

Part 11: Object-Relational Constructs in Oracle

References:

- Jeffrey D. Ullman: Object-Relational Features of Oracle
[<http://infolab.stanford.edu/~ullman/fcdb/oracle/or-objects.html>]
- Jeffrey D. Ullman: CS145 - Introduction to Databases (Autumn 2007)
[<http://infolab.stanford.edu/~ullman/fcdb/aut07/>]
Lecture Notes: Object-Relational SQL
[<http://infolab.stanford.edu/~ullman/fcdb/aut07/slides/or.pdf>]
- Oracle: Database Object-Relational Developer's Guide
[<https://docs.oracle.com/database/121/ADOBJ/adobjint.htm#ADOBJ7025>]
- Oracle: Oracle8 Concepts, 12. Using User-Defined Datatypes
[https://docs.oracle.com/cd/A64702_01/doc/server.805/a58227/ch_objs.htm]
- Oracle: Advanced Topics for Oracle Objects
[https://docs.oracle.com/cd/E11882_01/appdev.112/e11822/adobjadv.htm]
- : Oracle: Using PL/SQL Object Types.
[https://docs.oracle.com/cd/B13789_01/appdev.101/b10807/10_objs.htm]

Objectives

After completing this chapter, you should be able to:

- name some features of object-relational databases,
- define an object-type in Oracle,
-

Overview

1. Introduction

2. Object Types

3. References

4. Subtypes

5. Collection Types

Motivation for ODBMS (1)

- Object-DBMS were introduced around 1985.

There was a lot of research and system development in the 1980s and 1990s. Systems were, e.g., Gemstone, ObjectStore, ORION, O2, ONTOS, Poet. Standards for object-oriented database systems were developed by the ODMG (Object Database Management Group).

- DB applications consist of program code and persistently stored data, and the interface between SQL and a programming language is not smooth.

One of the origins of object-oriented databases were persistent programming languages, and one can view some ODBMS as C++ extended with the possibility to store objects persistently on disk (including objects referenced from these objects via pointers).

Motivation for ODBMS (2)

- Abstract data types and the object-oriented approach have shown that it is advantageous to see data and operations on the data as a unit.

One possibility for enforcing complex integrity constraints is to permit changes to the data only via procedures stored in the database. This helps also for logical data independence (changing the logical schema while still supporting the old interface for legacy applications).

- However, one must separate the basic data storage layer, which is shared between all applications, from the code of a single application.

Physical data independence (decoupling between programs and data storage structures) was one of the achievements of the relational model. Also simplifies the understanding of the data.

Motivation for ODBMS (3)

- The type structure of a classical relational system is very simple: Only relations with columns of pre-defined data types.

And often, lots of columns that are null.

- Types can help to reduce the number of bugs in programs.

They give a better documentation, and thus help in understanding. Furthermore, tools such as the compiler can do more checks.

- As discussed above, subtypes/subclasses are useful in describing the real world, but they are difficult to express in the relational model.

Motivation for ODBMS (4)

- When objects with a complex inner structure must be stored in a relational database, they often have to be split into many tuples, stored in different disk blocks.
- This results in a suboptimal performance.
 - Oracle has “clusters” as a storage structure for relations, which permits to store tuples of different relations in the same disk block.
- Furthermore, one must write code for composing and decomposing an application object.

There are tools and libraries (frameworks) for this.

Why ODBMS failed (1)

- Although OQL (from ODMG) was a good proposal for a declarative query language, it seems that many OODBMS did not support it.

E.g. ONTOS had only a very restricted query language. Most processing of the data had to be done in program code. Even if a system offers a good declarative query language, efficient evaluation needs a lot of work on the query optimizer. It is not easy to catch up with the well established relational systems.

- The visualization of data in tables is very intuitive and compact notation (much data on one screen).

How should a set of linked objects be visualized? Hypertext does not work on printed paper (reports), and even if one uses a web browser, one might get lost in following links and miss something important.

Why ODBMS failed (2)

- Companies had just made investments in relational technology.
- Programmers were trained in relational databases.
- There is a risk in switching to a new technology.

E.g., most ODBMS have vanished and are no longer supported, let alone developed further.

- Some systems were linked to a single language.

E.g., C++. ODMG has defined bindings to different languages. But supporting a new language is more difficult than in the relational case. While an SQL database has a different type system than most programming languages, one would expect from an ODBMS that it supports the object-oriented language of one's choice very well.

Why ODBMS failed (3)

- Object/relational mappers (e.g., Hibernate, LINQ) offer a similar interface to the programmer, but are based on a standard relational database.
- The big relational vendors integrated some object-oriented features into their DBMS:
 - ◇ This gives object-relational database systems.
 - ◇ It permits a gradual change from an existing relational database to trying and using some object-oriented features where they seem useful.

Object-Relational DBs (1)

- Postgres (developed by Michael Stonebreaker and his group at Berkeley) was probably the first object-relational system.

From the [\[VLDB 1987 paper\]](#): “The data model is a relational model that has been extended with abstract data types including user-defined operators and procedures, relation attributes of type procedure, and attribute and procedure inheritance. These mechanism can be used to simulate a wide variety of semantic and object-oriented data modeling constructs including aggregation and generalization, complex objects with shared subobjects, and attributes that reference tuples in other relations.”

Object-relational systems have grown out of extensible database systems and systems that stored nested relations with more general collection types (“structurally object-oriented systems”).

Object-Relational DBs (2)

- The main features of object-relational DBs are:
 - ◇ The possibility to defined own data types and use them for columns of relations.
 - ◇ In particular, one can use also collection types, so that table entries are no longer forced to be atomic.
 - ◇ Inheritance/subclasses
 - ◇ The possibility to define own operations, i.e. program code that is executed in the server, and can be used in SQL statements.

Object-Relational DBs (3)

- Object-relational constructs were introduced in the SQL-99 standard.

There were small extensions in the SQL:2003 standard.

- But systems still differ quite a lot.
- I personally wonder how much object-relational features are really used in practice.

Object-relational constructs are important for non-standard databases, e.g. CAD-databases and GIS-systems. For managing customers and orders, they seem less important.

- OR constructs offer new possibilities for translating from a conceptual model to the DBMS.

Overview

1. Introduction

2. Object Types

3. References

4. Subtypes

5. Collection Types

Object Types (1)

- The following defines an object type “Instructor” with four attributes:

```
CREATE TYPE INSTRUCTOR AS OBJECT(  
    NO            NUMERIC(5),  
    FIRST_NAME   VARCHAR(30),  
    LAST_NAME    VARCHAR(30),  
    EMAIL        VARCHAR(80));  
/  

```

The “/” on a line by itself is necessary to mark the end of the statement. NOT NULL cannot be used. If there is a syntax error, SQL*Plus will only print “Warning: Type created with compilation errors.” Enter “show errors” in this case. Methods are discussed below.

Object Types (2)

- The defined type can be used as a column type like the standard SQL types (in a “relational table”):

```
CREATE TABLE INST_RELTAB(  
    SCHOOL VARCHAR(20),  
    INST INSTRUCTOR);
```

Such objects are called “column objects”.

- It is also possible to create a table that directly stores objects of a given type (an “object table”):

```
CREATE TABLE INST_OBJTAB OF INSTRUCTOR;
```

Similar to a “one-column table”. The objects are called “row objects”. The user-defined type is used here as row type of the table.

Object Types (3)

- When inserting into a table with an object column, one calls a constructor method (which is implicitly defined in this case):

```
INSERT INTO INST_RELTAB  
VALUES('SIS',  
      INSTRUCTOR(1,'Stefan','Brass',NULL));
```

- The query “SELECT * FROM INST_RELTAB” gives

SCHOOL	INST(NO, FIRST_NAME, LAST_NAME, EMAIL)
SIS	INSTRUCTOR(1, 'Stefan', 'Brass', NULL)

Use “column inst format a60” and “column school format a10” to make the display width of the columns smaller (so they fit into one line).

Object Types (4)

- For inserting rows, the object table can be treated like a one-column table:

```
INSERT INTO INST_OBJTAB  
VALUES(INSTRUCTOR(1, 'Stefan', 'Brass', NULL));
```

- But one can also treat it like a table with the attributes as columns:

```
INSERT INTO INST_OBJTAB  
VALUES(2, 'Michael', 'Spring', 'ms@pitt.edu');
```

The constructor is implicitly called in this case.

Object Types (5)

- “SELECT * FROM INST_OBJTAB” shows the attributes as columns:

NO	FIRST_NAME	LAST_NAME	EMAIL
1	Stefan	Brass	
2	Michael	Spring	ms@pitt.edu

- In order to see the objects, write

```
SELECT VALUE(I) FROM INST_OBJTAB I;
```

```
VALUE(I) (NO, FIRST_NAME, LAST_NAME, EMAIL)
```

```
INSTRUCTOR(1, 'Stefan', 'Brass', NULL)
```

```
INSTRUCTOR(2, 'Michael', 'Spring', 'ms@pitt.edu')
```

Object Types (6)

- The object table can be used in queries like a table with the attributes as columns, e.g.

```
SELECT LAST_NAME  
FROM   INST_OBJTAB  
WHERE  NO = 1
```

The data dictionary view COLS lists the attributes of the type like columns of other tables. However, TABS contains entries only for relational tables. USER_ALL_TABLES lists also object tables (it has e.g. columns TABLE_NAME and TABLE_TYPE, which is INSTRUCTOR in this case).

USER_TYPES lists the types owned by the current user.

USER_DEPENDENCIES shows that the table INST_OBJTAB depends on the type INSTRUCTOR.

USER_TYPE_ATTRS lists attributes, USER_TYPE_METHODS lists methods.

Object Types (7)

- If a column of an object type is used, it is mandatory to use an explicit tuple variable (table alias):

```
SELECT INST.LAST_NAME -- Invalid Identifier
FROM   INST_RELTAB
```

- However, this works:

```
SELECT I.INST.LAST_NAME
FROM   INST_RELTAB I
```

The rule is: If an attribute reference contains a dot “.”, it must start with a tuple variable (table alias). Note that the table name is not sufficient, one must explicitly define a tuple variable (e.g. “SELECT INST_RELTAB.INST.NO FROM INST_RELTAB” does not work). For more information, search for “Name Resolution” in the Oracle manuals.

Methods (1)

- As in object-oriented programming languages, an object type can also have methods:

```
CREATE TYPE INSTRUCTOR AS OBJECT(  
    NO          NUMERIC(5),  
    FIRST_NAME VARCHAR(30),  
    LAST_NAME  VARCHAR(30),  
    EMAIL      VARCHAR(80),  
    MAP MEMBER FUNCTION NAME RETURN VARCHAR  
);  
/
```

One of the methods (“member functions”) can be a “map function”. This is special, because the result is used for comparisons.

Methods (2)

- The implementation of the methods (in PL/SQL) must be specified separately:

```
CREATE TYPE BODY INSTRUCTOR AS
    MAP MEMBER FUNCTION NAME RETURN VARCHAR IS
    BEGIN
        RETURN LAST_NAME || ', ' || FIRST_NAME;
    END;
END;
/
```

Note that for the return type of the method, one can only write "VARCHAR" without size.

Methods (3)

- One can call the function in SQL statements, e.g.

```
SELECT I.INST.NAME()  
FROM INST_RELTAB I
```

Note that the explicit tuple variable and the “()” for the empty argument list are required.

- Another example (function call under WHERE):

```
SELECT I.NO  
FROM INST_OBJTAB I  
WHERE I.NAME() = 'Brass, Stefan'
```

Note that this is the object table version, thus I is the object (in contrast to the above query, where I.INST is the object).

Methods (4)

- Because NAME() was declared as the “MAP” member functions, objects are comparable (by mapping them to strings with this function):

```
SELECT I.SCHOOL, J.SCHOOL
FROM   INST_RELTAB I, INST_RELTAB J
WHERE  I.INST = J.INST
```

Objects will be considered identical if they have the same NAME() value — even if they differ in other attributes. It is probably a bad map function, if this can happen.

- One can also use <, > and ORDER BY for a class with map function.

Methods (5)

- It is also possible to define procedures in classes which can have side effects.
- Also static methods are available.
- One can also define own constructors.
- It seems that everything in the `CREATE TYPE` is public, and one cannot define attributes in the private `CREATE TYPE BODY`.

Overview

1. Introduction

2. Object Types

3. References

4. Subtypes

5. Collection Types

References

- A table or attribute can also contain references to objects (pointers):

```
CREATE TABLE COURSES(  
    CRN NUMBER(5),  
    TITLE VARCHAR(20),  
    INST REF INSTRUCTOR);
```

- In SQL, one can use REF(I) to get a reference to object I, e.g.

```
INSERT INTO COURSES  
SELECT 10001, 'Database Management', REF(I)  
FROM INST_OBJTAB I  
WHERE I.NO = 1
```

Overview

1. Introduction
2. Object Types
3. References
4. Subtypes
5. Collection Types

Subtypes (1)

- If one wants to define a subtype of a type, the supertype must be explicitly marked as “NOT FINAL”:

```
CREATE TYPE INSTRUCTOR AS OBJECT(  
    NO            NUMERIC(5),  
    FIRST_NAME   VARCHAR(30),  
    LAST_NAME    VARCHAR(30),  
    EMAIL        VARCHAR(80))  
NOT FINAL;  
/
```

- Now one can define a subtype:

```
CREATE TYPE FACULTY_MEMBER UNDER INSTRUCTOR AS (  
    PROF_TYPE    CHAR(1));  
/
```

Subtypes (2)

- An object of the subtype `FACULTY_MEMBER` can be inserted into a table with the declared type `INSTRUCTOR`:

```
INSERT INTO INST_OBJTAB VALUES (  
    FACULTY_MEMBER(3, 'Paul', 'Munro', NULL, 'A'))
```

- “`SELECT * FROM INST_OBJTAB`” shows only the attributes of the type declared for the table:

NO	FIRST_NAME	LAST_NAME	EMAIL
1	Stefan	Brass	
2	Michael	Spring	ms@pitt.edu
3	Paul	Munro	

Subtypes (3)

- The full objects are shown as follows:

```
SELECT VALUE(I) FROM INST_OBJTAB I;
```

```
VALUE(I) (NO, FIRST_NAME, LAST_NAME, EMAIL)
```

```
INSTRUCTOR(1, 'Stefan', 'Brass', NULL)
```

```
INSTRUCTOR(2, 'Michael', 'Spring', 'ms@pitt.edu')
```

```
FACULTY_MEMBER(3, 'Paul', 'Munro', NULL, 'A')
```

Overview

1. Introduction
2. Object Types
3. References
4. Subtypes
5. Collection Types