Database and Information Systems

Michal Krátký

Department of Computer Science
Faculty of Electrical Engineering and Computer Science
VŠB – Technical University of Ostrava

2020/2021

Content

Content

Object-oriented DBMS

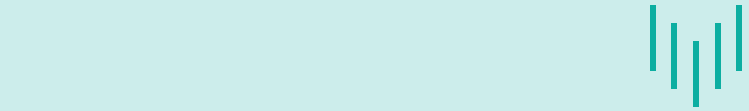
- **Object-oriented DBMS**: Object Data Management Group (ODMG)[1], Object Query Language (OQL).
- Expensive migration of relational data, as result some features of object-oriented technologies have been adopted by database industry (SQL99).
  $\Rightarrow$ **object-relational data model** (ORDM).
- Since the standard have been released after vendors of DBMS released own object-oriented extensions, we must talk about concrete implementations (Oracle and so on).
- Now, each DBMS includes some object-oriented features.

[1] `http://www.odmg.org/`

Object-oriented DBMS

- In 90's, vendors of DBMS integrated some object-oriented features into their DBMS:
  - User defined data types (including attributes and methods).
  - Stored procedures and triggers: the code is migrated into DBMS.
  - Collections (variable-length array, nested tables, ...).
  - The inheritance has been introduced.
  - Referencing and dereferencing.
- Some of these features have been integrated into SQL99.

Why Object-relational Data Models?



- Object types and their methods are stored in a database together with data.
- We can use collections and other more comlex data types.
- We can use references instead of join operations.
- The code is executed in a DBMS – a network is not a bottleneck.

Object Data Types

- Object data types include:
  - data – **attributes**
  - operations – **methods**

Object-relational data model – data are viewed by two principles: **relational** (records, relations) as well as **object-oriented** (objects, attributes, methods).

Example 1/3

We create an object data type `person_type` with attributes and two methods.

```
CREATE TYPE person_type AS OBJECT (
   idno NUMBER,
   first_name VARCHAR2(20),
   last_name VARCHAR2(25),
   email VARCHAR2(25),
   phone VARCHAR2(20),
  MAP MEMBER FUNCTION get_idno RETURN NUMBER,
   MEMBER PROCEDURE display (SELF IN OUT
                             NOCOPY person_type));
/
```

Example 2/3

Now, we must define an implementation. The `get_idno` method returns a unique number of a record.

```
CREATE TYPE BODY person_type AS
   MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
   BEGIN
      RETURN idno;
   END;
```

Example 3/3

The `display` method shows values of all attributes of an object.

```
  MEMBER PROCEDURE display (SELF IN OUT
                            NOCOPY person_type) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' ' ||
                         first_name || ' ' || last_name);
    DBMS_OUTPUT.PUT_LINE(email || ' ' || phone);
  END;
END;
/
```

Discussion

- Ok, it seems that these methods are not usable ...
- ... since ...
  - We can retrieve `idno` by a select query:
    `SELECT idno FROM Person ...`
  - We can retrieve values of all attributes by a select query:
    `SELECT * FROM Person ...`
- However, in some cases it is useful to use methods, but we rather use a relation table together with a stored procedure.

Instance of Object Data Type

Object data types can be used as the data type of an attribute.

```
CREATE TABLE contacts (
   contact person_type,
   contact_date DATE);
```

A record is inserted using an SQL command:

```
INSERT INTO contacts VALUES (
   person_type(65, 'Verna', 'Mills',
      'vmills@example.com', '1-650-555-0125'),
   '24 Jun 2003');
```

Methods

Methods:

- *Member methods* – methods of an object
- *Class methods* – methods of a data type
- *Constructors* – the default constructor is defined for each object data type

Method invocation, Example:

```
SELECT c.contact.get_idno() FROM contacts c;
```

Objects in tables 1/2

Two types of objects in a database table:

- **Object tables** – they include only objects: each record is an object. We call it as the *row object*.
  Objects referenced by other objects have to be stored in an object table.
- **Relational tables** – One or more attributes of a table have an object data type. We call it as the *column object*.

Objects in tables 2/2

Examples:

- **Object table:**

```
CREATE TABLE person_obj_table OF person_type;
```

- **Relational table:**

```
CREATE TABLE contacts (
    contact person_type,
    contact_date DATE);
```

Integrity Constraints for object tables

Like in the case of tables in the relational data model, we can define integrity constraints for object tables, for example:

```
CREATE OR REPLACE TYPE location_type AS OBJECT (
   building_no NUMBER,
   city VARCHAR2(40));
/
CREATE OR REPLACE TYPE office_type AS OBJECT (
   office_id VARCHAR(10),
   office_loc location_type,
   occupant person_type);
/
CREATE TABLE office_tab OF office_type (
   office_id PRIMARY KEY);
```

We can also create indices and triggers for objects tables.

Object tables 1/2

There are two views on the object table:

1. It is a table with one column where **each record is an instance** of an object data type and we can use methods of this type.

```
DECLARE
   person person_type;
BEGIN
   SELECT VALUE(p) INTO person
     FROM person_obj_table p WHERE p.idno = 101;
   person.display();
END;
/
```

`VALUE` returns the OID of the record.

Object tables 2/2

There are two views on the object table, in this way, we see a compromise between relational and object data models:

2. It is a **table including attributes** of the object data type and we can use common database operations.

```
INSERT INTO person_obj_table VALUES (
  person_type(101, 'John', 'Smith',
    'jsmith@example.com', '1-650-555-0135'));
```

Object Identifier

- **Object Identifier** (**OID**) identifies objects of object tables.
- OID is not directly accessible, we must use a reference (data type `REF`).
- DBMS automatically generates OID for objects/records of object tables.
- Records of relational tables including column objects are identified by a primary key, in this case we do not need OID for an identification.

Object Reference

- A pointer or reference on an object of an object table is represented by the `REF` data type.
- `REF` references an object of the object data type or it contains the `null` value.
- The reference therefore can replace a foreign key implementing a relationship of entity types in relational data model.

Object Reference, Example 1/2

```
CREATE TYPE emp_person_type AS OBJECT (
   name VARCHAR2(30),
   manager REF emp_person_type);
/
CREATE TABLE emp_person_obj_table OF
                                  emp_person_type;
```

Comment: We create an object table `emp_person_obj_table` including instances of `emp_person_type`. The manager of a person (the atribut `manager`) is a reference on an object of the type `emp_person_type`.

Object Reference, Example 2/2

```
INSERT INTO emp_person_obj_table VALUES (
  emp_person_type('John Smith', NULL));
```

Comment: We insert an instance of `emp_person_type` in the table.

```
INSERT INTO emp_person_obj_table
  SELECT emp_person_type('Bob Jones', REF(e))
    FROM emp_person_obj_table e
    WHERE e.name='John Smith';
```

Comment: We insert an object `Bob Jones` with the manager `John Smith`.

The form of reference?

```
COLUMN name FORMAT A10
COLUMN manager FORMAT A50
SELECT * FROM emp_person_obj_table e;
```

```
NAME       MANAGER
---------- --------------------------------------------------
John Smith
Bob Jones  0000220208424E801067C2EABBE040578CE70A0707424E8010
           67C1EABBE040578CE70A0707
```

Scoped REF

- When a table includes a reference on an object only one table, we can use **SCOPE IS**:

```
CREATE TABLE contacts_ref (
  contact_ref REF person_type
                SCOPE IS person_obj_table,
  contact_date DATE);

INSERT INTO contacts_ref
  SELECT REF(p), '26 Jun 2003'
    FROM person_obj_table p
    WHERE p.idno = 101;
```

- As result, the byte size of the reference is smaller and it produces lower overhead.

Dereference

- We can access an object using dereferencing of a reference:

```
SELECT DEREF(e.manager)
  FROM emp_person_obj_table e;

DEREF(E.MANAGER)(NAME, MANAGER)
----------------------------------------
EMP_PERSON_TYP('John Smith', NULL)
```

- In the case of the relational data model, we must use join to retrieve the record of the manager.

Implicit Dereferencing

- Implicit dereferencing:

```
SELECT e.name, e.manager.name
  FROM emp_person_obj_table e
  WHERE e.name = 'Bob Jones';
```

- In the case of the relational data model, we must use join to retrieve the name of the manager.

Getting a reference on an object table record

```
 1  DECLARE
 2     person_ref REF person_type;
 3     person person_type;
 4  BEGIN
 5     SELECT REF(p) INTO person_ref
 6        FROM person_obj_table p
 7        WHERE p.idno = 101;
 8
 9     SELECT DEREF(person_ref) INTO person FROM dual;
10     person.display();
11  END;
12  /
```

Comments:

Lines 5–7: getting an object reference

Lines 9 and 10: dereferencing and a method invocation

Example, Cursor reference

```
DECLARE
   TYPE rc IS REF CURSOR;
   v_rc rc;
   v_dummy ALL_OBJECTS.OBJECT_NAME%type;
   v_start NUMBER DEFAULT DBMS_UTILITY.GET_TIME;
BEGIN
   FOR i IN 1 .. 1000
   LOOP
      OPEN v_rc FOR
         'select object_name from all_objects
            where object_id = ' || i;
      FETCH v_rc INTO v_dummy;
      CLOSE v_rc;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE(round(
      (DBMS_UTILITY.GET_TIME-v_start)/100, 2) || ' s' );
END;
/
```

Inheritance

- In object-relational DBMS we can create a hierarchy of data types using inheritance.
- All properties of object oriented technologies are than available, for example polymorphism.

```
1  CREATE TYPE student_type UNDER person_type (
2     dept_id NUMBER,
3     major VARCHAR2(30),
4     OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2)
5  NOT FINAL;
6  /
```

Comment: `student_type` inherits from the type `person_type`

Alter of data types

- Like in the case of a schema a relation table using **ALTER TABLE**, we can change an object data type (to add attributes, methods and so on) using **ALTER TYPE**.
- In this case, the change has to be propagated in the complete object hierarchy.

Data Types

- A record (a structure) – created by `%ROWTYPE`
- Collections:
  - *Associative array* – similar to a hash table.
  - *Nested table* – it can be stored in a persistent table.
  - *varray* – *variable-size array* – it is the fixed-length array, it can be stored in a persistent table.
- Variables of these data types can be used in functions and procedures or they can be passed as parameters.

Varray

- It seems that the size of the array is variable, but the size is fixed.
- We define the capacity of the array, when the array is created.
- The current size of the array can be increased up-to the capacity.
- Random accesses to items of the array, e.g. `myArray(3)`.

Using Array

Creating of the type:

```
TYPE type_name IS {VARRAY | VARYING ARRAY} (size_limit)
    OF element_type [NOT NULL];
```

where:

- `type_name` – a name of the data type
- `element_type` – a data type of items of the array
- `size_limit` – the array capacity

Example

```
DECLARE
    -- we create a type (of array)
    -- including up-to 366 dates
    TYPE Calendar IS VARRAY(366) OF DATE;
BEGIN
    NULL;
END;
/
```

Nested table

- An array of the variable size.
- There is no ordering of items in the array, therefore the random access to items of the array is not possible.
- However, the items are numbered when the nested table is read from disk. After this, the random access to items of the nested table is possible.
- The nested table can be sparse; we can use the DELETE operation and to create an empty item.

Using Nested Table

Creating a type:

TYPE `type_name` **IS TABLE** OF `element_type` [NOT NULL];

where:

- `type_name` – a name of the data type
- `element_type` – a data type of items of the nested table

Associative Array

- A hash table - it includes couples of $\langle$ key, value $\rangle$.
- The key is a number or a string.
- When a unique key is not used in insert, update is executed.
- Keys are indexed: searching is not processed by a sequential search.
- It is not possible to store it in the persistent table.

Using Associative Array

Creating a type:

```
TYPE type_name IS TABLE OF element_type [NOT NULL]
    INDEX BY [PLS_INTEGER | BINARY_INTEGER |
              VARCHAR2(size_limit)];
```

Where:

- `type_name` – a name of the type
- `element_type` – a data type of values of the array
- The key data type is a number (PLS_INTEGER or BINARY_INTEGER) or a string (e.g., VARCHAR)

Example 1/2

```
DECLARE
   -- creating a data type - an associative
   -- array where the key is a string
   -- and the value is a number
   TYPE population_type IS TABLE OF NUMBER INDEX BY VARCHAR2(64);

   -- creating two instances
   country_population population_type;
   continent_population population_type;

   howmany NUMBER;
   which VARCHAR2(64);
BEGIN
   country_population('Greenland') := 100000; -- new item
   country_population('Iceland') := 750000;   -- new item
   -- accessing the value
   howmany := country_population('Greenland');

   continent_population('Australia') := 30000000;
```

Example 2/2

```
    continent_population('Antarctica') := 1000; -- new item
    continent_population('Antarctica') := 1001; -- item update

    -- it returns 'Antarctica'
    which := continent_population.FIRST;

    -- it returns 'Australia'
    which := continent_population.LAST;

    -- it returns a value to last key (the key 'Australia')
    howmany := continent_population(continent_population.LAST);
END;
/
```

Example, Bulk operations 1/2

```
DECLARE
   TYPE UserArray IS VARRAY(10000) OF Usertab%ROWTYPE;
   v_userArray UserArray;
   v_counter NUMBER := 0;
   v_start NUMBER DEFAULT DBMS_UTILITY.GET_TIME;
BEGIN
   v_userArray := UserArray(); -- inicialization
   v_userArray.EXTEND(10000); -- resize

   -- you must run it 10x because 100 000 items
   -- must be inserted
   FOR i IN 1 .. 10
   LOOP
```

Example, Bulk operations 2/2

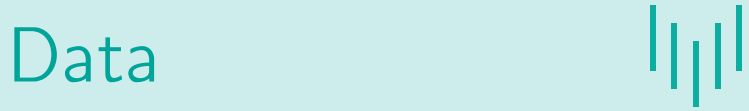
```
    FOR j IN 1 .. 10000          -- prepare the array
    LOOP
      v_counter := v_counter + 1;
      v_userArray(j).id := v_counter;
      v_userArray(j).fname := 'fname' || v_counter;
      v_userArray(j).lname := 'lname' || v_counter;
    END LOOP;

    -- bulk insert
    FORALL i IN v_userArray.FIRST..v_userArray.LAST
      INSERT INTO Usertab VALUES v_userArray(i);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(round((DBMS_UTILITY.GET_TIME-v_start)/
    100, 2) || 's');
END;
```

Collections, Properties, Summary

	Varray	**Nested Table**	**Associative Array**
Index	number	number	number/string
Variable size	No	Yes	Yes
It can be stored in the db. table	Yes	Yes	No
It keeps index of items	Yes	No (after the write into the db. table)	-

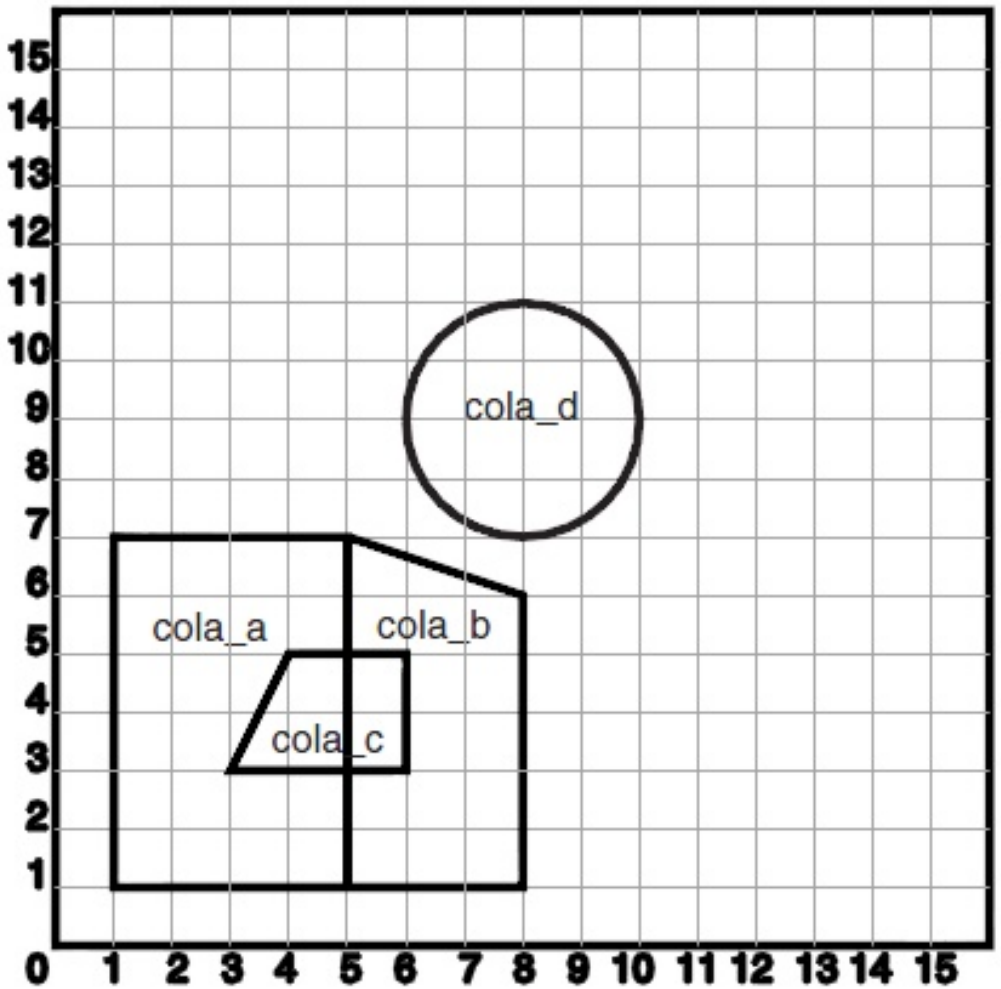
Object-Relational Model and Spatial Data



- Oracle Spatial, often referred to as Spatial, provides an SQL schema and functions that facilitate the storage, retrieval, update, and query of collections of spatial features in an Oracle database.
- Spatial supports the object-relational model for representing geometries.
  - This model stores an entire geometry in the Oracle native spatial data type for vector data, `SDO_GEOMETRY`.
  - An Oracle table can contain one or more `SDO_GEOMETRY` columns.
  - Data are retrieved using SQL with an spatial extension (Open GIS ODBC/SQL specification).
- Spatial Developer's Guide[2]

[2] http://www.oracle.com/pls/db112/portal.all_books

Example, Geometries



Example, Geometry Insertion

```
CREATE TABLE cola_markets (
mkt_id NUMBER PRIMARY KEY,
name VARCHAR2(32),
shape SDO_GEOMETRY);

INSERT INTO cola_markets VALUES(
  1,
  'cola_a',
  SDO_GEOMETRY(
    2003, -- two-dimensional polygon
    NULL,
    NULL,
    -- one rectangle (1003 = exterior)
    SDO_ELEM_INFO_ARRAY(1,1003,3),
    SDO_ORDINATE_ARRAY(1,1, 5,7) -- only 2 points needed to
    -- define rectangle (lower left and upper right) with
    -- Cartesian-coordinate data
));
```

References

- S.W. Dietrich, S.D. Urban: An Advanced Course in Database Systems, Prentice Hall, 2005.
- Oracle Database: Object-Relational Developer's Guide. `https://docs.oracle.com/en/database/oracle/oracle-database/18/books.html`, 2020.
- H. Garcia-Molina, J.D. Ullman, J. Widom: Database Systems, The Complete Book. Prentice Hall, 2002.
- Object Data Management Group (ODMG): The Standard for Storing Objects, (`http://www.odmg.org`), 2006.