Using object-relational databases

INDEX

- Introduction
- User-defined types
  - Object type
  - Methods
  - Object tables
  - Reference between objects
  - Collection types:
    - VARRAY type
    - NESTED tables
- Inheritance

Introduction

- An object-relational database (ORD), or object-relational database management system (ORDBMS), is a database management system (DBMS) similar to a relational database, but with an object-oriented database model:

  - **objects, classes and inheritance**

    are directly supported in database schemas and in the query language.

  - In addition, just as with proper relational systems, it supports extension of the data model with **custom data-types and methods**.

Introduction

- The basic goal for the ORD is to bridge the gap between relational databases and the object-oriented modeling techniques used in programming languages such as Java, C++, Visual Basic .NET or C#.

- The ORDBMS is integrated with an object-oriented programming language.

- Properties of ORDBMS are:
  - complex data
  - type inheritance
  - object behavior

Introduction

- **Complex data** creation is based on preliminary schema definition via the user-defined type (UDT).

- **Type inheritance**. That is, a structured type can have subtypes that reuse all of its attributes and contain additional attributes specific to the subtype.

- **Object behavior**. Program objects have to be storable and transportable for database processing, therefore they usually are named as persistent objects with its object identifier (OID).

Introduction

- In object-oriented programming (OOP) **object behavior** is described through the methods (object functions).

- The *methods* denoted by one name are distinguished by the number, types and order of their parameters and type of objects for which they attached (method signature).

- The OOP languages call this the *polymorphism* principle, which briefly is defined as "one interface, many implementations". Other OOP principles, *inheritance* and *encapsulation* are related both, with methods and attributes.

Introduction

- Advantages of ORDB:
  - They are compatible with traditional relational databases.
  - Manage complex data with minimum effort.
  - Data are more flexible and secure.
  - Part of application is on the server of database (methods).
  - Allows designing and programming large-scale systems that are:
    - ✔ easy to understand,
    - ✔ simple to debug and
    - ✔ fast to update.

Introduction

- ORACLE offers the first technological product of a database server hybrid, called object-relational and contains both technologies: relational and object.

- Oracle emphasizes the two main objectives following:
  - User-defined types.
  - Allow to get the data model closer to what is perceived and used in real-world applications.

User-defined types

- A data type defines a structure and common behavior for a set of application data.

- Oracle users can define their own data types using two categories:
  - object type
  - collection type

- To build user types we can use:
  - basic types provided by the system, and
  - other types previously defined by user.

User-defined types

- **Object type**.

  An object type represents a real world entity and consists of the following elements:

  - Name, used to identify uniquely the object type.

  - Attributes, that determine the structure and values of such data. Each attribute can be of a basic data type or user type.

  - Methods, procedures or functions that implement operations to be performed on real-world entities.

User-defined types

- Example:

```
  CREATE TYPE address_t AS OBJECT
( street          VARCHAR2(60),
  city            VARCHAR2(30),
  country         CHAR(2),
  postal_code     VARCHAR2(9));

  CREATE TYPE person AS OBJECT
( name            VARCHAR2(25),
  address         address_t)
```

User-defined types

- **Methods**.
  The specification of a method is found inside the creation of its type:

```
CREATE TYPE customer_t AS OBJECT
( custnum     NUMBER,
  name        VARCHAR2(50),
  address     address_t,
  phone       VARCHAR2(20),
  birth_date  DATE,
  MEMBER FUNCTION age RETURN NUMBER)
```

User-defined types

- **Methods** (version 1)

  The methods can be executed on objects of the same type.

```
CREATE TYPE BODY customer_t AS
  MEMBER FUNCTION age RETURN NUMBER IS
    aa NUMBER;
    d  DATE;
  Begin
    d:= sysdate;
    aa:= to_char(d,'yyyy') - to_char(birth_date,'yyyy');

    if (to_char(d,'mm') < to_char(birth_date,'mm')) or
       (to_char(d,'mm') = to_char(birth_date,'mm') and
       to_char(d,'dd') < to_char(birth_date,'dd')) then
      aa:= aa - 1;
    end if;
    return aa;
  end;
END;
```

User-defined types

- **Methods** (version 2)

  The methods can be executed on objects of the same type.

```
CREATE TYPE BODY customer_t AS
  MEMBER FUNCTION age RETURN NUMBER IS
    v_age NUMBER;
  Begin
    v_age:= FLOOR(MONTHS_BETWEEN(SYSDATE, birth_date) / 12);

    RETURN v_age;
  end;
END;
```

User-defined types

Constructor

Constructors are used to initialize an object. In most case, this initialization assigns values to the object's members.

A constructor is a function whose name is equal to its object's name. The function keyword is prepended with the **constructor** keyword.

While an ordinary function returns some type, a constructor function returns **self** as result.

```
CREATE TYPE customer_t AS OBJECT
(   custnum       NUMBER,
    name          VARCHAR2(50),
    address       address_t,
    phone         VARCHAR2(20),
    birth_date    DATE,
    CONSTRUCTOR FUNCTION customer_t(num NUMBER, name VARCHAR2)
    return self as result,
    MEMBER FUNCTION age RETURN NUMBER);
```

User-defined types

- **Constructor**

  Here's the body of the object.

```
CREATE TYPE BODY customer_t AS
  CONSTRUCTOR FUNCTION customer_t(num NUMBER, name VARCHAR2)
   return self as result IS
  BEGIN
    self.custnum:= num;
    self.name:= name;
    self.birth_date:= sysdate;
    RETURN;
  END;
  MEMBER FUNCTION age RETURN NUMBER IS
     v_age NUMBER;
  BEGIN
     v_age:= FLOOR(MONTHS_BETWEEN(SYSDATE, birth_date) / 12);

     RETURN v_age;
  END;
END;
```

User-defined types

- **Object tables.**

  An object table is a special kind of table storing an object in each row and providing access to the attributes of those objects as columns of the table.

```
CREATE TABLE customer_year_tab OF customer_t
(custnum PRIMARY KEY);

CREATE TABLE customer_old_tab
(    year      NUMBER,
     customer  customer_t);
```

- In Oracle we can consider a table of objects from two points of view:
  - As a table with one column whose type is an object type.
  - Like a table that has as many columns as attributes has the object.

User-defined types

- **Object tables**.

  So we can do a normal insert as we do with an ordinary table.

```
INSERT INTO customer_year_tab VALUES
(   2347,
    'Juan Pérez Ruíz',
    address_t ( 'Calle Castalia',
                'Onda',
                'ES',
                '34568'),
    '696-779789',
    DATE '1981-12-12');
```

User-defined types

- **Reference between objects.**
  A REF type attribute stores a reference to an object of the defined type, and implements a relationship of association between the two types of objects.

```
CREATE TABLE customer_tab OF customer_t;

CREATE TYPE order_t AS OBJECT
(   ordnum            NUMBER,
    customer          REF customer_t,
    request_date      DATE,
    delivery_address  address_t);
```

User-defined types

- **Reference between objects.**

When defining a type column to REF, you can restrict its domain to objects that are stored in a table.

```
CREATE TABLE order_tab OF order_t
(    PRIMARY KEY (ordnum),
     SCOPE FOR (customer) IS customer_tab);
```

So for inserting a new register in order_tab and doing that customer field points to the customer 3 of customer_tab, you can do this in this way:

```
INSERT INTO order_tab
SELECT 3001, REF(C),'30-MAY-1999', NULL
FROM customer_tab C WHERE C.custnum = 3;
```

User-defined types

Dangling REFs.

It is possible for the object identified by a REF to become unavailable through deletion of the object. Such a REF is called **dangling**. Oracle SQL provides a predicate (called IS DANGLING) to allow testing REFs for this condition.

Dangling REFs can be avoided by defining referential integrity constraints:

```
CREATE TABLE order_tab OF order_t
(    PRIMARY KEY (ordnum),
     customer REFERENCES customer_tab);
```

User-defined types

- **Collection type.**

In order to implement 1:N relationships, Oracle can define collection types. A collection data type is formed by an indefinite number of elements, all of the same type. Thus, it is possible to store a set of attribute tuples in:

- ✔ an array (VARRAY)
- ✔ a nested table.

User-defined types

- Collection type. **VARRAY.**

An array is an ordered set of elements of the same type. Each element has an associated index indicating its position within the array. Oracle allows VARRAY to be of variable length, but you must specify a maximum size when you declare the VARRAY type.

```
CREATE TYPE phones AS VARRAY(10) OF VARCHAR2(12);
```

You can use the VARRAY type for:

- ✔ Define the data type of a column in a relational table.
- ✔ Define the data type of an attribute of an object type.
- ✔ To define a PL/SQL variable, a parameter, or the type returned by a function.

User-defined types

- Collection type. **NESTED TABLES.**

A nested table is a set of elements of the same type without any predefined order. These tables can only have a column that can be of a basic data type in Oracle, or an object type defined by the user. In the latter case, the nested table can also be viewed as a table with as many columns as attributes have the type object.

```
CREATE TYPE player_t AS OBJECT
( id_player NUMBER,
  name      VARCHAR2(20),
  position  CHAR(10));

CREATE TYPE players_t AS TABLE OF player_t;
CREATE TYPE team_t AS OBJECT
( id_team     NUMBER,
  name        VARCHAR2(20),
  players     players_t,
  points      NUMBER);
```

User-defined types

- Collection type. **NESTED TABLES.**

```
CREATE TABLE team_tab OF team_t
( PRIMARY KEY(id_team)) NESTED TABLE players STORE AS
  nested_team_tab;
```

You need to perform this last step because the declaration of a nested table does not reserve any space for storage.

For inserting you only need to use the constructor of the nested table:

```
INSERT INTO team_tab VALUES
(2,'CAI',players_t(player_t(1,'Gasol','pivot'),pla
  yer_t(2,'Corbalan','base'),player_t(3,'Martin','
  alero')),5);
```

User-defined types

- Collection type. **NESTED TABLES.**

For inserting a new player into team 2 (that is already inserted into the team_tab) you need to use the clause TABLE:

```
INSERT INTO TABLE(SELECT players FROM team_tab
  WHERE id_team = 2)
VALUES(4,'Epi','base')
```

Inheritance

- Inheritance, substantial in orientation to objects, Oracle provides a simple inheritance model (does not support multiple inheritance): a subtype inherits all the methods and attributes of its only supertype. Additionally, the subtype can add new attributes and methods and redefine inherited methods.

```
CREATE TYPE person AS OBJECT
(       name          VARCHAR2(25),
        birth_date    DATE,
        FINAL MEMBER FUNCTION age RETURN NUMBER,
        MEMBER FUNCTION printme RETURN VARCHAR2)
NOT FINAL;
```

This statement creates a type root of a hierarchy of objects (the novelty here is the use of the clause "NOT FINAL").

Inheritance

- The following syntax allows to create a subtype:

```
CREATE TYPE employee UNDER person
(    salary NUMBER,
     MEMBER FUNCTION pay RETURN NUMBER,
     OVERRIDING MEMBER FUNCTION printme RETURN
  VARCHAR2);
```

The type 'employee' inherits all the attributes and methods of the supertype 'person'. In this example, we must add the attribute 'salary' and the function 'pay'. The function 'printme' is redefined using the clause 'OVERRIDING'.

Inheritance

```
CREATE TYPE BODY person AS
  FINAL MEMBER FUNCTION age RETURN NUMBER IS
  BEGIN
     RETURN (FLOOR(MONTHS_BETWEEN(SYSDATE, birth_date) / 12));
  END;

  MEMBER FUNCTION printme RETURN VARCHAR2 IS
  BEGIN
    RETURN (self.name || ' (' || self.birth_date || '): ' ||
  self.age || ' years old');
  END;
END;
```

Inheritance

```
CREATE TYPE BODY employee AS
  MEMBER FUNCTION pay RETURN NUMBER IS
  BEGIN
     RETURN (self.salary + self.age);
  END;

  OVERRIDING MEMBER FUNCTION printme RETURN VARCHAR2 IS
  BEGIN
    RETURN (self.name || ' (' || self.birth_date || '): ' ||
  self.age || ' years old and salary of ' || self.pay || '
  €');
  END;
END;
```

Inheritance

```
declare
  persona person:= person('David',date '2001-10-25');
  emp employee;
begin
  dbms_output.put_line(persona.printme);

  emp:= employee('John',date '2005-08-25',50000);
  dbms_output.put_line(emp.printme);
End;
```

---------- SQL Statement Output ----------

David (25-OCT-01): 21 years old

John (25-AUG-05): 17 years old and salary of 50017 €

END