; otherwise we would have written MEMBER PROCEDURE.

The methods in Oracle

- IN NUMBER indicates that when the method is invoked, for the parameter NoExams, of type NUMBER, a value must be specified. In the case in which the method returns a value through the parameter, we will use the qualifier OUT instead of IN.
- RETURN NUMBER indicates that the function returns a value of type NUMBER.
- PRAGMA RESTRICT_REFERENCE specifies restrictions which apply to the method; example: WNDS (*Write No Database*) indicates that the method can not modify the database. There are other more stringent restrictions as RNDS (*Read No Database*).

The methods in Oracle

To define the method we use the command CREATE TYPE BODY:

```
CREATE TYPE BODY StudentTY AS
MEMBER FUNCTION ExamsToGive(NoExams
   NUMBER) RETURN NUMBER IS
BEGIN
        RETURN 25 - NoExams
END;
END
```

The methods in Oracle

New methods can be created within the same command, separated by a semicolon, before the final clause END. These methods may be used on tables based on the type StudentTY.

Suppose that StudenteTY is used as a data type for a column whose name is Student in the table Enrolled, whose structure is as shown below.

```
CREATE TABLE Enrolled
   (    EnrollmentNo      CHAR (4),
        Student      StudentTY)
```

The methods in Oracle

We perform on the table an insertion operation.

```
INSERT INTO Enrolled VALUES
        ( '2345', StudentTY ( 'Mario Rossi', 15))
```

You can call the function ExamsToGive which is part of the given type StudentTY, with the same notation used for attributes:

```
SELECT   Student.ExamsTOGive(Student.NoExams)
FROM Enrolled;
```

The function will return 10.

The object views in Oracle

The ***object views*** allow us to get an *object relational* perspective of a purely relational DB.

Suppose for example that the table Student already exists and has been created as a relational table:

```
CREATE TABLE Student
(  EnrollmentNo        CHAR (4) PRIMARY KEY,
   Name              VARCHAR (50),
   Street                VARCHAR (100),
   City              VARCHAR (40),
   Province  CHAR (2))
```

The object views in Oracle

You can still create data types AddressTY and PersonTY and use object views to correlate these types of data to the table Student. This opportunity is particularly useful if we intend to extend the DB towards an object-relational perspective.

```
CREATE VIEW StudentOV(EnrollmentNO, Person)
  AS
SELECT EnrollmentNO,
  PersonTY(Name, AddressTY(Street, City,
  Province))
FROM Student
```

The object views in Oracle

The view object StudentOV is composed by two columns EnrollmentNo and Person; the last of type PersonTY. Optionally, we can also create an abstract data type including all the necessary attributes:

```
CREATE TYPE StudentTY AS OBJECT
(  EnrollmentNo          NUMBER,
   Person                PersonTY)
```

and base the object view on it.

The object views in Oracle

In Student, insertions may be operated by treating Student as a traditional relational table:

```
INSERT INTO Student VALUES
( '2345', 'Russo Calogero', 'Via Catania 67', 'Acitrezza', 'CT')
```

Alternatively, by means of the StudentOV object view, by using the constructor methods:

```
INSERT INTO StudentOV VALUES
( '3000',
   PersonTY ( 'Bianco Rosalia',
   AddressTY ( 'Via Caldera 88', 'Catania' 'CT'))
)
```

The object views in Oracle

With the availability of the two possibilities, every application will perform the processing of data according to its own standards.

If the database will evolve towards the extensive use of abstract data types, you still can make entries by following this mode *independently* that abstract types were created before or after the corresponding tables.

The object views in Oracle

The use of object views has two ***advantages***:

- It allows to apply the abstract data types also to existing tables, so it improves the level of standardization of applications and the ability to reuse existing parts of the database;
- The methods defined for abstract data types can be applied to the data of the new tables, and to those contained in existing tables.

Arrays of variable length

A ***variable length array*** (VARRAY) is an aggregation, that is, a set of objects of the same type, which within a table is treated as a column.

CREATE TYPE AspectsVA AS VARRAY(10) OF VARCHAR(25)

The command creates the type AspectsVA. The AS VARRAY (10) clause indicates that it can contain, *at most,* ten values.

Arrays of variable length

The array can be used in creating a table or another abstract data type:

```
CREATE TABLE products
(  Art_Cod          CHAR (4),
   Cat_Cod          CHAR (4),
   Description      CHAR (40),
   Aspects          AspectsVA,
   Price            NUMBER,
   VAT              NUMBER,
   ShippingCosts NUMBER)
```

Arrays of variable length

The system creates for the array of variable length a constructor method, which must be used for the insertion of values:

```
INSERT INTO products VALUES
(  'T100', 'T10', 'Round Table',
   AspectsVA('Light Cherry', 'African Walnut',
                  'Mahogany'),
   255, 20, 0)
```

Arrays of variable length

The insertion has entered three values, seven remain unused; we can specify these values as NULL in the INSERT command:

```
INSERT INTO products VALUES
(  'T100', 'T10', 'Round Table',
        AspectsVA ( 'Light Cherry', 'African Walnut',
                    'Mahogany',
   NULL, NULL, NULL, NULL, NULL, NULL, NULL),
255, 20, 0)
```

It is the user's responsibility to check that the number of values to be inserted does not exceed the capacity of the array.

Arrays of variable length

If the array was based on an abstract data type:

```
CREATE TYPE AspectTY AS OBJECT
(Description  VARCHAR(25))

CREATE TYPE AspectsVA AS VARRAY(10) OF AspectTY
```

it would have been necessary to nest the call to AspectTY inside AspectsVA:

```
INSERT INTO products VALUES
(   'T100', 'T10', 'Round Table',
    AspectsVA(AspectTY( 'Light Cherry'),
                    AspectTY( 'African Walnut'),
                    AspectTY( 'Mahogany')),
255, 20, 0)
```

Arrays of variable length

The query operations on arrays of variable length can not be made with a simple SELECT command.

To make any query on the array the PL / SQL language should be used, which provides instructions for control of cycles.

Nested tables in Oracle

A ***nested table*** is a table inside another table.

For each row in the main table, the nested table can contain multiple rows. In a sense, it is a way to store multiple relationships (1: N) within a table. As that between classes and students in a course.

```
CREATE TYPE StudentTY AS OBJECT
   ( EnrollmentNo        CHAR (4),
     Name              VARCHAR (50))
CREATE TYPE StudentsNT AS TABLE OF
   StudentTY
```

Nested tables in Oracle

The clause AS TABLE OF of the command CREATE TYPE specifies that this data type is used as a basis for a nested table.

```
CREATE TABLE course
   (title        VARCHAR (25),
    Students                    StudentsNT)
NESTED TABLE Students STORE AS
                        StudentsNT_TAB
```

In the command of Course table creation, we shall indicate the name of an additional table where the data of the nested table will be actually stored. In fact, these data are not stored within the main table, but separately.

Nested tables in Oracle

The integrity between the two tables is handled internally to Oracle via pointers.

Unlike the arrays of variable length, nested tables support SQL queries. However, you can not simply write:

```
SELECT EnrollmentNo  - wrong
FROM StudentsNT_TAB
WHERE name = 'Lucia Bianco'
```

Nested tables in Oracle

To select columns from the nested table as EnrollmentNo, it is first required to flatten the table, so that we can query it. For this purpose, in Oracle 8, the function **THE** is available.

We must select the column of interest (Students) from the main table (Course), enclose it within the function THE and assign an alias (XX), and then perform the query on the columns of the nested table using the alias previously defined in the selection list (XX.EnrollmentNo) as the table name in the WHERE clause (XX.Name):

Nested tables in Oracle

```
SELECT XX.EnrollmentNo
FROM THE(SELECT Students
        FROM Course
        WHERE Title = 'Analytic Geometry') XX
WHERE XX.Name = 'Lucia Bianco'
```

Nested tables in Oracle

The function THE must be used whenever you need to INSERT and UPDATE directly into the nested table.

For example, to add a student to the analytic geometry course:

```
INSERT INTO
   THE(SELECT Students
       FROM Course
       WHERE Title = 'Analytic Geometry')
VALUES (StudentTY ( '3111', 'Alessandro Del
                Pierino'))
```

Nested tables in Oracle

NOTE: the THE operator was introduced in Oracle 8. In the next versions of Oracle it was replaced by the expressions TABLE. The use of THE is now *deprecated*.

```
SELECT S.EnrollmentNo
FROM Course C, TABLE(C.Students) S
WHERE C.Title = 'Analytic Geometry' AND
   S.Name = 'Lucia Bianco'
```

Nested tables in Oracle

So far we have seen how to manipulate the data within the nested table, but what happens if we want to work on the main table?

If one attempts to directly enter the values of the columns of the nested table with an INSERT command in the main table, he gets an error because the main table contains one column for each nested table.

To get the right result we have to use the operators CAST and MULTISET in the form:

CAST (MULTISET (<subquery>) AS <TypeNT>)

MULTISET applied to the subquery allows to return more than one row as a result.

Nested tables in Oracle

If not specified, the subquery would be treated as a scalar.

On the rows returned by MULTISET, CAST makes a ***casting***, which is a result of the conversion to TypeNT, that is, the abstract data type of a nested table.

```
INSERT INTO Course VALUES
(   'Databases',
    CAST (MULTISET (
                SELECT *
                FROM THE ( SELECT Students
                           FROM Course
                           WHERE Title =
                                        'Analytic geometry'))
                AS StudentsNT))
```

Nested tables in Oracle

Note: the CAST and MULTISET operators are also deprecated. From version 9 on, for the insertion of the data, it is necessary to use the constructors for the data type of the nested table.

```
INSERT INTO Course VALUES
(  'Analytic geometry',
StudentsNT(StudentsTY ( '2345', 'Calogero Russo'),
                StudentsTY ( '3000', 'Lucia Bianco'),
                StudentsTY ( '2999', 'Maria Caputo'))
)
```

Nested tables in Oracle

To check whether the insertion command worked properly, we can make a selection on the table Course to see students contained in the nested table where the title is Databases:

```
SELECT NT.Name
FROM THE (SELECT Students
                  FROM Course
                  WHERE Title= 'Databases') NT
```

or, from version 9 onwards:

```
SELECT NT.Name
FROM Course C, TABLE(C.Students) NT
WHERE C.Title = 'Databases '
```

Nested tables in Oracle

The query result will be:

Calogero Russo
Lucia Bianco
Maria Caputo
Alessandro Del Pierino

Nested tables in Oracle

Qualitative considerations on the data model. The relational tables can be easily related to other relational tables through the use of foreign keys; if the data is related to several other tables it may be advised not to nest the data within the table.

Considerations on performance. If the number of rows of a nested table is high, we might have performance problems as it is not possible to index them. On the other hand, the fact that the nested tables are stored separately from the main table can improve query performance by distributing the read / write data across multiple separate tables.

Row objects in Oracle

The ***row objects*** are referenced objects that are accessible by others through objects references (Object identifiers).

In an ***object table***, each row is a row object and has an identification value - ***OID*** (***object identifier***) – assigned when the row is created.

```
CREATE TYPE DegreeProgramTY AS OBJECT
 ( code          CHAR (3),
   name          VARCHAR (25))
```

Row objects in Oracle

The command:

CREATE TABLE DegreeProgram OF DegreeProgramTY

creates the object table DegreeProgram which is based on the abstract data type DegreeProgramTY. The table appears in all respects as a normal relational table, but each line has its own associated OID.

To insert the rows in the table one should use the constructor method of the corresponding data type:

INSERT INTO DegreeProgram VALUES DegreeProgramTY ('010', 'Theoretical')

Row objects in Oracle

To select names from the object table DegreeProgram we can simply make use of a direct query about attributes of the data type:

```
SELECT name
FROM DegreeProgram
```

At the time of insertion of a row in an object table, the row is assigned an OID. To view these OIDs we can use the operator REF:

```
SELECT REF(DP)
FROM DegreeProgram DP
WHERE DP.name = 'Theoretical'
```

Row objects in Oracle

The result will be something difficult to understand, such as:

00BFF569FD.

We must then use another operator, DEREF, to convert the result of REF in values.

Example:

```
CREATE TABLE Student
(Name                          VARCHAR (50),
FollowedDegreeProgram          REF DegreeProgramTY)

INSERT INTO Student
   SELECT 'Mario Rossi', REF(DP)
   FROM DegreeProgram DP
   WHERE DP.name = 'Theoretical'
```

Row objects in Oracle

```
SELECT DEREF(S.FollowedDegreeProgram)
FROM Student S
WHERE name = 'Mario Rossi'
```

The query uses a reference to a row object to pass from the table Student to a second table Student DegreeProgram. Thus, the association is obtained by pointers.

The object table itself is not mentioned in the query. The only listed table is Student. It is not necessary to know the name of the object table to apply DEREF to its values.

Row objects in Oracle

Though DegreeProgram is a table of objects, it is possible to perform on it selection operations as if it was a relational table:

```
SELECT * FROM DegreeProgram
```

Similarly, you can delete a object:

```
DELETE FROM DegreeProgram
WHERE name = 'Theoretical'
```

If some lines of Students do reference to an object removed in DegreeProgram, as seen in the example of the student Mario Rossi, they will have a wrong REF.

Row objects in Oracle

The OID will not be reused: If a new row will later be inserted with value “Theoretical”, this will have a new OID and will not be accessed from the rows of Student containing the deleted OID (as in the case of Mario Rossi).

It is therefore necessary to check database updates in order to maintain referential integrity of pointers.

Type inheritance

With version 8 and later, in Oracle we can create types (better, ***subtypes***) by inheritance from other types (***supertypes***).

We speak, therefore, of a ***type hierarchy***.

The type inheritance supports the ***substitution*** principle. This is in the possibility of using instances of a subtype in any context in which a supertype can be used.

Type inheritance

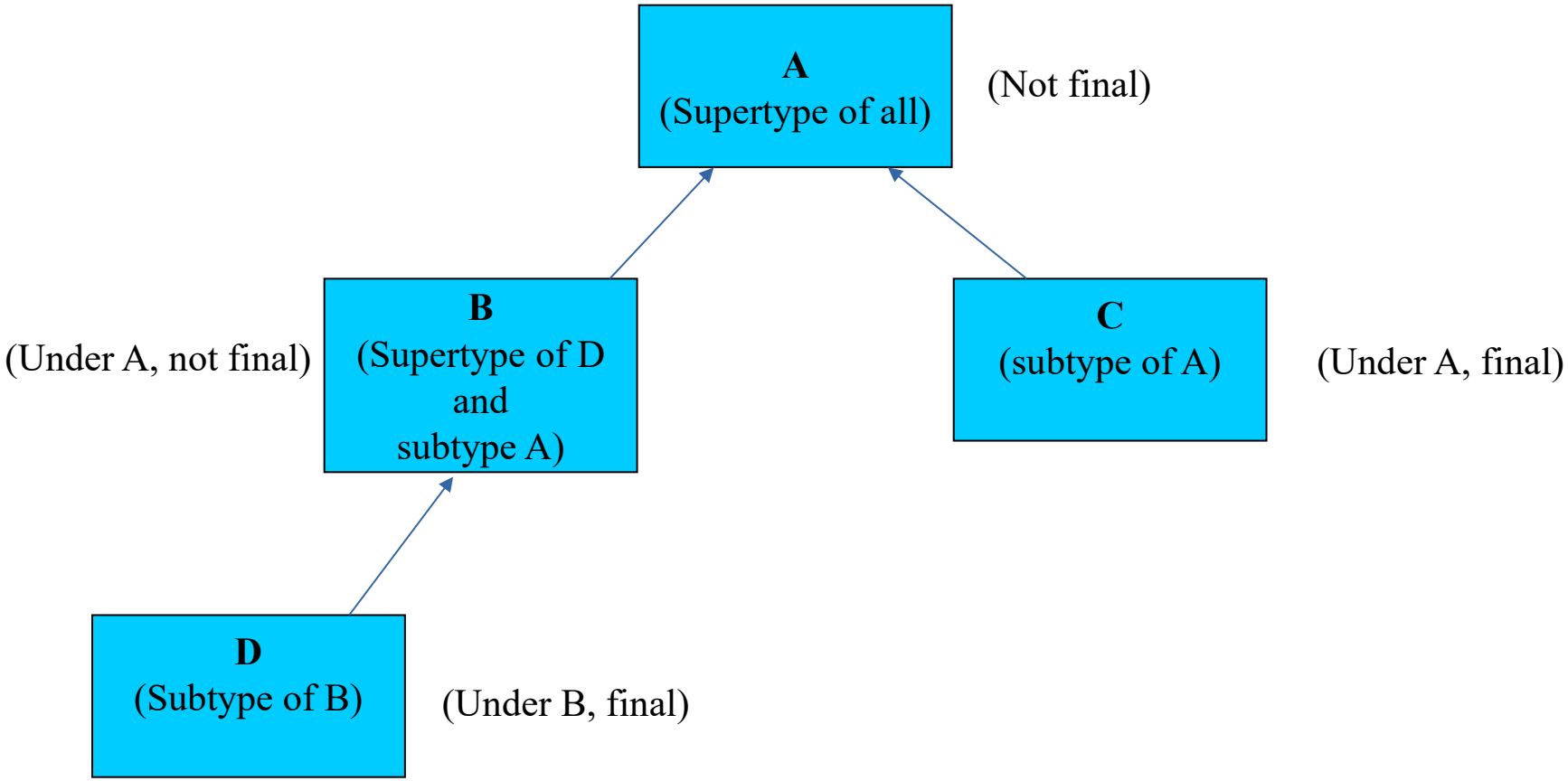
The keyword to inherit types are UNDER and NOT FINAL.

UNDER followed by the name of the supertype is used to 'communicate' Oracle that a new type having also the attributes of the supertype is being built.

NOT FINAL states that the type that we are creating will have, in turn, a subtype.

Oracle does not support the concept of multiple inheritance, but only that of single inheritance.

Type inheritance



Type inheritance

If we want to create a type *Person* (Supertype), a type *Student* (descendant of person) and a type *PartTimeStudent* (descendant of student), we have to write:

```
CREATE TYPE Person AS OBJECT
(   Tax_code  VARCHAR (15),
    Name      VARCHAR (30),
    Address   VARCHAR (100)) NOT FINAL;

CREATE TYPE Student UNDER Person
(   Faculty                VARCHAR (15),
    Course_of_study VARCHAR (30)) NOT FINAL;

CREATE TYPE PartTimeStudent UNDER Student
(   Num_hours NUMBER);
```

Type inheritance

A subtype automatically inherits, in addition to the attributes, also the methods declared in the supertype or, in turn, inherited.

We can also *add* other methods or *redefine* those inherited. The added methods may also have the same name as those inherited, but with different formal parameters in type and number. In this case we have *overloading*.

```
CREATE TYPE MyType_typ AS OBJECT (...,
   MEMBER PROCEDURE foo (x NUMBER), ...) NOT FINAL;
CREATE TYPE MySubType_typ UNDER MyType_typ (...,
   MEMBER PROCEDURE foo (x DATE) ...);
```

Type inheritance

```
CREATE TYPE MyType_typ AS OBJECT (
 ...,
 MEMBER PROCEDURE Print ()
 MEMBER FUNCTION foo (x NUMBER) FINAL ...)
NOT FINAL;

CREATE TYPE MySubType_typ UNDER MyType_typ (...,
  overriding MEMBER PROCEDURE Print (), ...);
```

Note that the *methods* may be declared FINAL or NOT FINAL. If a method is declared *final*, subtypes can not overwrite it.

By default, methods are not final.

The methods in Oracle: syntax

```
create or replace object type <object_name>
  [authid {definer | current_user}] IS object
  (  [instance_variables {sql_datatype | plsql_datatype}],

     [CONSTRUCTOR function <constructor_name>
     [( parameter_list)] return self as result,

     [{member | static} function <function_name>
     [( parameter_list)] return {sql_datatype | plsql_datatye},

     [{member | static} procedure <procedure_name>
     [( parameter_list)],

     {[map function <map_name> return {char | date | number|
varchar2} |
      [order function <order_name> return {sql_datatype |
plsql_datatype}}])

     [not] instantiable [not] final;
/
```

The methods in Oracle: syntax

```
CREATE OR REPLACE TYPE hello_there IS OBJECT
( who VARCHAR2(20),
  CONSTRUCTOR FUNCTION hello_there
    RETURN SELF AS RESULT,
  CONSTRUCTOR FUNCTION hello_there( who VARCHAR2 )
    RETURN SELF AS RESULT,
  MEMBER FUNCTION get_who RETURN VARCHAR2
  MEMBER PROCEDURE set_who (who VARCHAR2)
  MEMBER PROCEDURE to_string )
INSTANTIABLE NOT FINAL;
```

The methods in Oracle: syntax

```
-- Object Body
CREATE OR REPLACE TYPE BODY hello_there IS

  CONSTRUCTOR FUNCTION hello_there RETURN SELF AS RESULT IS
    HELLO_THERE hello := hello_there('Generic Object.');
  BEGIN
    self := hello;
    RETURN;
  END hello_there;

  CONSTRUCTOR FUNCTION hello_there (who VARCHAR2) RETURN SELF AS RESULT IS
  BEGIN
    self.who := who;
    RETURN;
  END hello_there;

  MEMBER FUNCTION get_who RETURN VARCHAR2 IS
  BEGIN
    RETURN self.who;
  END get_who;

  MEMBER PROCEDURE set_who (who VARCHAR2) IS
  BEGIN
    self.who := who;
  END set_who;

  MEMBER PROCEDURE to_string IS
  BEGIN
    dbms_output.put_line('Hello '||self.who);
  END to_string;
END;
```

Polymorphism in Oracle

6th - polymorphism in oracle.doc

Type inheritance

Once the type Person is created, we can create tables of objects Person:

```
CREATE TABLE Persons OF Person;
```

The table may store instances of Person, but also of type Student and PartTimeStudent.

This raises the problem, during querying, to specify what type of object is returned. To retrieve the type, it is possible to use the function VALUE, which has as its parameter an alias of a table.

```
SELECT VALUE(p) FROM Persons p
WHERE p.name = "Lucia Bianco";
```

The result are objects of type Person with the name "Lucia Bianco", regardless of whether the "Lucia Bianco"s are stored as Students (or PartTimeStudent).

Type inheritance

If we want to only select students, we have to use the IS OF predicate:

```
SELECT VALUE (p)
FROM Persons p
WHERE VALUE (p) IS OF (Student);
```

In summary ...

The idea behind the object-relational model is to add features to the system of types. Then, we can still use SQL but usually:

- The columns can be of a new type (ADT)
- You can define user methods on ADT
- The columns can be of a complex type
- The reference types are allowed and you can apply the "deref 'operator'
- Inheritance of types and subtables
- Compatibility with traditional SQL schemas

All relational DBMS vendors are moving in this direction (SQL3).
Big business!

Object-Relational Data Base Design

There are no widely accepted methodologies for designing object-relational databases.

The richness of the *object relational* (OR) model and the collection of the various constructs make the logical design of an OR database challenging.

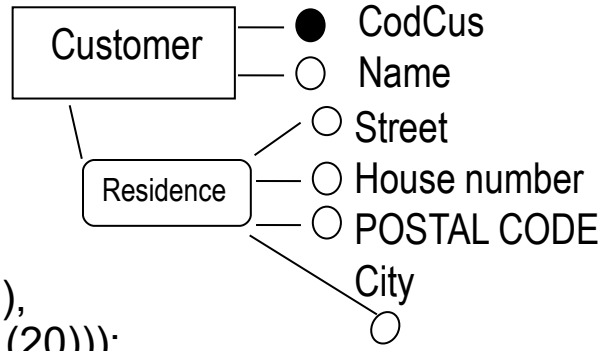
Some methodological proposals are related to a particular ORDBMS or are related to a specific domain (for example, spatial data).

In the next slides we will refer to SQL:2003.

The logical model for ORSQL: 2003 includes:

Logical model in SQL: 2003

- The row types and structured types used to define user-defined types.
  - With a row type in a column of a table we can model the composite attributes defined at the ER diagram level.




```
CREATE TABLE CUSTOMER
   (CodCus DECIMAL (4) PRIMARY KEY,
    Name VARCHAR (20),
    Residence ROW (street VARCHAR (20), Nr VARCHAR (4),
                   postal_code INTEGER, city VARCHAR (20)));
```

- Structured types are useful to define typed tables or columns of tables.

```
CREATE TYPE t_residence AS (street VARCHAR (20), nr VARCHAR (4),
                   postal_code INTEGER, city VARCHAR (20));
CREATE TABLE CUSTOMER
   (CodCus DECIMAL (4) PRIMARY KEY,
    name VARCHAR (20),
    residence t_residence);
```

Logical model in SQL: 2003

- The objects, that is rows of typed tables that include OID.
- The collection types: ARRAY and MULTISET, which are used to implement multi-valued attributes.
- The collection types and row types can be used to implement de-normalized tables.
- Nesting of ARRAY has no restrictions (but the maximum size is set by the manufacturer).
- In MULTISET one can distinguish between NULL value and empty set.
- On MULTISET, aggregate functions are defined (COLLECT, for the creation of a MULTISET by the argument value in each row of the group, MERGE, for the union of MULTISETs, INTERSECTION for the intersection).

Logical design of EER Models

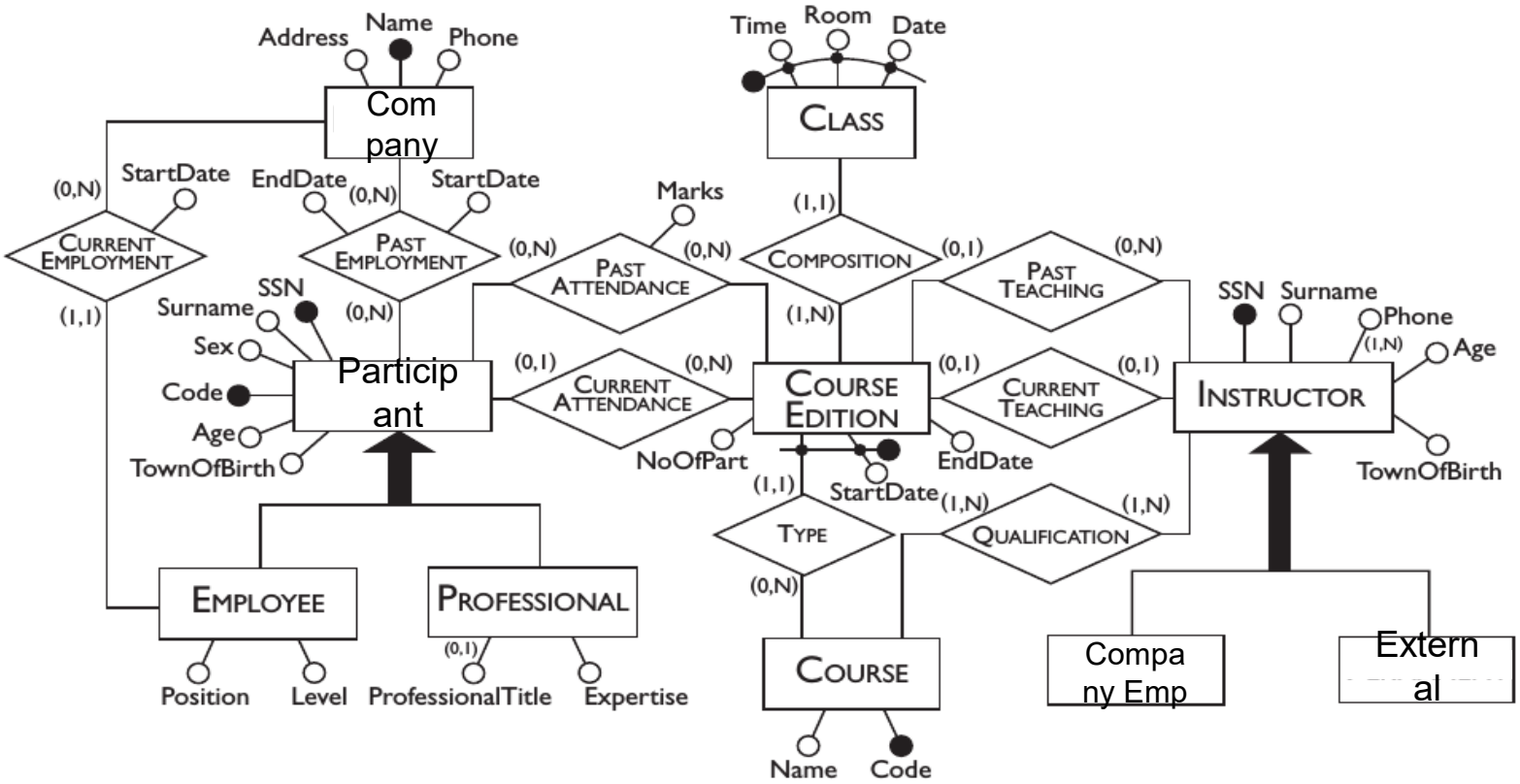
While restructuring an EER model is not necessary, it is also true that the process of translation into the OR model is generally more complex than in the relational model as entities and associations can be represented in different ways.

For example, an association 1:N can be represented as a foreign key, but also using the typed tables and reference types.

The choice of the most appropriate solution depends mainly on the queries that will be carried out and the possible need to reduce the number of explicit joins which will be carried out for them.

In the following we will show two examples of transformation, one for conceptual models expressed in EER and the other in UML.

Conceptual model



An example of logical design

We take the example of the education company.

On data described by this scheme, the following operations are provided:

Operation 1: Insert a new participant indicating all his/her data.

Operation 2: Assign a participant to a course edition.

Operation 3: Insert a new instructor indicating all his/her data and courses that can teach.

Operation 4: Assign a qualified instructor to an edition of a course.

An example of logical design

Operation 5: Print all the information about the current editions of a course, with title, class timetable and number of participants.

Operation 6: Print all courses offered, with information on instructors who can teach them.

Operation 7: For every instructor, find the participants in all the courses he taught.

Operation 8: Compute a statistics on all course participants with all the information about them, the edition which they attended and the respective mark.

Workload

Volumes Table

Concept	Type	Volume
Classes	E	8000
Course Edition	E	1000
Course	E	200
Instructor	E	300
External	E	250
Company Emp	E	50
Participant	E	5000
Employee	E	4500
Professional	E	500

Concept	Type	Volume
Company	E	500
Past attendance	R	10000
Current attendance	R	500
Composition	R	8000
Type	R	1000
Past teaching	R	900
Current teaching	R	50
Qualification	R	600
Past employment	R	400
Current employment	R	3000

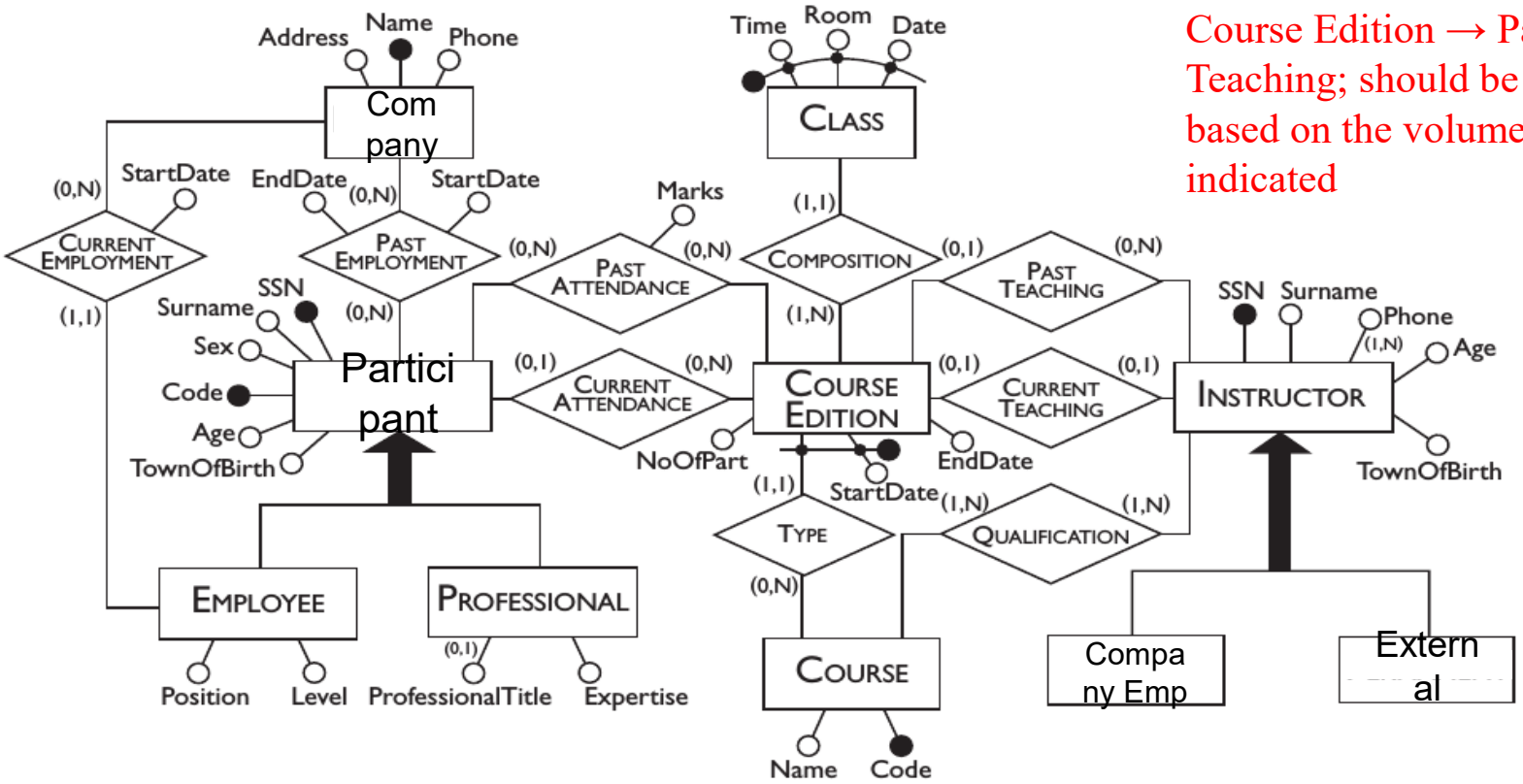
Workload

Operations Table

Operation	Type	Frequency
Op. 1	Interactive	40 / day
Op. 2	I	50 / day
Op. 3	I	2 / day
Op. 4	I	15 / day
Op. 5	I	10 / day
Op. 6	I	20 / day
Op. 7	I	5 / week.
Op. 8	Batch	10 / month

Conceptual model

* The minimum cardinalities:
Course Edition → Current attendance; Course Edition → Past attendance;
Course Edition → Past Teaching; should be reviewed based on the volumes indicated



Analysis of the redundancies

There is only one redundant data in the diagram: the attribute Number of participants in Course Edition which can be derived by the Current Attendance.

- Memory Occupancy: 4×1000 = 4000 bytes
- Operations involved: (2), (5) and (8). The (8) is infrequent and batch, so it can be neglected.

We calculate the Access tables:

Analysis of the redundancies

Accesses with redundancy

Operation 2			
Concept	Cnstr	Acc	Type
Participant	E	1	R
CurrentAtt'nce	R	1	W
CourseEdition	E	1	R
CourseEdition	E	1	W

Operation 5			
Concept	Cnstr	Acc	Type
CourseEdition	E	1	R
Type	R	1	R
Course	E	1	R
Composition	R	8	R
Class	E	8	R

Accesses without redundancy

Operation 2			
Concept	Cnstr	Acc	Type
Participant	E	1	R
CurrentAtt'nce	R	1	W

Operation 5			
Concept	Cnstr	Acc	Type
CourseEdition	E	1	R
Type	R	1	R
Course	E	1	R
Composition	R	8	R
Class	E	8	R
Current Att.nce	R	10	R

Analysis of the redundancies

With redundancy:

(2) 2×50 = 100 read accesses and
2×50 = 100 write accesses (counted twice because writing)

(5) 19×10 = 190 read accesses per day

Total: 490 accesses per day (double counting write operations)

No redundancy:

(2) 50 read accesses and
50 write accesses (counted twice because writing)

(5) 29×10 = 290 read accesses per day

Total: 440 accesses per day

In the presence of redundancy we have disadvantages both in memory and time. We decide to eliminate the redundancy from the schema.

Partitioning and merging

- Horizontal partitioning entity Course Edition?

Pros: The operation (5) applies only to *current* editions. Also relationships Current teaching and Current Participation refer only to ongoing editions.

Cons: Associations Type and Composition should be duplicated. In addition, operations (7) and (8), which do not distinguish between current and past editions, would be more expensive because they require a visit from two distinct entities.

Decision: no partitioning

Partitioning and merging

Merging of relationships Past teaching and Current teaching, as well as Past attendance and Current Attendance?

Pros The operations (7) and (8) make no difference. Also, it would not be necessary to transfer occurrences by an association to another when a course edition ends.

Cons: Presence of attribute null values in Mark. The waste would be 500×4 = 2000 bytes

Decision: merge.

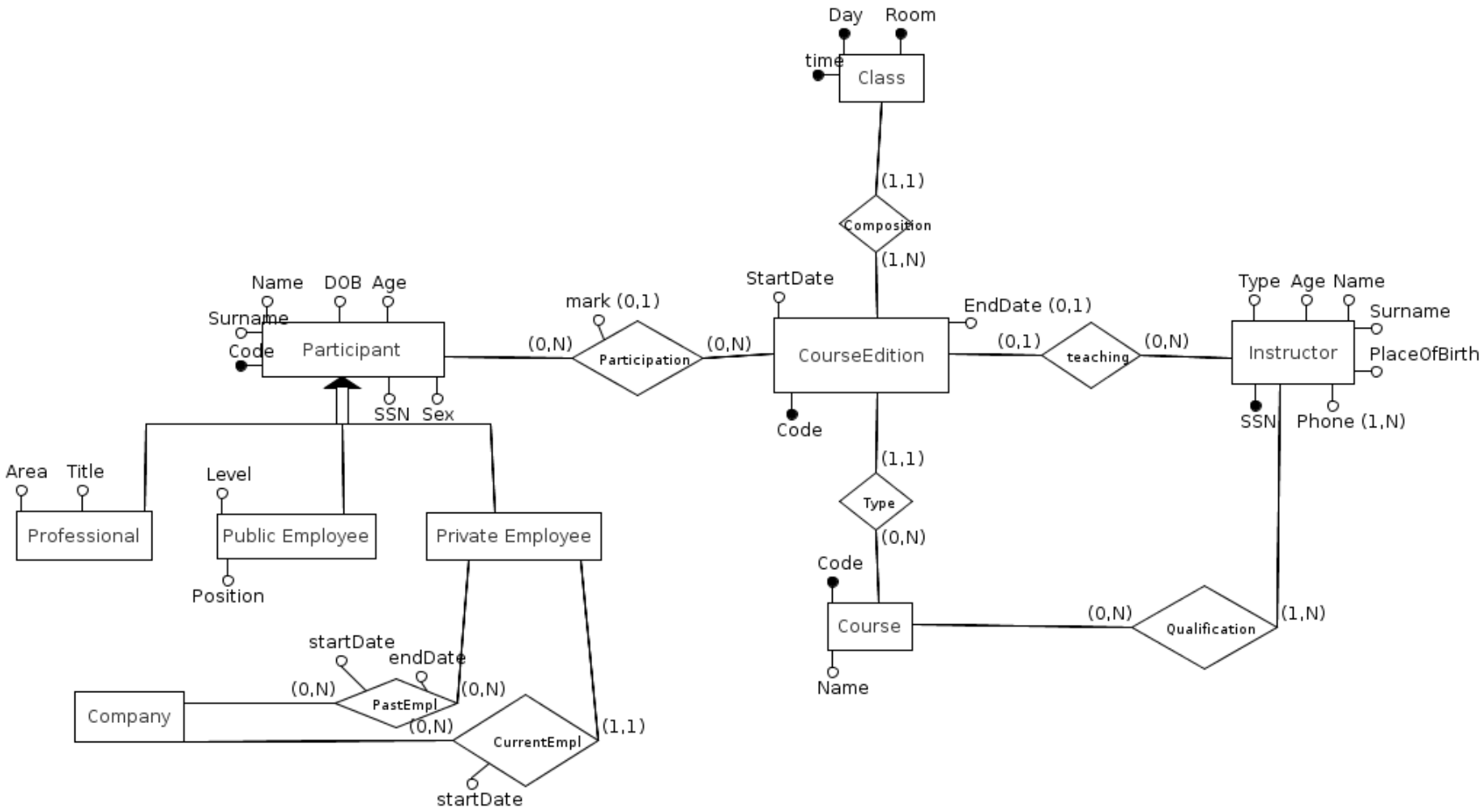
Choice of main identifiers

- Only the entities Participants has two identifiers: the tax code and the inner code. Between the two it is preferable to choose the second, because a tax code requires 16 bytes of memory, while an internal code, which serves to distinguish more than 10000 occurrences requires no more than 3 bytes (in a representation for characters).

- Warning: We have to take into account that in the OR model all objects have an OID, so it is not essential to associate each entity with an identifier.

Choice of main identifiers

The entity Course Edition has an identifier made by Start date and Course. This identifier is then used in the logical model for two associations (Participation and Teaching), with many occurrences. We may think to change the identifier of the entity by adding a Code. This would consist of the course number and an identifier of the course edition (an additional character is sufficient, because every course has five editions) on average.

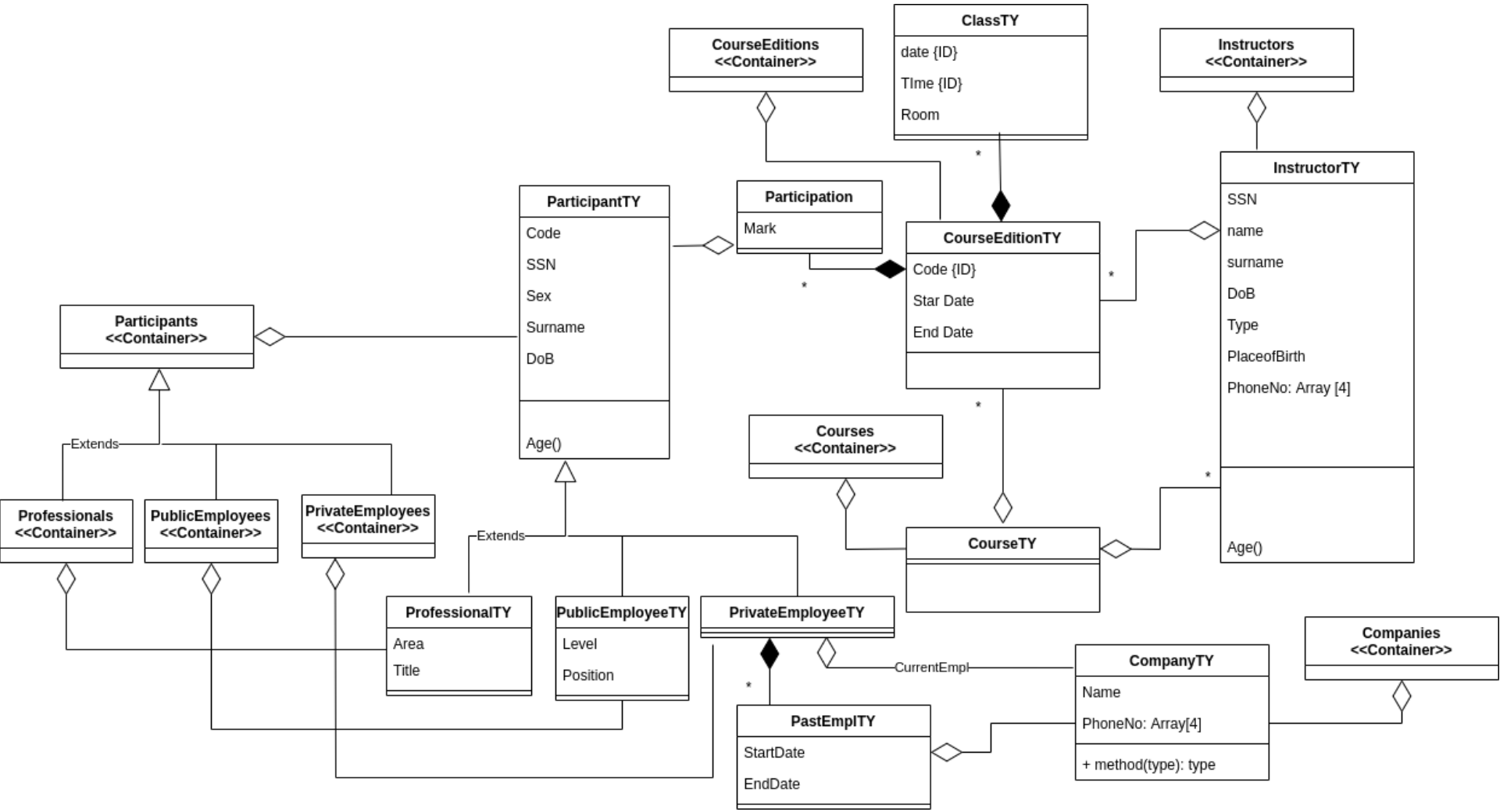

Day
Room
time
Class
(1,1)
Composition
(1,N)
Name
DOB
Age
Surname
Code
Participant
SSN
Sex
mark (0,1)
(0,N)
Participation
(0,N)
StartDate
CourseEdition
Code
EndDate (0,1)
(0,1)
teaching
(0,N)
Type
Age
Name
Surname
Instructor
PlaceOfBirth
SSN
Phone (1,N)
Area
Title
Professional
Level
Public Employee
Position
Private Employee
(1,1)
Type
(0,N)
Code
Course
Name
(0,N)
Qualification
(1,N)
startDate
endDate
(0,N)
PastEmpl
(0,N)
Company
(0,N)
CurrentEmpl
(1,1)
startDate

```
Create type PartecipantTY as (
Code char(5) primary key,
Surname char(15),
Name char(15),
DoB Date,
Sex char(1),
Age number) NOT  FINAL

Create type ProfessionalTY UNDER PartecipantTY as (
Area char(5),
Title char(5)) FINAL

Create type PublicEmployeeTY UNDER PartecipantTY as (
Level char(5),
Position char(5)) FINAL

Create type PrivateEmployeeTY UNDER PartecipantTY as (
PastEmpl REF(PastEmplTY) MULTISET,
CurrentEmpl REF(CompanyTY),
StartDateCurrentEmp Date)  FINAL

Create type PastEmplTY as (
StartDate date,
EndDate date,
Company REF(CompanyTY))

Create type CompanyTY as(
Name char (25) primary key,
Address char(30),
Phone Char(14))
```

```
Create type InstructorTY as(
   SSN char (16) primary key,
   Type char(4),
   Surname char(15),
   Name char(15),
   PlaceOfBirth char(15),
   Phone char(15) ARRAY[5],
   Teaching REF(CourseEditionTY) MULTISET
), MEMBER FUNCTION Age() retunrs number;

Create type ClassTY as(
   Day date,
   Time Time,
   Room char(4))

Create type CourseTY as(
   Code char(4) primary key,
   Name char(15),
   Editions REF(CourseEditionTY) MULTISET,
   Instructors REF(InstructorTYTY) MULTISET, )

Create type CourseEditionTY as(
   Code char(4) primary key,
   StartDate date,
   EndDate date,
   Schedule ClassTY MULTISET,
   Participations ParticipationTY MULTISET
   )


Create type ParticipationTY as (
    Participant REF(ParticipantTY),
    Mark Number)
```

Translation into the OR model

```
Create table Partecipants of PartecipantTY

Create table Professionals  of ProfessinalTY under Partecipants

Create table PublicEmployees  of PublicEmployeeTY UNDER Partecipants

Create table PrivateEmployees  of PrivateEmployeeTY UNDER Partecipants

Create table PastEmpl of PastEmplTY

Create table Companies of CompanyTY

Create table CourseEditions of CourseEditionTY

Create table Courses of CourseTY

Create table Instructors of InstructorTY

Create table Partecipants of PartecipantTY

Create table Participations of PartecipationTY
```


ClassTY
date {ID}
TIme {ID}
Room
CourseEditions
<<Container>>
Instructors
<<Container>>
InstructorTY
SSN
name
surname
DoB
Type
PlaceofBirth
PhoneNo: Array [4]
Age()
ParticipantTY
Code
SSN
Sex
Surname
DoB
Age()
Participation
Mark
CourseEditionTY
Code {ID}
Star Date
End Date
Participants
<<Container>>
Courses
<<Container>>
CourseTY
Extends
Professionals
<<Container>>
PublicEmployees
<<Container>>
PrivateEmployees
<<Container>>
ProfessionalTY
Area
Title
PublicEmployeeTY
Level
Position
PrivateEmployeeTY
CurrentEmpl
PastEmplTY
StartDate
EndDate
CompanyTY
Name
PhoneNo: Array[4]
+ method(type): type
Companies
<<Container>>
*

Transformation from EER Models

Translation of entities.

An entity can be translated in several ways:

1. With a type
2. With a non-typed table
3. With a typed table (creating the corresponding row type before the creation of the table).

The first choice may be reasonable if the entity instances do not represent fundamental objects of the domain of interest, but collateral information.

Example: address

The difference between 2. and 3. depends on how we intend to define the associations involving that entity. Using typed tables we will have the opportunity to use reference types for modeling associations.

Transformation from EER Models

The definition of a type and related typed tables is not necessary in case

- the type is not used in the context of a reference type
- the type is not used in the context of a hierarchy of inheritance
- the type does not contain methods.

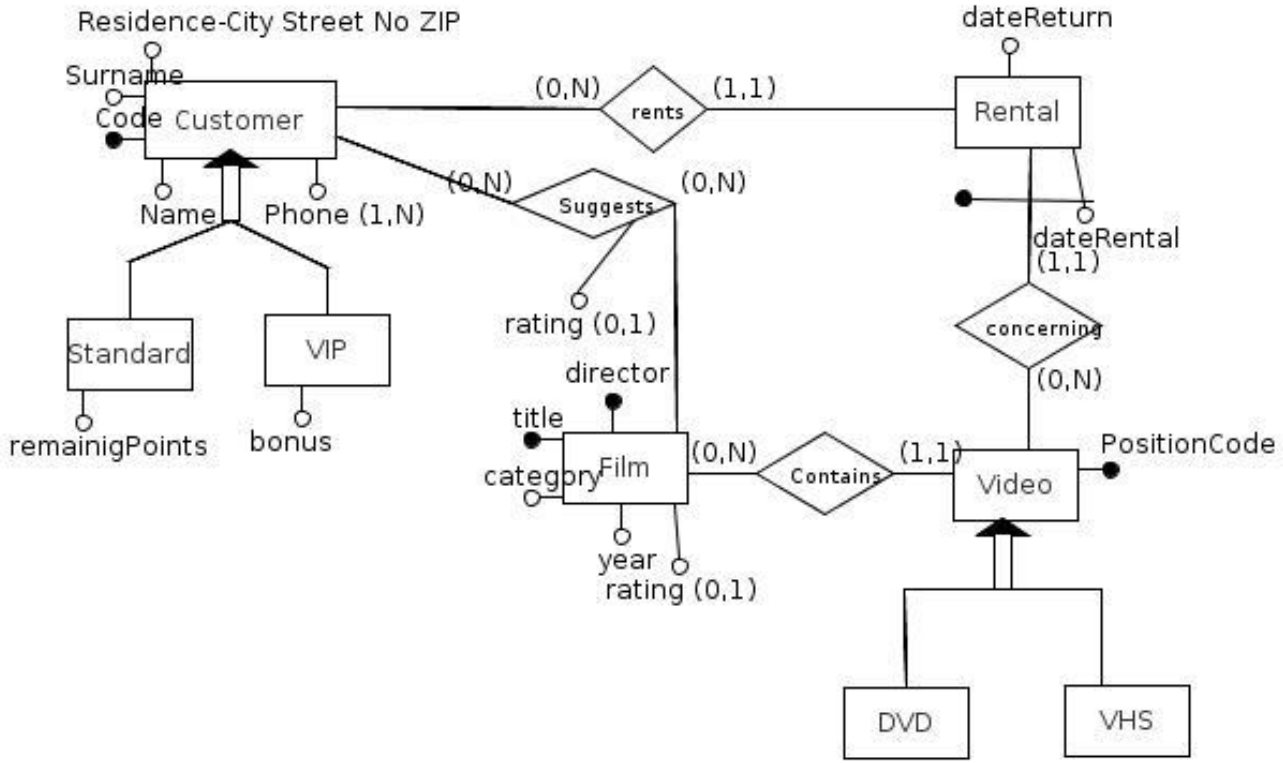
If, however, at least one of the preceding conditions occurs, the type definition is needed.

Video library

We want to design a database for a video library. The video library allows rental of about 1000 films.
For each movie, we want to store the title, name and surname of the director, the year of production, the category and the criticisms (expressed in a scale of decimal values between 0 and 5, including extremes), if present. Each movie is available for rental on a number of videos (where videos can be cassettes or DVDs). Every video available in the video library (about 3000) is identified by a placement code and media type (video cassette or DVD). The database will also have to store information about the video store customers (about 2000). For each customer of the video store we want to keep his name, surname, date of birth, residence (understood as city, street, civic number and zip code) and one or more telephone numbers. Each customer is identified by a code that corresponds to the card number issued for him / her to take advantage of video library services. Each customer can have a certain number of videos at the same time (no more than three). Each customer can also recommend movies to other customers, expressing a judgment for them (expressed on a scale of decimal values between 0 and 5, including extremes). The number of daily rentals to the video library is around 200. For each rental, we want store the date (day, month, year) in which the rental was made and, for the rentals made, the date (day, month, year) of restitution. The video library provides a customer loyalty program. Each rental allows the accumulation of a certain number of points. When accumulated points exceed a certain threshold, customers are qualified as VIP clients and are entitled to one bonus (I.e. a percentage discount). The database must store, for each VIP customer, its current value bonus. For standard customers, or those who have not yet earned enough points to access the VIP category, we want to store the number of missing points to access this category.

Transformation from EER Models

EER Model



```
Create type t_film as (
   Title VARCHAR(30),
   Director VARCHAR(20),
   Year DECIMAL(4),
   Category CHAR(15),
   Rating NUMERIC(3,2)
) NOT  FINAL;

Create type t_video as (
   PositionCode decimal(4),
   Film REF(t_film),
   Type CHAR default 'd'
) NOT FINAL;

Create type t_customer as (
   CodCus DECIMAL(4),
   Name VARCHAR(20),
   Surname VARCHAR(20),
   DoB  DATE,
   Phone CHAR(15) MULTISET,
   Residence ROW(street VARCHAR(20),
                 No Name VARCHAR(10),
                 Zip INTEGER,
                 City VARCHAR(20)),
   suggestions t_suggestions MULTISET
) NOT FINAL;
```

```
Create type t_rental as(
   PositionCode DECIMAL(4),
   DateRental DATE default
   current_date,
   DateRet DATE,
   Customer REF(t_customer),
   Video REF(t_video)
) NOT FINAL;

Create type t_suggestions as(
   Rating NUMERIC(3,2),
   Film REF(t_film) scope Film
) NOT FINAL;

Create type t_VIP UNDER t_customer
   as(
   Bonus NUMERIC(5,2)) FINAL;

Create type t_standard UNDER
   t_customer as(
   RemainingpPoints INTEGER) FINAL;
```

```
Create table Film of t_film (REF is idF,
    primary key (title, director),
    CHECK (rating >= 0.99 AND rating <= 5.00),
    Year with options not null,
    Category with options not null);

Create table Video of t_video
    (REF is idV,
    PositionCode with options primary key,
    Film with options SCOPE Film not null,
    type with options not null);

Create table Rental of t_rental
    (REF is idR,
    PRIMARY KEY (PositionCode,DateRental),
    UNIQUE (PositionCode,DateReturn),
    Customer with options SCOPE Customer);

Create table Customer of t_customer
    (REF is idC,
    PRIMARY KEY (CodCus),
    UNIQUE (DoB, name, surname),
    Name with options not null,
    Surname with options not null
    DoB with options not null
    Residence with options not null);

Create table Standard of t_standard under Customer;
Create table VIP of t_VIP under Customer;
```

Transformation from EER Models

Translation of associations.

Associations 1:N. They can be translated as a reference. See for example the translation of *Rents*. It is convenient in case we need to access the data of a customer from the data of the rentals. Alternatively, we could have introduced in the type **`t_customer`** a column type collection **`rentals`** with type of elements **`REF(t_rental)`**. This representation is convenient in the case where, as a client, it is necessary to frequently access to information relating to its rentals.

Transformation from EER Models

Translation associations.

Associations N:M. The translation of a many-to-many based on reference types requires mandatory use of an attribute of type collection. In the example, the translation of *recommend* can be obtained by extending the type **`t_customer`** to model movie recommendations from a customer or extending the type **`t_film`** to model the customers who have recommended a movie (or both). The choice of the type to be extended depends on the queries and the workload. To represent the attributes of relationships, we can add a row type **`t_suggestions`**.

Transformation from UML models

Example: *Online Movie Database*

Keep information about movies, actors, directors and other related information.

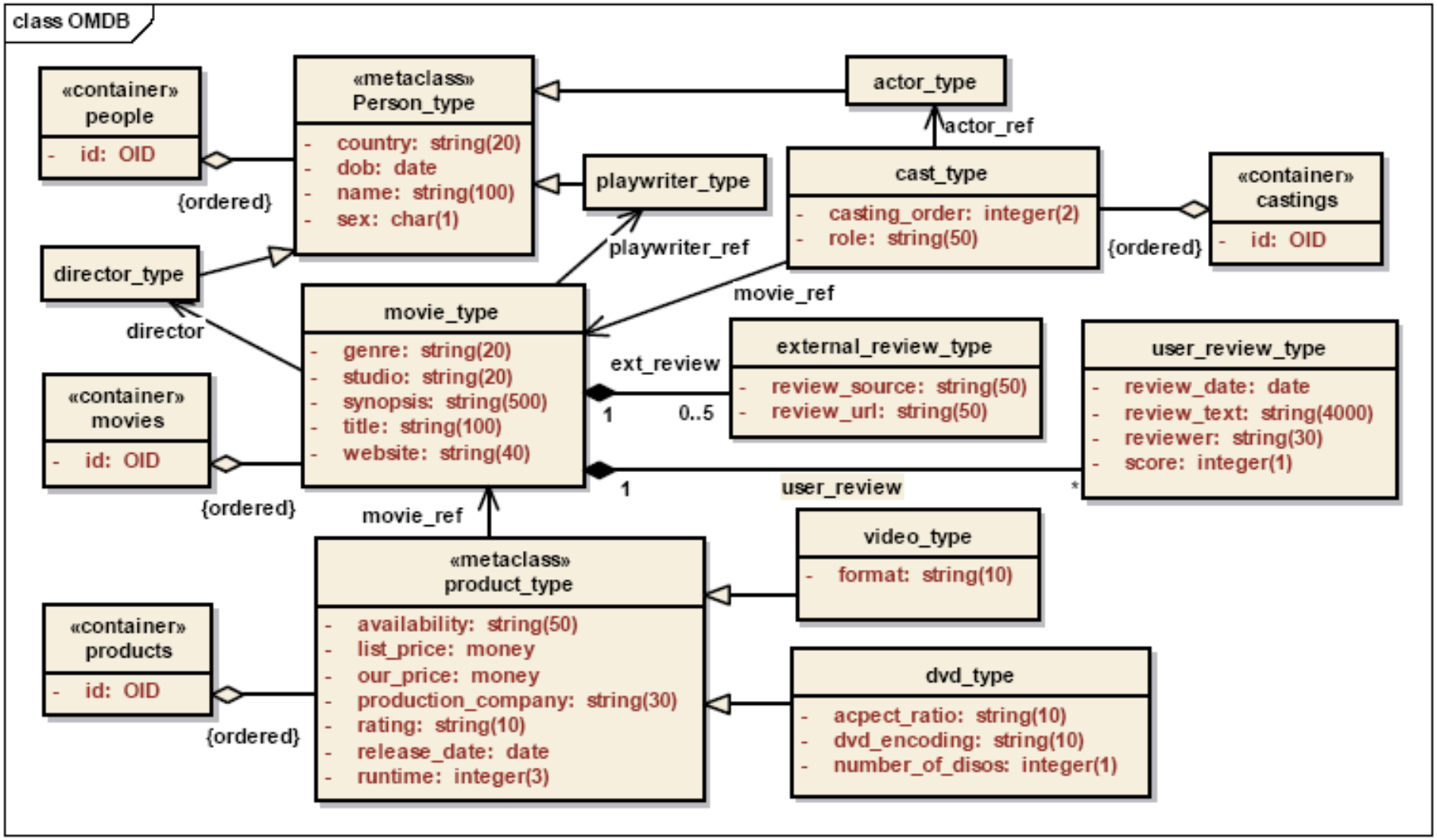
Users can explore these on the website and buy the products (eg, videos, DVDs, books, CDs). The information on the film includes the title, the director, the official web site, the genre, the studio, a brief synopsis, and cast.

Each film has up to 5 external reviewers and an unlimited number of 'reviews' of users.

The DB includes information about products for sale. In particular, information on video and DVD include the title, the approval rating, the price, release date, and other relevant information.

The UML model is shown below.

Transformation from UML models



Transformation from UML models

In the class diagram:

- the classes «*containers*» represent the tables;
- the «*metaclasses*» correspond to abstract types, i.e. not instantiable;
- the arrow A → B that connects two types A and B indicates that the type A contains an attribute whose type is a reference to a given type B;
- the composition between A and B indicates that the type A contains a structured attribute of type B;
- aggregation connects a class «*container*» with the type of contained objects.

Transformation from UML models

Table of typical rules of transformation from UML to SQL: 2003

Conceptual construct	**SQL:2003**	
	Type level	**Instance level**
Entity type	Structured type	Typed table
Attribute	Column type	
Multivalued	ARRAY/MULTISET	
Composed	ROW/structured type in column	
Calculated		Trigger/method
Relationship		
1:1	REF/REF	2 Typed tables
1:N	REF/[ARRAY|MULTISET]	2 Typed tables
N:M	ARRAY/[ARRAY|MULTISET]	2 Typed tables
Aggregation	REF/[ARRAY|MULTISET]	2 Typed tables
Composition	REF/[ARRAY|MULTISET]	2 Typed tables
Generalization	Type hierarchy	Table hierarchy
Hierarchy	Based on composition	

Transformation from UML models

There are several options to implement the aggregation and composition using SQL: 2003 structures.

Also, there are different approaches for identifiers, which we can classify as object-oriented or based on value. The first use the OID associated with structured types and stored in the REF type columns. The latter make use of the primary keys.

Finally, we must determine whether to implement the associations in a unidirectional or bidirectional manner. Although the expression is the same, the efficiency of navigation and updating changes.

By applying the rules of typical changes in the above table and without reference to the queries on OMDB, you will get the following type definitions from the UML diagram:

Transformation from UML models

```
create type external_review_type as object (...);
create type external_review_array
as varray(5) of external_review_type;
create type user_review_type as object (...);
create type user_review_table as table of
user_review_type;
create type person_type as object(...)not final;
create type director_type under person_type (...);
create type actor_type under person_type (...);
create type playwriter_type under person_type (...);
create type movie_type as object (...);
create type product_type as object(...)not final;
create type dvd_type under product_type (...);
create type video_type under product_type (...);
create type cast_type as object (...);
```

Transformation from UML models

Considerations on standardization of data.

The use of non-atomic attributes (eg, *residence*) violates the first normal form, but generally does not lead to redundancy in the data.

On the contrary, the use of collections (such as VARRAY) to implement repeated attributes (for example, *home phone*, *office phone*, *cell phone*) leads to a complex retrieval procedure and should be avoided.

The use of collections in SQL:2003 should be reserved for the implementation of compositions and aggregations. In such cases, the collections represent a separate data structure (although incorporated in other tables) and do not lead to redundancy in the data.

Transformation from UML models

Considerations on standardization of data (Cont.)

Using the OID we avoid the partial dependencies between key and non-key attributes (violating the second normal form), but the interdependencies between non-key attributes may still exist and lead to data redundancy, violating the third normal form and causing anomalies of update. Therefore, the principles of normalization are still applied even when using the OID.

Transformation from UML models

OID vs. keys

OIDs and REFs are generally used to implement complex relationships. The main advantage is the ability to navigate complex relationships without the need for relational connections, with a reduction of DB readings to find compound objects. Finally, many queries are simplified, as in the following example:

Lists the cast for the film 'The Godfather' (in order of importance) showing the actors and their roles.

```
SELECT C.ACTOR_REF.NAME, C.CAST_ROLE
FROM CASTINGS C
WHERE C.MOVIE_REF.TITLE = 'THE GODFATHER'
ORDER BY C.CASTING_ORDER;
```

Translation from UML models

Row types and structured types

Only structured types have OID and should be used, if it is required, by REF. Explicit row types (i.e., associated with the columns) can not be referenced directly (but, instead, it is necessary to use the dot notation, e.g. **`employee.address.postcode`**).

Structured types provide general support to the characteristics of objects, including inheritance and overriding.

In the example of OMDB, the implementation of a **`Person_type`** with its subtypes (**`Actor_type`**, **`Director_type`**, **`Playwriter_type`**) is a single table **`People OF Person_type`**. This project leads to a simple solution for queries like:

Translation from UML models

There are films in which the director is also one of the actors / actresses?

```
SELECT C.MOVIE_REF.TITLE, C.MOVIE_REF.DIRECTOR_REF.NAME,
    C.ROLE
FROM CASTINGS C
WHERE C.ACTOR_REF = C.MOVIE_REF.DIRECTOR_REF;
```

Implement the composition and aggregation

The general rule for using collections in implementing compositions and aggregations are as follows: each detail object must belong to a single master object and the detail objects must have no meaning outside of the context of the master object (example: book and book chapters).

Further decisions involve the selection of the collection. For example, in ORACLE, the question arises whether to use VARRAY or NESTED TABLES.

Translation from UML models

Implement the composition and aggregation (Cont.)

The VARRAY are more efficient if the collection is manipulated as a unit. The NESTED TABLE offer more options to access individual records using SQL queries that take advantage of the database optimizer, but require more memory (16 bytes for a NESTED_TABLE_ID pointing to the *parent* tuple of the nested table).

OMDB example of using:

- VARRAY for the five external auditors,
- NESTED_TABLE for user reviews (because it may be many).

In both cases we are using aggregation (revisions do not make sense if you do not refer to a film).

Translation from UML models

Implement the composition and aggregation (Cont.)

We avoided the use of collections to implement CASTINGS, which instead is implemented as a separate table that has greater flexibility and allows the use of all the expressiveness of the SQL.

For all the films, lists the main actress, i.e. the one that earns more.

```
SELECT C.MOVIE_REF.TITLE, C.MOVIE_REF.GENRE,
C.MOVIE_REF.DIRECTOR_REF.NAME, C.ACTOR_REF.NAME
FROM CASTINGS C
WHERE C.CASTING_ORDER =  (    SELECT
 MIN(CASTING_ORDER)
            FROM CASTINGS
            WHERE ACTOR_REF.SEX ='F'
            AND MOVIE_REF=C.MOVIE_REF);
```