4. Objects

Object elements have been introduced into the database and PL/SQL language since Oracle 8 version. Objects can now be stored in the database just like relational data. Also, the SQL language can be used to manipulate both relational and object data. From this version of DBMS Oracle is therefore an object-relational system.

4.1. Defining object types and their use

The object type, like the package, has its specification and its body. As in packages, the body is not mandatory and only appears if methods have been specified for the object type. The basic syntax for specification of the object type is presented below.

CREATE [**OR REPLACE**] **TYPE** type_name
[**AUTHID** {**CURRENT_USER** | **DEFINER**}]
{{{**AS | IS} OBJECT} | UNDER** parent_type_name}
({attribute_name [**REF**] attribute_type [, …]}
[,{[**MAP** | **ORDER**] [[**NOT**] **OVERRIDING**]
{**MEMBER | STATIC** | **CONSTRUCTOR**}
method_specification [, …]}]
[,{method_purity_level [, …]}])
[[**NOT**] **INSTANTIABLE**]
[[**NOT**] **FINAL**];

The AUTHID clause specifies whether the type will have the rights of the user using it or the rights defining the type. If the AS OBJECT clause appears in the above syntax, a base object type is defined (not inheriting from another type). If the UNDER clause occurs, instead of AS OBJECT, a derived type is defined that inherits from another (specified after UNDER), already existing, object type (parent type). The attribute type can be both a built-in type and an object type. The attribute can also be a reference (pointer) type to an object type. In this case the type name is preceded by the REF keyword. The specification of the method listed after MEMBER (normal method), STATIC (static method) or CONSTRUCTOR (the function that defines user own constructor - since Oracle 9.2 version) is its header.

The method can be both a function and a procedure, so its header is, respectively:

FUNCTION function_name [({parameter [, ...]})]
RETURN returning_type
lub
PROCEDURE procedurę_name [({parameter [, ...]})]

The method can override (cover up) any method with the same name in the parent type. In this case the OVERRIDING clause appears. In the absence of a definition of the own constructor function with the same name as the type name, the system defines the default constructor for the object, with name the same as name of type. The definition of the own constructor's function causes overriding the default constructor. The type returned by this function must be in such case specified by RETURN SELF AS RESULT clause. Before the function header may be placed a MAP or ORDER element. It means respectively **mapping method** or **ordering method** (this cannot be a static method or a constructor method). These methods specify ways to compare objects (without these methods, the only comparison operators are = and <>) used, among others, under the DISTINCT, GROUP BY and ORDER BY clauses. The mapping method, for the object for which it was called, returns the scalar value used to compare objects, while the ordering method gets the object (as a parameter) to be compared with the object for which the method was called and returns -1, 0 or 1 depending on whether the retrieved object is smaller, equal or larger than the object for which the method was called. The purity level of the method is determined in the same way as in packages (the RESCRICT_REFERENCES directive) and as in packages only applies to functions. The INSTANTIABLE and FINAL clauses relate to inheritance. NOT INSTANTIABLE defines the creation of an abstract type and FINAL does not allow creating child types for the defined type (the abstract type cannot be FINAL).

The syntax of definition of the body of the object type is as follows:

CREATE [**OR REPLACE**] **TYPE BODY** type_name {**AS | IS}**
{[**MAP** | **ORDER**] [[**NOT**] **OVERRIDING**]
{**MEMBER** | **STATIC** | **CONSTRUCTOR**}
definition_method_block_body [; …]**}**
END;

The method block contains its header and body. To refer within the method body to the object for which the method is being called the SELF qualifier is used. All other elements in the above definition have the same meaning as those in the definition of the object type specification. As already mentioned, the definition of the object type body not occur if the type does not possess methods.

The specification of INSTANTIABLE FINAL type can be modified according to the basic syntax presented below.

ALTER TYPE type_name
{{**REPLACE** [**AUTHID** {**CURRENT_USER** | **DEFINER**}]
AS OBJECT ({attribute_name [**REF**] attribute_type [, …]}
[, {[**MAP** | **ORDER**]
{**MEMBER | STATIC | CONSTRUCTOR**}
method_specification [, …]}])}
|
{{**ADD** | **DROP**} {[**MAP** | **ORDER**] **MEMBER**
method_specification }
|
{**ATTRIBUTE** attribute_name
[[**REF**] attribute_type]}
{**CASCADE** | **INVALIDATE}**}
};

Most of the clauses in the above syntax have already been explained as part of the CREATE TYPE command. The above syntax allows one to add a method by modifying the type as a whole (REPLACE clause) or by modifying its components (ADD or DROP clause). When deleting an attribute, its type is not listed. CASCADE means

cascading changes in types and dependent relations. INVALIDATE means changes only in the modified type, without checking connections with other types and relations. One of these two clauses must be used.

It is not possible to modify the body of the type. Any change in the list of the methods of type specification requires the overwriting (redefinition) of the type body definition.

The object type specification and its body are removed using the DROP command according to the syntax:

DROP TYPE type_name [**FORCE**];

DROP TYPE BODY type_name;

FORCE in the above syntax forces removal of the type regardless of the associations of the removed type with other types or tables.

Task. *Due to the growing crisis, Tiger ordered each cat to find one additional feeding place in the form of a human farm. Data on feeding places are to be collected in the* `Fedding_places` *relation with attributes* `nickname` *and* `place_owner`*, where* `nickname` *and* `feeding_place` *define the cat and the feeding place owner, respectively. The attribute* `person` *has the object type PERSON characterized by the* `name` *and the* `address`*, where* `address` *has object type ADDRESS characterized by the* `street` *and the* `house_number`*. For the PERSON type, is to be defined a mapping method and method that returns the full details of the person (in the form of a string). This type shall to be the basis for inheriting it by other types. Define such a structure.*

```
CREATE OR REPLACE TYPE ADDRESS AS OBJECT
(street VARCHAR2(25),
 house_number NUMBER(2));

TYPE ADDRESS compiled
```

```
CREATE OR REPLACE TYPE PERSON AS OBJECT
(name VARCHAR2(15),
 person_address ADDRESS,
 MAP MEMBER FUNCTION Compare RETURN VARCHAR2,
 MEMBER FUNCTION Data RETURN VARCHAR2,
 PRAGMA RESTRICT_REFERENCES(Data,RNDS,WNDS,RNPS,WNPS))
NOT FINAL;

TYPE PERSON compiled

CREATE OR REPLACE TYPE BODY PERSON AS
  MAP MEMBER FUNCTION Compare RETURN VARCHAR2 IS
  BEGIN
  RETURN name||person_address.street||
         person_address.house_number;
  END;
    MEMBER FUNCTION Data RETURN VARCHAR2 IS
  BEGIN
   RETURN name||', '||person_address.street||
          ' street '||person_address.house_number;
  END Data;
END;

TYPE BODY PERSON compiled

CREATE TABLE Feeding_places
(nickname VARCHAR2(15) CONSTRAINT fe_pk PRIMARY KEY
                CONSTRAINT fe_ni_fk REFERENCES Cats(nickname),
 place_owner  PERSON);

table FEEDING_PLACES created.
```

Note: The object type attribute cannot be a primary or unique key.

The following are examples illustrating the use of the ALTER TYPE command for the object type `ADDRESS`.

```
ALTER TYPE ADDRESS REPLACE AS OBJECT
(street VARCHAR2(25),
 house_number NUMBER(2),
 MEMBER FUNCTION Get_street RETURN VARCHAR2);

type ADDRESS altered.
```

```
CREATE OR REPLACE TYPE BODY ADDRESS AS
 MEMBER FUNCTION Get_street RETURN VARCHAR2 IS
 BEGIN
   RETURN street;
 END;
END;
```

```
TYPE BODY ADDRESS compiled
```

```
ALTER TYPE ADDRESS
ADD ATTRIBUTE apartnment_no NUMBER
INVALIDATE;
```

```
type ADDRESS altered.
```

```
ALTER TYPE ADDRESS
ADD MEMBER FUNCTION Get_house_no RETURN NUMBER
INVALIDATE;
```

```
type ADDRESS altered.
```

```
CREATE OR REPLACE TYPE BODY ADDRESS AS
 MEMBER FUNCTION Get_street RETURN VARCHAR2 IS
 BEGIN
   RETURN street;
 END;
 MEMBER FUNCTION Get_house_no RETURN NUMBER IS
 BEGIN
   RETURN house_number;
 END;
END;
```

```
TYPE BODY ADDRESS compiled
```

```
ALTER TYPE ADDRESS
DROP MEMBER FUNCTION Get_house_no RETURN NUMBER
INVALIDATE;
```

```
type ADDRESS altered.
```

```
ALTER TYPE ADDRESS
DROP MEMBER FUNCTION Get_street RETURN VARCHAR2
INVALIDATE;
```

```
type ADDRESS altered.
```

```
ALTER TYPE ADDRESS
DROP ATTRIBUTE apartnment_no
INVALIDATE;
```

```
type ADDRESS altered.
```

```
DROP TYPE BODY ADDRESS;
```

```
type body ADDRESS dropped.
```

Data for the `Fedding_places` relation can be entered, using in the INSERT command, default constructors of objects.

```
INSERT INTO Feeding_places
       VALUES('TIGER',PERSON('JAN',ADDRESS('FIELD',2)));
INSERT INTO Feeding_places
       VALUES('LOLA',PERSON('JAN',ADDRESS('FIELD',2)));
INSERT INTO Feeding_places
       VALUES('BOLEK',PERSON('SOPHIE',ADDRESS('LONG',7)));
INSERT INTO Feeding_places
       VALUES('SMALL',PERSON('ADAM',ADDRESS('WET',21)));

1 rows inserted.
1 rows inserted.
1 rows inserted.
1 rows inserted.
```

Task. *Display nicknames of cats along with their feeding places.*

```
SELECT nickname "Cat",F.place_owner.Data() "Feeding place"
FROM Feeding_places F
ORDER BY place_owner;

Cat             Feeding place
--------------- -------------------------
SMALL           ADAM, WET street 21
TIGER           JAN, FIELD street 2
LOLA            JAN, FIELD street 2
BOLEK           SOPHIE, LONG street 7
```

The above solution uses the `Data` method and, implicitly, mapping method `Compare`, which allows one to order the resulting rows by the value of the object `place_owner`. The condition of referring to fields or methods of an object is specifying the name of the relation containing the object (or its alias specified in the FROM clause) before referring to the field or method.

Relations with objects one can join with classic relations.

Task. *Display nicknames of male cats along with their feeding places.*

```
SELECT nickname "Cat",F.place_owner.Data() "Feeding place"
FROM Feeding_places F NATURAL JOIN Cats
WHERE gender='M';

Cat             Feeding place
--------------- -------------------------
SMALL           ADAM, WET street 21
TIGER           JAN, FIELD street 2
BOLEK           SOPHIE, LONG street 7
```

One can also perform grouping by objects (implicit using of the mapping method here) and by object fields.

Task. *Specify how many cats reside in each feeding place.*

```
SELECT place_owner "Place owner",COUNT(*) "Number of cats"
FROM Feeding_places
GROUP BY place_owner;

Place owner                                Number of cats
------------------------------------------ ---------------
Z.PERSON('ADAM',Z.ADDRESS('WET',21))       1
Z.PERSON('JAN',Z.ADDRESS('FIELD',2))       2
Z.PERSON('SOPHIE',Z.ADDRESS('LONG',7))     1

SELECT F.place_owner.name "Host",COUNT(*) "Number of cats"
FROM Feeding_places F
GROUP BY F.place_owner.name;

Host            Number of cats
--------------- ----------------------
SOPHIE          1
ADAM            1
JAN             2
```

Allowed is also modification of entire objects and their fields.

Task. *Provide to the cat with the nickname 'BOLEK' a new host and then change the name of this host to 'KLAUDIA'.*

```
UPDATE Feeding_places
SET place_owner=PERSON('KAROLA',ADDRESS('GREEN',16))
WHERE nickname='BOLEK';

1 rows updated.

UPDATE Feeding_places F
SET F.place_owner.name='KLAUDIA'
WHERE nickname='BOLEK';

1 rows updated.

ROLLBACK;
rollback complete.
```

To objects and their fields one can also reference in the HAVING clause and the WHERE clause of SELECT query, and in the WHERE clause of DML commands. Object types used in the above way (as types of relation attributes) do not significantly affect the way of defining database queries. The objects used in this way are called **column objects**.

Another type of objects are so-called **row objects**. A relation using a row object consists of rows that are objects (row objects). Such a relation is defined according to the syntax:

CREATE TABLE relations_name **OF** name_of_row_object_type
[({relations_constraint [, ...]})];

Each relation row object has an `OID` identifier assigned by the system. The identifier value can be obtained by using the REF function/operator in the SELECT query, whose argument is an alias for the relation with the row object. It should be noted that all constraints of object (including those related to individual object fields) are not specified as part of the object type definition but only as part of the relation definition consisting of elements of this type (through relation constraints). **The primary key of the relation with the row object cannot be based on a field of reference type (REF)**. If in the object is no other key candidate, an additional attribute must be defined in object, which will be act as artificial key of the relation.

Task. *Define a relation with row object of type PERSON and then fill this relation with data.*

```
CREATE TABLE PersonsR OF PERSON
(CONSTRAINT psr_pk PRIMARY KEY (name));

table PERSONSR created.

INSERT INTO PersonsR VALUES(PERSON('JAN',ADDRESS('FIELD',2)));
INSERT INTO PersonsR VALUES('SOPHIE',ADDRESS('LONG',7));
INSERT INTO PersonsR VALUES('ADAM',ADDRESS('WET',21));

1 rows inserted.
1 rows inserted.
1 rows inserted.
```

As the example above shows, in the INSERT command working on a relation with a row object, there is no need to explicitly point to the name of the constructor of that object. If the row object contains nested objects, the use of nested object constructor names is required.

In a SELECT query based on a relation with row object, containing nested objects, nested objects can only be referenced by using the VALUE operator/function returning the row object. The operator's argument can only be an alias of relation with row object.

Task. *Display, remembered in the* `PerdonsR` *relation, names of persons and street names where they live.*

```
SELECT name "Name",VALUE(P).person_address.street "Street"
FROM PersonsR P;

Name            Street
--------------- -------------------------
JAN             FIELD
ZOFIA           LONG
ADAM            WET
```

As mentioned, to get the value OID identifier, one should use the REF function with argument is, similarly like in function VALUE, being an alias for the relation with the row object. The VALUE and REF functions can also be used as part of PL/SQL commands

For other SQL commands working on relations with row objects, similar rules apply as for relations with column objects.

As already mentioned, to the fields of object type as well as to their components one can reference in the WHERE clause of SQL command. Such searching can be accelerated by defining the appropriate index. Indexes for components of object type, being non-object, both for column and row objects, are defined according to standard syntax. However, one cannot apply indexes to entire fields of object type.

Task. *Define the index for the attribute* `place_owner` *in the* `Feeding_places` *relation and the index for the* `street` *attribute of the* `PersonsR` *relation.*

.

```
CREATE INDEX Feeding_places_name_ind
ON Feeding_places(place_owner.name);

index FEEDING_PLACES_NAME_IND created.

CREATE INDEX PersonsR_street_ind
ON PersonsR(person_address.street);

index PERSONSR_STREET_IND created.
```

Object types can be inherited by other object types.

Task. *Tiger stated that the feeding places in the village of residence was not enough. So he and ordered his subordinates to find one additional feeding place outside the family village. Data of foreign feeding places are to be collected in the* `Foreign_feeding_places` *relation with attributes* `nickname` *and* `place_owner`*, where the* `nickname` *is defining the cat, the* `place_owner` *defining the external feeding place. This owner is to be described by the* `FOREIGN_PERSON` *type differing from the* `PERSON` *type only by the parameter specifying the city of residence of the feeding place owner. For the type* `FOREIGN_PERSON`*, a method should be defined that returns, in the form of a string, the full details of the owner of the external feeding place. This type is no longer to be inherited.*

```
CREATE OR REPLACE TYPE FOREIGN_PERSON UNDER PERSON
(city VARCHAR2(25),
 MEMBER FUNCTION Foreign_data RETURN VARCHAR2,
 PRAGMA RESTRICT_REFERENCES(Foreign_data,RNDS,WNDS,RNPS,WNPS))
FINAL;

TYPE FOREIGN_PERSON compiled

CREATE OR REPLACE TYPE BODY FOREIGN_PERSON AS
 MEMBER FUNCTION Foreign_data RETURN VARCHAR2 IS
 BEGIN
   RETURN city||', '||SELF.Data();
 END Foreign_data;
END;

TYPE BODY FOREIGN_PERSON compiled

CREATE TABLE Foreign_feeding_places
(nickname VARCHAR2(15) CONSTRAINT ffp_pk PRIMARY KEY
        CONSTRAINT ffp_ca_fk REFERENCES Cats(nickname),
 place_owner FOREIGN_PERSON);

table FOREIGN_FEEDING_PLACES created.

INSERT INTO Foreign_feeding_places
VALUES('TIGER',FOREIGN_PERSON('MARIA',ADDRESS('GOLD',22),
       'WARSZAWA'));
INSERT INTO Foreign_feeding_places
VALUES('LOLA',FOREIGN_PERSON('MARIA',ADDRESS('GOLD',22),
       'WARSZAWA'));
INSERT INTO Foreign_feeding_places
VALUES('BOLEK',FOREIGN_PERSON('CHARLES',ADDRESS('CARBON',17),
       'KATOWICE'));
INSERT INTO Foreign_feeding_places
VALUES('SMALL',FOREIGN_PERSON('ROMAN',ADDRESS('POTATO',11),
       'POZNAN'));

1 rows inserted.
1 rows inserted.
1 rows inserted.
1 rows inserted.
```

Task. *Display nicknames of cats along with the data of their foreign feeding places.*

```
SELECT nickname "Cat",
       SUBSTR(F.place_owner.Foreign_data(),1,45)
       "Emergency feeding place"
FROM Foreign_feeding_places F;

Cat             Emergency feeding place
--------------- -------------------------------------------
TIGER           WARSZAWA, MARIA, GOLD street 22
LOLA            WARSZAWA, MARIA, GOLD street 22
BOLEK           KATOWICE, CHARLES, CARBON street 17
SMALL           POZNAN, ROMAN, POTATO street 11
```

An alternative way of defining reference relationships in relation to classic foreign keys may be the use of reference types (types defined by the REF keyword placed before the name of the object type defining the type of the relation attribute or the type of the object field).

Task. *Each cat has only one feeding place (one host). However, the same feeding place (i.e. the estate of the same host) can be inhabited by many cats. Model such a relationship between entities in a classic way (by reference relationships defined using foreign keys) and then in an object-oriented way (using a reference type).*

<u>The classic version of the solution of the task using foreign keys:</u>

```
CREATE TABLE PersonsT  -- feeding places
(name VARCHAR2(15) CONSTRAINT pet_pk PRIMARY KEY,
 person_address ADDRESS);

table PERSONST created.

CREATE TABLE Feeding_placesT   -- cats with reference to
feeding places
(nickname VARCHAR2(15) CONSTRAINT fpt_pk PRIMARY KEY
        CONSTRAINT fpt_ca_fk REFERENCES Cats(nickname),
 name VARCHAR2(15) CONSTRAINT fpt_pn_fk
                   REFERENCES PersonsT(name));

table FEEDING_PLACEST created.
```

The object version of the task solution using the relation with the column object:

The solution consists in the definition of the relation `Feeding_placesO` with the column object being the reference to the `PERSON` type defining the owner of feeding place and with the attribute specifying the cat's nickname. For reminder, the following is also the definition of the previously defined object type `PERSON` and of the relation `PersonsR` with a row object of this type.

```
CREATE OR REPLACE TYPE PERSON AS OBJECT
(name VARCHAR2(15),
 person_address ADDRESS,
 MAP MEMBER FUNCTION Compare RETURN VARCHAR2,
 MEMBER FUNCTION Data RETURN VARCHAR2,
 PRAGMA RESTRICT_REFERENCES(Data,RNDS,WNDS,RNPS,WNPS))
NOT FINAL;

TYPE PERSON compiled

CREATE OR REPLACE TYPE BODY PERSON AS
  MAP MEMBER FUNCTION Compare RETURN VARCHAR2 IS
  BEGIN
  RETURN name||person_address.street||
         person_address.house_number;
  END;
    MEMBER FUNCTION Data RETURN VARCHAR2 IS
  BEGIN
   RETURN name||', '||person_address.street||
          ' street '||person_address.house_number;
  END Data;
END;

TYPE BODY PERSON compiled


CREATE TABLE PersonsR OF PERSON
(CONSTRAINT psr_pk PRIMARY KEY (name));

table PERSONSR created.

CREATE TABLE Feeding_placesO -- cats with reference
                             -- to feeding places
(nickname VARCHAR2(15) CONSTRAINT fpo_pk PRIMARY KEY
          CONSTRAINT fpo_ca_fk REFERENCES Cats(nickname),
 owner REF PERSON SCOPE IS PersonsR);

table FEEDING_PLACESO created.
```

The SCOPE IS clause indicates the name of the object relation (this cannot be a normal relation) to which objects the reference type should refer. Here it is relation of `PersonsR` with a row object of the type `PERSON`. The foreign key connecting `Feeding_placesO` relation with `Cats` relation, due to the latter's existence, was defined in a relational way.

Below, the `Feeding_placesO` relation is filled with sample data. The only way to accomplish this task is to use the INSERT command that uses a subquery that selects a reference value to the appropriate (related) row object of the `PersonsR` relation.

```
INSERT INTO Feeding_placesO
SELECT 'TIGER',REF(P) FROM PersonsR P WHERE P.name='JAN';
INSERT INTO Feeding_placesO
SELECT 'LOLA',REF(P) FROM PersonsR P WHERE P.name='JAN';
INSERT INTO Feeding_placesO
SELECT 'BOLEK',REF(p) FROM PersonsR P WHERE P.name='SOPHIE';
INSERT INTO Feeding_placesO
SELECT 'SMALL',REF(P) FROM PersonsR P WHERE P.name='ADAM';

1 rows inserted.
1 rows inserted.
1 rows inserted.
1 rows inserted.
```

Task. *Display cat nicknames and names and details about their hosts.*

```
SELECT nickname "Cat", F.owner.name "Host",
       SUBSTR(F.owner.Data(),1,20) "Host data"
FROM Feeding_placesO F;

Cat             Host            Host data
--------------- --------------- --------------------
LOLA            JAN             JAN, FIELD street 2
TIGER           JAN             JAN, FIELD street 2
BOLEK           SOPHIE          SOPHIE, LONG street
SMALL           ADAM            ADAM, WET street 21
```

In the above command, reference was made directly to the fields and methods indicated by the reference (field `name` and method `Data()`). This option exists only for SQL. In PL/SQL, in this case, the DEREF function should be used to process the reference to the object, which is

a parameter of the function. The above query, collecting the data of the Tiger's host (in PL/SQL, the SELECT ... INTO command can return only one row), would have there the shape:

```
SELECT nickname, DEREF(F.owner).name DEREF(F.owner).Data()
INTO ni,na,da
FROM Feeding_placesO F
WHERE nickname='TIGER';
```

where `ni`, `na` and `da` are the variables to which the returned values will be assigned.

The transformation opposite to DEREF is performed by the REF function.

The following is the use of the DEREF function to display the feeding place data of the cat nicknamed `'SMALL'`.

```
SELECT DEREF(owner) "Host"
FROM Feeding_placesO
WHERE nickname='SMALL';

Host
------------------------------------
Z.PERSON('ADAM',Z.ADDRESS('WET',21))
```

Although the query only works on the `Feeding_placesO` relation, however, in the examples above, in the background, relation `Feeding_placesO` was joined with the `PersonsR` relation, without specifying of join criteria. Therefore, neither the name of the related object relation nor the join condition is needed here to reach the values stored in this relation.

<u>The object version of the task solution using the relation with row object:</u>

The solution is to define the relation `Feeding_placesRO` with row object `FEEDING_PLACERO` type containing a field specifying the nickname of the cat and a field specifying the owner of feeding place being the reference to the `PERSON` type.

```
CREATE TYPE FEEDING_PLACERO AS OBJECT -- cats with reference
                                      -- to feeding places
(nickname VARCHAR2(15),
 owner REF PERSON);

TYPE FEEDING_PLACERO compiled

CREATE TABLE Feeding_placesRO OF FEEDING_PLACERO
(owner SCOPE IS PersonsR,
 CONSTRAINT fpr_pk PRIMARY KEY (nickname),
 CONSTRAINT fpr_ca_fk FOREIGN KEY (nickname)
                      REFERENCES Cats(nickname));

table FEEDING_PLACESRO created.
```

As previously mentioned, in relations using row objects, object constraints are determined not as part of the object type definition but only as part of the definition of relation with row type object. In the above case, this applies to the indication, through the SCOPE IS clause, of the relation of `PersonsR` as the relation, to objects of which is occurs reference. This also applies to the definition of primary key of the `Feeding_placesRO` relation being the field of object, as well as the definition of the foreign key referencing to `Cats` relation, which key is also the field of object. The latter, due to the already existing relation `Cats`, was defined in a relational way.

Data for the `Feeding_placesRO` relation can be entered in exactly the same way as it was done for the `Feeding_placesO` relation. The same shape also has a query returning the data of cats and the data of their feeding place. Also, handling this relation in PL/SQL, just like it was for the `Feeding_placesO` relation, would require the eventual use of the DEREF and REF functions.

Deleting a related object may result in the creation of so-called "dangling" reference in the object in which this reference was defined. Such references can be identified by using the IS DANGLING operator (the reverse operator is IS NOT DANGLING).

Task. *Check if there are "dangling" references to the object of PERSON type in the* `Feeding_placesO` *relation.*

```
SELECT nickname FROM Feeding_placesO
WHERE owner IS DANGLING;

NICKNAME
---------------

DELETE FROM PersonsR WHERE name='ADAM';

1 rows deleted.

SELECT nickname FROM Feeding_placesO
WHERE owner IS DANGLING;

NICKNAME
---------------
SMALL

ROLLBACK;
rollback complete.
```

Information about object types can be found in the USER_OBJECTS system view and information about the methods of object types in the USER_METHOD_PARAMS, USER_METHOD_RESOULTS and USER_TYPE_METHODS system views.

4.2. Object views

Object views are used to adapt the existing relational database to use the mechanisms provided by the object extension of the Oracle database. These views give the opportunity to impose object structures such as object data types and methods on existing relational tables. Object views can have attributes of object type (then they will be views with column objects) or consist of rows that are objects of a particular type (then they will be views with row objects). In both cases, they will download data from existing relational tables. A relational database, thanks to object views, can therefore be seen as an object-relational database.

Task. *Create the* `Feeding_places1` *relation as a standard relational table and then, by defining object views, allow to use object mechanisms for it.*

```
CREATE TABLE Feeding_places1
(nickname VARCHAR2(15) CONSTRAINT fp1_pk PRIMARY KEY
        CONSTRAINT fp1_ca_fk REFERENCES Cats(nickname),
 name VARCHAR2(15),
 street VARCHAR2(25),
 house_number NUMBER(2));

table FEEDING_PLACES1 created.

INSERT INTO Feeding_places1 VALUES('TIGER','JAN','FIELD',2);
INSERT INTO Feeding_places1 VALUES('LOLA','JAN','FIELD',2);
INSERT INTO Feeding_places1 VALUES('BOLEK','SOPHIE','LONG',7);
INSERT INTO Feeding_places1 VALUES('SMALL','ADAM','WET',21);

1 rows inserted.
1 rows inserted.
1 rows inserted.
1 rows inserted.

CREATE OR REPLACE VIEW Feeding_placesov (nickname,owner) AS
SELECT nickname,PERSON(name,ADDRESS(street,house_number))
FROM Feeding_places1;

view FEEDING_PLACESOV created.
```

For defined in such a way object view `Feeding_places1`, with a column object of the type `PERSON`, all queries presented for the previously defined `Feeding_places` relation apply. So one can also use here the methods defined for the `PERSON` type. As an example, the bellow task will be performed:

Task. *Display nicknames of cats along with data of their feeding places.*

```
SELECT nickname "Cat",SUBSTR(F.owner.Data(),1,45)
                      "Feeding place"
FROM Feeding_placesov F
ORDER BY owner;

Cat             Feeding place
--------------- --------------------------------------------
SMALL           ADAM, WET street 21
TIGER           JAN, FIELD street 2
LOLA            JAN, FIELD street 2
BOLEK           SOPHIE, LONG street 7
```

At the object view level, one can also use reference types to model reference relationships, that exist in a relational database (unfortunately, this does not apply to references the relation with itself). Below, a relationship mapping the reference relationship of `Feeding_places1` relation with `Cats` relation will be modeled (`Feeding_placesov` view does not include such a connection). To make this possible, using the object perspective, one must define OID identifiers for the rows of the `Cats` relation, downloaded by this view. The view that accomplishes this task must contain rows of the type consistent with the row type of `Cats` relation, so it must consist of row objects of this type. It is therefore necessary to first define the abstract type with a structure consistent with the scheme of `Cats` relation. This definition is provided below. The new type was equipped additionally with two methods, one returning the name of the cat's gender (`About_gender` method), the other returning the full monthly mice ration of the cat (the `Mice_income` method).

```
CREATE OR REPLACE TYPE CAT_TYPE AS OBJECT
(name VARCHAR2(15),
 gender VARCHAR2(1),
 nickname VARCHAR2(15),
 function VARCHAR2(10),
 chief VARCHAR2(15),
 in_herd_since DATE,
 mice_ration NUMBER(3),mice_extra NUMBER(3),
 band_no NUMBER(2),
 MEMBER FUNCTION About_gender RETURN VARCHAR2,
 MEMBER FUNCTION Mice_income RETURN NUMBER);

CREATE OR REPLACE TYPE BODY CAT_TYPE  AS
 MEMBER FUNCTION About_gender RETURN VARCHAR2 IS
        BEGIN
          RETURN CASE NVL(gender,'U')
                    WHEN 'M' THEN 'Male cat'
                    WHEN 'W' THEN 'Female cat'
                    WHEN 'A' THEN 'Unknown'
                    ELSE 'Wrong'
                  END;
 END;
 MEMBER FUNCTION Mice_income RETURN NUMBER IS
        BEGIN
          RETURN NVL(mice_ration,0)+NVL(mice_extra,0);
        END;
END;

TYPE BODY CATS_TYPE compiled
```

In the next step, one must define an object view based on the CAT_TYPE type (with a row object of this type), which downloads data from the Cats relation, with assigning the OID identifiers to the rows that are downloaded.

```
CREATE OR REPLACE VIEW Cats_with_oid OF CAT_TYPE
WITH OBJECT IDENTIFIER (nickname) AS
SELECT name,gender,nickname,function,chief,
       in_herd_since,mice_ration,mice_extra,
       band_no
FROM Cats;

view CATS_WITH_OID created.
```

The OF keyword is followed by the type name, which specifies the type of view row. The WITH OBJECT IDENTIFIER clause specifies the attribute (or list of attributes) used as the basis for the OID identifier (usually it is the primary key of the relation on which the view is based). The view is built on the basis of data downloaded from the Cats relation, whose structure (SELECT clause) is consistent with the structure of the type CAT_TYPE.

The Feeding_places1 relation must refer to the OID identifiers of the rows of Cats relations, downloaded by the Cats_with_oid view. Therefore, on the basis of Feeding_places1 relation, an object view should be created that implements such references. To define the reference, the MAKE_REF function will be used with arguments: the name of the object being referenced and the name of the attribute (or their list) creating a foreign key (implementing the modeled relationship at the relational level) in the relation that is the basis of the view.

```
CREATE OR REPLACE VIEW Feeding_places_with_oid AS
SELECT MAKE_REF(Cats_with_oid,nickname) nickname,
       name,street,house_number
FROM Feeding_places1;

view FEEDING_PLACES_WITH_OID created.
```

The above view is a view with a column object (contains a nickname field that is a reference to an object of the type CAT_TYPE). If any other relation were related by reference with the Feeding_places1 relation (it had a foreign key coming from Feeding_places1 relation), then to the rows Feeding_places1 relation should also be assigned OID identifier. Therefore, it would be necessary to define a type compatible to the Feeding_places1 relation schema (e.g. named FEEDING_PLACE1_TYPE), within which there would be a reference type field (REF) to the type compatible to schema of the related Cats relation, i.e. the type CAT_TYPE. This field would correspond to the foreign key of this relation. Because the attribute pseudo in the Feeding_places1 relation plays both the role of the primary and foreign key and the reference cannot be the basis of the OID identifier, should be added to the type FEEDING_PLACE1_TYPE the attribute nickname_id of the type which corresponds to the type of the attribute nickname of this relation. This additional attribute will contain the values of the Feeding_places1 relation nickname attribute. As it has a simple type and containing unique values, will be the basis for the OID identifier. This procedure is not necessary if the foreign key does not also play the role of the primary key. The FEEDING_PLACE1_TYPE type definition would be:

```
CREATE OR REPLACE TYPE FEEDING_PLACE1_TYPE AS OBJECT
(nickname_id VARCHAR2(15),
 nickname REF CAT_TYPE
 name VARCHAR2(15),
 street VARCHAR2(25),
 house_number NUMBER(2));

TYPE FEEDING_PLACE1_TYPE compiled
```

Next, instead of the Feeding_places_with_oid view, one should define a view with a FEEDING_PLACE1_TYPE type row object with the WITH OBJECT IDENTIFIER clause, indicating the attribute used to build the OID (in this case nickname_id). So the perspective would be defined as follows:

```
CREATE OR REPLACE VIEW Feeding_places1_with_oid OF
FEEDING_PLACE1_TYPE
WITH OBJECT IDENTIFIER (nickname_id) AS
SELECT nickname nickname_id,
       MAKE_REF(Cats_with_oid,nickname) nickname,
       name,street,house_number
FROM Feeding_places1;

view FEEDING_PLACES1_WITH_OID created.
```

As the solution to the task below shows, queries to object views containing references have the same structure as queries to relations containing references. Also, as in that case, the solution in PL/SQL language to the following task will require the use of the DEREF function to process the reference to the object, which is a parameter of the function.

Task. *Display nicknames, genders and addresses of cats with feeding place at Jan's.*

```
SELECT FPOID.nickname.nickname "Cat",
       SUBSTR(FPOID.nickname.About_gender(),1,12) "Gender",
       SUBSTR(street||' '||house_number,1,20) "Address"
FROM Feeding_places_with_oid FPOID
WHERE name='JAN';

Cat             Gender       Address
--------------- ------------ --------------------
TIGER           Male cat     FIELD 2
LOLA            Female cat   FIELD 2
```

In the solution of the above task, equivalently one can use the `Feeding_places1_with_oid` view.

For object views, the INSTEAD OF triggers discussed earlier can be fully used. With object view, one can create object types and use them at the same time with existing relational tables. Object type methods can be used for both data from relational tables and data from tables containing objects. Therefore, object views give the opportunity, depending on the needs, to treat the database as relational or as object-relational.

4.3. Object-oriented PL/SQL language

In the classic relational database, queries within PL/SQL blocks had pure SQL syntax, and PL/SQL commands supported the structural programming paradigm. The ability to define abstract types together with methods, which types can inherit from each other, forces changes in the syntax of SQL and PL/SQL commands. In addition, the ability to define object views allows one to treat relational databases in the same way as these object-oriented. These changes cause the PL/SQL language to receive a new quality called the PL/SQL object language.

Task. *Define an anonymous block displaying the total mice rations of female cats. Use the previously defined abstract type* `CAT_TYPE` *and the object view* `Cats_with_oid`*.*

```
DECLARE
  cat CAT_TYPE;
  CURSOR mice_of_ladies IS
  SELECT VALUE(CO)
  FROM Cats_with_oid CO
  WHERE CO.About_gender()='Female cat';
BEGIN
  DBMS_OUTPUT.PUT_LINE('Nick of female cat    Salary');
  DBMS_OUTPUT.PUT_LINE('----------------------------');
  OPEN mice_of_ladies;
  LOOP
    FETCH mice_of_ladies INTO cat;
    EXIT WHEN mice_of_ladies%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(RPAD(cat.nickname,21,' ')||' '||
                         cat.Mice_income());
  END LOOP;
  CLOSE mice_of_ladies;
END;

anonymous block completed

Nick of female cat    Salary
----------------------------
LOLA             72
FLUFFY           55
EAR              40
FAST             65
LITTLE           64
HEN              61
MISS             52
LADY             51
```

The VALUE function is used to download data with the structure of an abstract data type. In the above case, the `CAT_TYPE` type object is returned from the `Cats_with_oid` view which is built of row objects of this type. In solving the above task, methods named `About_gender` and `Mice_income` were used, defined for the `CAT_TYPE` type.

Task. *Define an anonymous block that adds a new cat to the herd. After inserting a new member of the herd, add 5 additional mice to cats with a total mice ration less than the average total mice ration for the whole herd. Use the object view* `Cats_with_oid` *for this purpose.*

```
DECLARE
  cat CAT_TYPE:=CAT_TYPE('RYCHO','M','FAT','CAT',
                    'TIGER','2020-02-09',50,NULL,1);
  k CAT_TYPE;
  ma Functions.max_mice%TYPE;
  mi Functions.min_mice%TYPE;
  i NUMBER;
  existing_nickname EXCEPTION;
  out_of_ration EXCEPTION;
BEGIN
  SELECT COUNT(*) INTO i FROM Cats_with_oid
  WHERE nickname=cat.nickname;
  IF i>0 THEN RAISE existing_nickname;
  END IF;
  SELECT max_mice,min_mice INTO ma,mi
  FROM Functions
  WHERE function=cat.function;
-- above, the relation not the object view was used
-- because this view has not been defined for this relation
  IF cat.mice_ration BETWEEN mi AND ma
     THEN INSERT INTO Cats_with_oid VALUES (cat);
     ELSE RAISE out_of_ration;
  END IF;
  SELECT AVG(CO.Mice_income()) INTO i
  FROM Cats_with_oid CO;
  FOR kitten IN (SELECT VALUE(CC) ko
                 FROM Cats_with_oid CC)
  LOOP
    k:=kitten.ko;
    IF k.Mice_income()<i
    THEN UPDATE Cats_with_oid
         SET mice_extra=NVL(mice_extra,0)+5
         WHERE nickname=k.nickname;
    END IF;
```

```
  END LOOP;
EXCEPTION
  WHEN existing_nickname
      THEN DBMS_OUTPUT.PUT_LINE('Nickname already exists!!!');
  WHEN NO_DATA_FOUND
       THEN DBMS_OUTPUT.PUT_LINE('Wrong function!!!');
  WHEN out_of_ration
       THEN DBMS_OUTPUT.PUT_LINE('Mice out of ration!!!');
  WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;

anonymous block completed
```

In the above solution, the code fragment:

```
  SELECT AVG(CO.Mice_income()) INTO i
  FROM Cats_with_oid CO;
  FOR kitten IN (SELECT VALUE(CC) ko
                 FROM Cats_with_oid CC)
  LOOP
    k:=kitten.ko;
    IF k.Mice_income()<i
    THEN UPDATE Cats_with_oid
         SET mice_extra=NVL(mice_extra,0)+5
         WHERE nickname=k.nickname;
    END IF;
  END LOOP;
```

can be replaced with:

```
UPDATE Cats_with_oid K
SET mice_extra=NVL(mice_extra,0)+5
WHERE K.Mice_income()<(SELECT AVG(CC.Mice_income())
                       FROM Cats_with_oid CC);
```

The longer fragment of code was placed to illustrate the use of the explicit cursor to handle relational database data through the object view and to illustrate the use of the method defined within the abstract type `CAT_TYPE`, which is the object view basis (the view consists of rows of this type). In solving the above task, the default constructor for the `CATS_TYPE` type and its method called `Mice_income` was used. As the solutions of the above tasks show, by creating for a relation an abstract type, one can simultaneously extend support of the relation to methods defined within this type. This applies to both SQL and PL/SQL.

Task. *Display nicknames of cats performing the 'CAT' function, their genders and the values of total mice rations, taking into account the modifications made in the previous task.*

```
SELECT nickname "Nickname",
       SUBSTR(C.About_gender(),1,14) "Gender",
       SUBSTR(C.Mice_income(),1,14) "Mice income"
FROM Cats_with_oid C
WHERE function='CAT';

Nickname        Gender         Mice income
--------------- -------------- --------------
FAT             Male cat       55
ZERO            Male cat       48
EAR             Female cat     45
SMALL           Male cat       45

ROLLBACK;

rollback complete.
```

4.4. Nested tables and variable size arrays

Nested tables and variable size tables are the second and third type of collections discussed in this lecture. They are part of the object extension of Oracle database , so they are presented after the fragment of material related to this extension. Earlier, the first type of collection was presented, i.e. index tables.

4.4.1. Nested tables

Nested tables are available since Oracle 8 version. Their name is associated with the fact that the type of this table can be a type of relation attribute, so it can be nested in another table.

The nested table type is defined according to the syntax:

TYPE type_name **AS TABLE OF** table_element_type [**NOT NULL**]

The absence of an INDEX BY BINARY_INTEGER clause which is a feature of the index table, in the definition indicates a nested table. The element type of a nested table can be any scalar type (except the types: BOOLEAN, NCHAR, NCLOB, NVARCHAR2 and REF CURSOR), as well as any object type. Access to the element of a nested table is obtained in a similar way to access to an index table element.

Nested tables gain additional features in comparison to index tables, i.e.

- they can be modified with SQL commands and saved in the database as values of relation attributes (nested table in the table - hence the name),
- negative index values are not available for them and the indexes must be sequential,
- they can be completely indeterminate (NULL). This can be checked using the IS NULL operator,
- additional attributes (methods) have been defined for them:
  – EXTEND, EXTEND(n), EXTEND(n, m) – respectively, appends an indeterminate (NULL) row with the index LAST+1, attaches n empty rows at the end of the table, copies the element with the index n, m times at the end of the table,
  – TRIM, TRIM (n) – respectively, deletes the last row of the table, deletes the last n rows of the table (if n> COUNT then the SUBSCRIPT_BEYOND_COUNT exception appears).

The table nested immediately after the declaration is indeterminate (NULL) and requires initialization (construction). Assigning a value to an uninitialized table causes the exception ORA-6531 COLLECTION_IS_NULL.

A constructor with the same name as the type name is used to initialize the nested table. The following is an example definition of a nested table type and declarations of several variables of this type along with their initialization.

```
TYPE TABLE_OF_NUMBERS IS TABLE OF NUMBER;
tl1 TABLE_OF_NUMBERS:= TABLE_OF_NUMBERS(7);
tl2 TABLE_OF_NUMBERS:= TABLE_OF_NUMBERS(3,4,9);
tl3 TABLE_OF_NUMBERS:= TABLE_OF_NUMBERS();
```

During initialization, table items are indexed from 1 to an index equal to the number of items initialized. The `tl3` variable is a nested array containing no elements but already specified (NOT NULL). To enlarge the table size beyond that resulting from initialization, one should use the EXTEND attribute (method). Without this, attempting to assign a value to a table element outside of the index range will result in exception ORA-6533 SUBSCRIPT_BEYOND_COUNT.

Nested tables as relation attributes

In order for the nested table type to be used as the attribute type of a relation created using the CREATE TABLE command, one must first define the type corresponding to this nested table using the CREATE TYPE command. As part of the CREATE TABLE command, only the type saved in the database dictionary can be used, and the CREATE TYPE command makes it available as one of the possible attribute types.

Task. *Tiger decided to record all offenses of cats, remembering their date and description. For this purpose, he ordered to define a relation whose attributes would be the cat's nickname and a nested table containing all the offenses of that cat. Define such a relation.*

```
CREATE OR REPLACE TYPE OFFENSE_OF_CAT AS OBJECT
(offense_date DATE,
 offense_desc VARCHAR2(50));

TYPE OFFENSE_OF_CAT compiled

CREATE OR REPLACE TYPE LIST_OF_OFFENSES
AS TABLE OF OFFENSE_OF_CAT;

TYPE LIST_OF_OFFENSES compiled

CREATE TABLE Offenses
(nickname VARCHAR2(15) PRIMARY KEY REFERENCES Cats(nickname),
 about_offenses LIST_OF_OFFENSES)
NESTED TABLE about_offenses STORE AS Varehouse_of_offenses;

table OFFENSES created.
```

The NESTED TABLE clause is mandatory. `Varehouse_of_offenses` specifies the name of, generated by the system, the so-called a storage table, used to store actual nested table data. The `about_offenses` attribute is a reference to this table. To the storage table user has no direct access, although its description is in the USER_TABLES view. If one try to modify it, one will receive error ORA-22812. The modification of the storage table can be done by DML commands working on the relation whose attribute is this table.

It is worth noting that the use in relations of array type attributes violates the principles of building relational databases (omits referential relationships), however, it better reflects the structure of real data and significantly speeds up access to data (costly operation of relations joining is omitted).

Operations on relations with nested tables

Relations with nested tables can be read and modified both in pure SQL and as part of PL/SQL. However, the method of performing such operations may be different.

Task. *Insert two rows to the* `Offenses` *relation, one using pure SQL and the other using the PL/SQL block.*

```
INSERT INTO Offenses VALUES
('FAST',
 LIST_OF_OFFENSES(OFFENSE_OF_CAT('2011-01-03',
                          'TOO AGGRESSIVELY DEMANDED PRIZE'),
                 OFFENSE_OF_CAT('2011-01-04',
                          'DISCUSSION DUE TO NO PRIZE')));

DECLARE
-- initialization of the nested table
  the_first_offense LIST_OF_OFFENSES:=
  LIST_OF_OFFENSES(OFFENSE_OF_CAT('2011-01-01',
                                  'EATING OF HUNTED MOUSE'));
BEGIN
  INSERT INTO Offenses VALUES('ZERO',the_first_offense);
END;
anonymous block completed
```

Task. *Assuming that the cat with the nickname 'ZERO' was already noted, modify* `Offenses` *relation by adding to him a new one offense. The task should be performed using pure SQL and using the PL/SQL block.*

```
INSERT INTO TABLE(SELECT about_offenses
                  FROM Offenses
                  WHERE nickname='ZERO')
VALUES(OFFENSE_OF_CAT('2011-01-10','EATING OF HUNTED MOUSE'));

1 rows inserted.

ROLLBACK;
rollback complete.

DECLARE
  table_of_offenses Offenses.about_offenses%TYPE;
  new_offense OFFENSE_OF_CAT:=
              OFFENSE_OF_CAT('2011-01-10',
                             'EATING OF HUNTED MOUSE');
BEGIN
  SELECT about_offenses INTO table_of_offenses
  FROM Offenses WHERE nickname='ZERO';
  table_of_offenses.EXTEND;
  table_of_offenses(table_of_offenses.COUNT):=new_offense;
  UPDATE Offenses
  SET about_offenses=table_of_offenses
  WHERE nickname='ZERO';
END;

anonymous block completed
```

In the above solution, for pure SQL, the TABLE operator is used, which is available only for SQL language, returns the nested table downloaded using a subquery (the THE operator does the same, however Oracle recommends using the TABLE operator). The table downloaded in this way can be modified with DML commands. The solution for PL/SQL, due to the lack of access to the TABLE operator, is different. First, the nested table for the cat nickname `'ZERO'` is downloaded, then, using the EXTEND method, table obtains a new row, which receives value of the new offense. The last step is to modify the `Offenses` relation for the `'ZERO'` cat, where its entire offense table is overwritten.

Task. *Display the offenses of a specified cat. The task should be performed using pure SQL and using the PL/SQL block.*

```
SELECT nickname "Culprit",offense_date "Date",
       offense_desc "Offense"
FROM Offenses O,
     TABLE(SELECT about_offenses
           FROM Offenses WHERE nickname='&nickname')
WHERE O.nickname='ZERO';

NICKNAME - ZERO

Culprit         Date                Offense
--------------- ------------------- ---------------------------
ZERO            2011-01-01          EATING OF HUNTED MOUSE
ZERO            2011-01-10          EATING OF HUNTED MOUSE


DECLARE
  table_of_offenses Offenses.about_offenses%TYPE;
  o Offenses.nickname%TYPE:='&nickname';
BEGIN
  SELECT about_offenses INTO table_of_offenses FROM Offenses
  WHERE nickname=o;
  DBMS_OUTPUT.PUT_LINE
              ('Offenses of the cat with the nickname '||o);
  FOR i IN 1..table_of_offenses.COUNT
  LOOP
    DBMS_OUTPUT.PUT(table_of_offenses(i).offense_date);
    DBMS_OUTPUT.PUT_LINE
                ('  '||table_of_offenses(i).offense_desc);
  END LOOP;
EXCEPTION
      WHEN NO_DATA_FOUND THEN DBMS_OUTPUT.PUT_LINE
                                        ('Wrong nickname');
END;

NICKNAME - ZERO

Offenses of the cat with the nickname ZERO
2011-01-01  EATING OF HUNTED MOUSE
2011-01-10  EATING OF HUNTED MOUSE
```

Assigning a nested table to a PL/SQL variable causes allocating for its indexes from 1 to COUNT.

4.4.2 Tables of variable size

Tables of variable size are available since Oracle 8 version and are similar to tables of C language. They are defined according to the syntax:

TYPE type_name **AS** {**VARRAY** | **VARYING ARRAY**} (size)
OF table_element_type [**NOT NULL**];

Type of table element cannot be BOOLEAN, NCHAR, NCLOB, NVARCHAR2, REF CURSOR (collections are available as of Oracle 9i version). Access to the element of a variable size table is obtained in a similar way to access to element of an index table and nested table.

Tables of variable size have additional features compared to nested tables. Those are:

- maximum size specified,
- stored in the database retain the way of ordering and index values,
- in the database they are stored in the same area as the relation with attribute of this type (in the CREATE TABLE there is no NESTED TABLE clause),
- one cannot use the TRIM attribute (method) to delete their items,
- an additional attribute (method) is defined for them: LIMIT - returns the maximum size of the array.

Similarly to nested tables, tables of variable size should be initialized and the addition of another element within PL/SQL involves using the EXTEND method.

Tables of variable size as relation attributes

Tables of variable size, similarly like nested tables, can be a type of relation attribute, but there is no way within SQL to modify their elements (the TABLE and THE operators mentioned above). Therefore, they should always be treated as a whole (any modification requires their overwriting).

Task. *Define an anonymous block that will allow to remember cats offences in a relation whose attributes will be the cat's nickname and the list of offenses being a table of variable size.*

```
CREATE OR REPLACE TYPE LIST_OF_OFFENSES1
AS VARRAY(20) OF OFFENSE_OF_CAT;

TYPE LIST_OF_OFFENSES1 compiled

CREATE TABLE Offenses1
(nickname VARCHAR2(15)PRIMARY KEY REFERENCES Cats(nickname),
 about_offenses LIST_OF_OFFENSES1);

table OFFENSES1 created.


DECLARE
  table_of_offenses LIST_OF_OFFENSES1:=LIST_OF_OFFENSES1();
   o Offenses1.nickname%TYPE:='&nickname_of_cat';
   new_offense
OFFENSE_OF_CAT:=OFFENSE_OF_CAT('&date_of_offense',
                               '&description_of_offense');
   np NUMBER;
BEGIN
  SELECT COUNT(*) INTO np FROM Cats WHERE nickname=o;
  IF np=0 THEN
      RAISE_APPLICATION_ERROR(-20101,'Wrong nickname');
  END IF;
  SELECT COUNT(*) INTO np FROM Offenses1 WHERE nickname=o;
  IF np=0 THEN
     table_of_offenses.EXTEND;
     table_of_offenses(1):=new_offense;
     INSERT INTO Offenses1 VALUES (o,table_of_offenses);
  ELSE
     SELECT about_offenses INTO table_of_offenses
     FROM Offenses1 WHERE nickname=o;
     IF table_of_offenses.COUNT=table_of_offenses.LIMIT THEN
        RAISE_APPLICATION_ERROR
             (-20102,'Fxhausted limit of offenses');
     END IF;
     table_of_offenses.EXTEND;
     table_of_offenses(table_of_offenses.COUNT):=new_offense;
     UPDATE Offenses1 SET about_offenses=table_of_offenses
     WHERE nickname=o;
  END IF;
END;
```

```
NICKNAME_OF_CAT – LOLA
DATE_OF_OFFENSE – 2020-03-01
DESCRIPTION_OF_OFFENSE - SHE DID NOT ACCEPT THE GIFT FROM
                          THE CHIEF

anonymous block completed

NICKNAME_OF_CAT – LOLA
DATE_OF_OFFENSE – 2020-03-02
DESCRIPTION_OF_OFFENSE - SHE DID NOT APPEAR ON THE CHIEF CALL

anonymous block completed
```

Although elements of tables of variable cannot be modified using DML commands, one can formulate queries to them, in the same way as queries to nested tables, using the TABLE operator.

Task. *Display the offenses of the cat with the nickname* `'LOLA'`*, saved as a table of variable size in the relation* `Offenses1`*.*

```
SELECT nickname "Culprit",offense_date "Data",
       offense_desc "Vice"
FROM Offenses1 O1,TABLE(SELECT about_offenses
                        FROM Offenses1 WHERE nickname='LOLA')
WHERE O1.nickname='LOLA';

Culprit    Data          Vice
---------- ------------- ----------------------------------------
LOLA       2020-03-01    SHE DID NOT ACCEPT THE GIFT FROM THE CHIEF
LOLA       2020-03-02    HE DID NOT APPEAR ON THE CHIEF CALL
```

5. Bulk binding

The PL/SQL code is executed by the PL/SQL machine on the server or client side. However, regardless of on which side the block is being executed, the always contained therein "clean" SQL commands are sent to the executor of those commands, which is located in the DBMS. The resulting data is then sent back to the PL/SQL machine. This transfer process, called context switching, decrease the performance of the code being executed. This performance especially decreases when the DML command is executed within a loop (the number of context switching equals the number of loop circuits). The way to solve this problem is to use the so-called bulk binding (introduced since Oracle 8i version) consisting of inserting data modifying the database into the collection, and then sending the entire collection to the SQL machine, which performs modification. This reduces the number of context switching to once.

Bulk DML commands are implemented using the FORALL command with the following syntax:

FORALL collection_index **IN** begining_index .. final_index
[**SAVE EXCEPTION**]
DML_command;

where index is of type collection index, begining_index and final_index are the number of the first and last item of the collection, respectively. A collection item with the index collection_index is part of the DML command body (defines a new attribute value). In one switch of context, all data is updated in a bulk way using the contents of the collection. The SAVE EXCEPTION clause, introduced since Oracle 9i version, allows handling errors at the collection line level (new exception ORA-2481: error(s) in array DML). It writes errors to the implicit cursor attribute SQL% BULK_EXCEPTION, and then allows the FORALL statement to continue processing the remaining lines. This attribute resembles a PL/SQL table built of records containing error_index and error_code fields indicating the error line number and error code, respectively (the error description is returned by SQLERRM(-SQL%BULK_EXCEPTIONS(i).error_code)) function. As a table, it has the already known COUNT attribute.

Task. *Tiger come to the conclusion that his secret account created to fight the conspiracy could also be used to privately supplement his state. Enable the Tiger to enter any number of mice into its secret account.*

```
CREATE OR REPLACE
PROCEDURE for_tiger(nm NUMBER) AS
  TYPE td IS TABLE OF DATE INDEX BY BINARY_INTEGER;
  m td;
  ne NUMBER;
  wrong_number_of_mice EXCEPTION;
BEGIN
  IF nm<=0 THEN RAISE wrong_number_of_mice; END IF;
  FOR i IN 1..nm
  LOOP
    m(i):=SYSDATE;
  END LOOP;
  FORALL i IN 1..nm SAVE EXCEPTIONS
  INSERT INTO Tiger(entry_date) VALUES (m(i));
EXCEPTION
  WHEN wrong_number_of_mice THEN NULL;
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE
               ('An exception occurred: '||SQLERRM);
    ne:=SQL%BULK_EXCEPTIONS.COUNT;
    FOR i IN 1..ne
    LOOP
      DBMS_OUTPUT.PUT_LINE('Error '||i||': mouse '||
           SQL%BULK_EXCEPTIONS(i).error_index||' - '||
           SQLERRM(-SQL%BULK_EXCEPTIONS(i).error_code));
    END LOOP;
END for_tiger;

PROCEDURE for_tiger compiled

EXEC for_tiger(7);
anonymous block completed

SELECT COUNT(*)-COUNT(release_date) "On account"
FROM Tiger;

On account
----------------------
7

ROLLBACK;
rollback complete.
```

Task. *The fight against conspiracy required strict control over the state of the mice in the warehouse, hence it became necessary to monitor each change in the ration of mice and ration extra. Write a COMPOUND trigger, which in bulk way saves data of changes of these rations (who, to whom, when, what operation, value before change, value after change) in the* `Changes_of_rations` *relation.*

```
CREATE TABLE Changes_of_rations
(who VARCHAR2(15),
 whom VARCHAR2(15),
 when_ch DATE,
 operation VARCHAR2(6),
 ration_old NUMBER(3),
 ration_new NUMBER(3),
 extra_old NUMBER(3),
 extra_new NUMBER(3));
```

```
table CHANGES_OF_RATIONS created.
```

```
CREATE OR REPLACE TRIGGER Mice_control
FOR INSERT OR UPDATE OF mice_ration, mice_extra
ON Cats
COMPOUND TRIGGER
  TYPE changes_t IS TABLE OF Changes_of_rations%ROWTYPE
                INDEX BY SIMPLE_INTEGER;
  rhz  changes_t;
  ind  SIMPLE_INTEGER := 0;
  nmax CONSTANT SIMPLE_INTEGER := 1000;

  PROCEDURE write_changes
  IS
    lwpis CONSTANT SIMPLE_INTEGER := rhz.COUNT();
  BEGIN
    FORALL nr IN 1..lwpis
    INSERT INTO Changes_of_rations VALUES rhz(nr);
    rhz.delete();
    ind := 0;
  END write_changes;

  AFTER EACH ROW
  IS
  BEGIN
    ind := ind + 1;
    rhz(ind).who:=SYS_CONTEXT('USERENV', 'SESSION_USER');
    rhz(ind).whom := :NEW.nickname;
    rhz(ind).when_ch := SYSDATE;
    rhz(ind).ration_new := :NEW.mice_ration;
    rhz(ind).extra_new := :NEW.mice_extra;
```

```
    IF INSERTING THEN
       rhz(ind).operation := 'INSERT';
       rhz(ind).ration_old := NULL;
       rhz(ind).extra_old := NULL;
    ELSE
       rhz(ind).operation := 'UPDATE';
       rhz(ind).ration_old := :OLD.mice_ration;
       rhz(ind).extra_old := :OLD.mice_extra;
    END IF;
    IF ind >= nmax THEN
       write_changes();
    END IF;
  END AFTER EACH ROW;

  AFTER STATEMENT
  IS
  BEGIN
    write_changes();
  END AFTER STATEMENT;

END Mice_control;

TRIGGER Mice_control compiled

INSERT INTO Cats
VALUES('RYCHO','M','FAT','HONORARY','TIGER','2020-05-
09',10,2,1);
INSERT INTO Cats
VALUES('SOPHIE','W','SKINNY','HONORARY','TIGER','2020-05-
10',5,NULL,1)

1 rows inserted.
1 rows inserted.

UPDATE Cats
SET mice_ration=20,
    mice_extra=5
WHERE nickname<>'TIGER';

19 rows updated.
```

```
SELECT * FROM Changes_of_rations;

WHO   WHOM       WHEN_CH         OPERATION  RATION_OLD   RATION_NEW   EXTRA_OLD    EXTRA_NEW

----- ---------- --------------- ---------- ------------ ------------ ------------ -----------
Z     FAT        2020-05-10      INSERT                  10                        2
Z     SKINNY     2020-05-10      INSERT                   5
Z     FAT        2020-05-10      UPDATE     10           20            2           5
Z     SKINNY     2020-05-10      UPDATE     5            20                        5
Z     CAKE       2020-05-10      UPDATE     67           20                        5
Z     TUBE       2020-05-10      UPDATE     56           20                        5
Z     LOLA       2020-05-10      UPDATE     25           20            47          5
Z     ZERO       2020-05-10      UPDATE     43           20                        5
Z     FLUFFY     2020-05-10      UPDATE     20           20            35          5
Z     EAR        2020-05-10      UPDATE     40           20                        5
Z     SMALL      2020-05-10      UPDATE     40           20                        5
Z     BOLEK      2020-05-10      UPDATE     50           20                        5
Z     ZOMBIES    2020-05-10      UPDATE     75           20            13          5
Z     BALD       2020-05-10      UPDATE     72           20            21          5
Z     FAST       2020-05-10      UPDATE     65           20                        5
Z     LITTLE     2020-05-10      UPDATE     22           20            42          5
Z     REEF       2020-05-10      UPDATE     65           20                        5
Z     HEN        2020-05-10      UPDATE     61           20                        5
Z     MISS       2020-05-10      UPDATE     24           20            28          5
Z     MAN        2020-05-10      UPDATE     51           20                        5
Z     LADY       2020-05-10      UPDATE     51           20                        5

 21 rows selected

ROLLBACK;
rollback complete.
```

As part of the trigger implementing above task, the AFTER row part prepares data to the modification, the AFTER command part saves the data.

The implicit cursor used by the bulk DML, during the FORALL command, has an additional SQL%BULK_ROWCOUNT attribute with semantics of index table. The SQL%BULK_ROWCOUNT(i) element stores the number of rows processed during execution of the i-th DML command. If the i-th execution does not affect rows, 0 is returned. Although the attribute discussed has semantics of index table, the methods associated with that table cannot be used for it.

In addition to bulk DML, the bulk binding can be also used for bulk queries. The BULK COLLECT clause is used in the following PL/SQL commands: SELECT, FETCH, and DML commands with the RETURNING INTO clause. The BULK COLLECT clause appears in each of these commands before the INTO keyword. After INTO here, however, there is no list of variables, but a list of collections into which are inserted values returned by commands.

The RETURNING INTO clause, which has not been discussed so far, is used when information about the rows modified by these commands is needed. One way to get this information is to use the SELECT command after running the DML command. However then, it is necessary to make another reference to the SQL machine, which is not optimal. After extending the syntax of all DML commands in Oracle 8 version to include the RETURNING INTO clause, this can be done during one reference to the system kernel as part of the DML command. The syntax discussed is as follows:

DML_command
[**RETURNING** {expression [, …]} **INTO** {variable [, …]}];

The information represented by the PL/SQL or SQL list of expressions, separated by commas, is downloaded to a list of SQL variables separated by commas. Corresponding variables and expressions must be of compatible types. The expression values downloaded include the modifications made by the DML command.

Task. *To enable the topping up of his secret account, the Tiger decided to take back one mouse from mice ration of each cat. The pretext was to be punishment for too slow adaptation to EU standards in terms of the length of hunted mice (notorious eating of undersized mice in place). Using bulk binding, display current rations of mice for all cats, change them, and then display their values after change.*

```
DECLARE
  TYPE tn IS TABLE OF Cats.nickname%TYPE;
  TYPE tm IS TABLE OF Cats.mice_ration%TYPE;
  tab_nn tn:=tn();
  tab_mi tm:=tm();
  i BINARY_INTEGER;
  PROCEDURE mice
  IS
  BEGIN
    FOR i IN 1..tab_nn.COUNT
    LOOP
      DBMS_OUTPUT.PUT_LINE(RPAD(tab_nn(i),12)||
                           '   '||tab_mi(i));
    END LOOP;
  END;
```

```
BEGIN
  SELECT nickname,mice_ration
  BULK COLLECT INTO tab_nn,tab_mi
  FROM Cats WHERE nickname!='TIGER';
  DBMS_OUTPUT.PUT_LINE('
       Mice before change
                      ');
  mice;
  UPDATE Cats
  SET mice_ration=mice_ration-1
  WHERE nickname!='TIGER'
  RETURNING nickname,mice_ration
  BULK COLLECT INTO tab_nn,tab_mi;
  DBMS_OUTPUT.PUT_LINE('
       Mice after change
                      ');
  mice;
END;

anonymous block completed

       Mice before change

CAKE           67
TUBE           56
LOLA           25
ZERO           43
FLUFFY         20
EAR            40
SMALL          40
BOLEK          50
ZOMBIES        75
BALD           72
FAST           65
LITTLE         22
REEF           65
HEN            61
MISS           24
MAN            51
LADY           51
```

```
     Mice after change

CAKE           66
TUBE           55
LOLA           24
ZERO           42
FLUFFY         19
EAR            39
SMALL          39
BOLEK          49
ZOMBIES        74
BALD           71
FAST           64
LITTLE         21
REEF           64
HEN            60
MISS           23
MAN            50
LADY           50

ROLLBACK;
rollback complete.
```

The above task can be solved in an equivalent way by replacing the bulk SELECT query with an explicit cursor from which the data is retrieved with the bulk FETCH command. The code of PL/SQL block, after these modifications, is as follows:

```
DECLARE
  TYPE tn IS TABLE OF Cats.nickname%TYPE;
  TYPE tm IS TABLE OF Cats.mice_ration%TYPE;
  tab_nn tn:=tn();
  tab_mi tm:=tm();
  i BINARY_INTEGER;
  CURSOR choice IS
  SELECT nickname,mice_ration
  FROM Cats
  WHERE nickname!='TIGER';
  PROCEDURE mice
  IS
  BEGIN
    FOR i IN 1..tab_nn.COUNT
    LOOP
      DBMS_OUTPUT.PUT_LINE(RPAD(tab_nn(i),12)||
                           '   '||tab_mi(i));
    END LOOP;
  END;
```

```
BEGIN
  OPEN choice;
  FETCH choice BULK COLLECT INTO tab_nn,tab_mi;
  CLOSE wybor;
  DBMS_OUTPUT.PUT_LINE('
       Mice before change
                        ');
  mice;
  UPDATE Cats
  SET mice_ration=mice_ration-1
  WHERE nickname!='TIGER'
  RETURNING nickname,mice_ration
  BULK COLLECT INTO tab_nn,tab_mi;
  DBMS_OUTPUT.PUT_LINE('
       Mice after change
                        ');
  mice;
END;
```

Bulk dynamic SQL

Since Oracle 9i version, internal dynamic SQL has been expanded for the SELECT command, DML commands and FETCH command to possibilities of bulk SQL.

Syntax of the bulk dynamic version of SELECT statement is as follows:

EXECUTE IMMEDIATE string_expression
BULK COLLECT INTO {collection [, …]};

where the string expression defines the SELECT statement (without the INTO clause!)

Syntax of the bulk dynamic version of DML commands is as follows:

FORALL collection_index **IN** begining_index .. final_index
EXECUTE IMMEDIATE string_expression
USING {bound_element_of_collection_with_index_collection_index
[, …]}
[**RETURNING BULK COLLECT INTO** {collection [, …]}];

where string expression defines a DML command that contains bound variables preceded by colon character, corresponding to proper elements of the bound collections (USING clause). A DML command defined by a string can contain its internal RETURNING clause with a list of returned values. The returned values are then substituted into the corresponding elements of the collection, which (collections) are specified after the RETURNING BULK COLLECT clause (the last clause is then required).

Syntax of the bulk dynamic version of the FETCH command is as follows:

FETCH cursor_variable
BULK COLLECT INTO {collection [, …]};

where the cursor variable is open using the OPEN ... FOR ... command from native dynamic SQL.

Task. *Solve the task of creating individual secret mice accounts using dynamic bulk SQL.*

```
DECLARE
  TYPE tn IS TABLE OF Cats.nickname%TYPE;
  TYPE tl IS TABLE OF NUMBER;
  TYPE td IS TABLE OF DATE;
  tab_nn tn:=tn();
  tab_le tl:=tl();
  tab_de td:=td();
  text_cur VARCHAR2(200):=
           'SELECT level,nickname
            FROM Cats
            START WITH chief IS NULL
            CONNECT BY PRIOR nickname=chief';
  cur cursor.c;
  maxl NUMBER(2):=0;
  chow_much NUMBER(4);
  jt NUMBER(1);
BEGIN
  OPEN cur FOR text_cur;
  FETCH cur BULK COLLECT INTO tab_le,tab_nn;
  CLOSE cur;
```

```
  FOR j IN 1..tab_le.COUNT
  LOOP
    IF tab_le(j)>maxl THEN maxl:=tab_le(j); END IF;
    SELECT COUNT(*) INTO jt FROM User_tables
    WHERE table_name=tab_nn(j);
    IF jt=1 THEN EXECUTE IMMEDIATE 'DROP TABLE '||tab_nn(j);
    END IF;
    EXECUTE IMMEDIATE 'CREATE TABLE '||tab_nn(j)||'
               (entry_date DATE,release_date DATE)';
  END LOOP;
  FOR j IN 1..tab_le.COUNT
  LOOP
    tab_de.TRIM(tab_de.COUNT);
    FOR i IN 1..maxl-tab_le(j)+1
    LOOP
      tab_de.EXTEND; tab_de(i):=SYSDATE;
    END LOOP;
    FORALL i IN 1..maxl-tab_le(j)+1
    EXECUTE IMMEDIATE 'INSERT INTO '||tab_nn(j)||
                ' (entry_date) VALUES (:en_da)'
    USING tab_de(i);
  END LOOP;
  FOR j IN 1..tab_le.COUNT
  LOOP
    EXECUTE IMMEDIATE
    'SELECT COUNT(*)-COUNT(release_date) FROM '||tab_nn(j)
    INTO chow_much;
    DBMS_OUTPUT.PUT_LINE(RPAD(tab_nn(j),10)||
              ' - Number of available mice: '||chow_much);
  END LOOP;
END;

anonymous block completed

TIGER      - Number of available mice: 4
BALD       - Number of available mice: 3
CAKE       - Number of available mice: 2
FAST       - Number of available mice: 2
MISS       - Number of available mice: 2
TUBE       - Number of available mice: 2
BOLEK      - Number of available mice: 3
LITTLE     - Number of available mice: 3
LOLA       - Number of available mice: 3
REEF       - Number of available mice: 3
EAR        - Number of available mice: 2
LADY       - Number of available mice: 2
MAN        - Number of available mice: 2
SMALL      - Number of available mice: 2
ZOMBIES    - Number of available mice: 3
FLUFFY     - Number of available mice: 2
HEN        - Number of available mice: 2
ZERO       - Number of available mice: 1
```

Task. *After taking away one mouse from the basic ration of each cat, conscience said to Tiger that this not good. So he decided to increase each cat's ration by one mouse. Perform this task using bulk dynamic SQL. Display the total mouse ration after its increase.*

```
DECLARE
  TYPE tn IS TABLE OF Cats.nickname%TYPE;
  TYPE te IS TABLE OF Cats.mice_extra%TYPE;
  TYPE tt IS TABLE OF NUMBER(5);
  tab_nn tn:=tn();
  tab_me te:=te();
  tab_rt tt:=tt();
  cur cursor.c;
  text_cur VARCHAR2(200):='SELECT nickname,NVL(mice_extra,0)+1
                           FROM Cats';
  maxl NUMBER(2):=0;jt NUMBER(1);
BEGIN
  OPEN cur FOR text_cur;
  FETCH cur BULK COLLECT INTO tab_nn,tab_me;
  CLOSE cur;
  FORALL i IN 1..tab_nn.COUNT
  EXECUTE IMMEDIATE
  'UPDATE Cats
   SET mice_extra=:tme
   WHERE nickname=:tps
   RETURNING NVL(mice_ration,0)+NVL(mice_extra,0)INTO :total'
  USING tab_me(i),tab_nn(i)
  RETURNING BULK COLLECT INTO tab_rt;
  DBMS_OUTPUT.PUT_LINE('
  Nickname       Mice
  --------------------');
  FOR i IN 1..tab_nn.COUNT
  LOOP
    DBMS_OUTPUT.PUT_LINE('  '||RPAD(tab_nn(i),12)||
                                 '   '||tab_rt(i));
  END LOOP;
END;

anonymous block completed
```

```
Nickname       Mice
--------------------
CAKE           68
TUBE           57
LOLA           73
ZERO           44
FLUFFY         56
EAR            41
SMALL          41
TIGER          137
BOLEK          51
ZOMBIES        89
BALD           94
FAST           66
LITTLE         65
REEF           66
HEN            62
MISS           53
MAN            52
LADY           52
```