Datenbanken II A: DB-Entwurf

Chapter 10: Object-Relational Constructs

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2022/23

http://www.informatik.uni-halle.de/~brass/dd22/

Objectives

After completing this chapter, you should be able to:

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

Contents

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 where developed by the ODMG (Object Database Management Group).
  [https://en.wikipedia.org/wiki/Object_Data_Management_Group]
  [https://cs.ulb.ac.be/public/_media/teaching/odmg.pdf]

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

  One of the origins of object-oriented databases were persistant 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 can be also helpful 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 predefined 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 declaractive 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 diffcullt 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 (addition of new data types) 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.

Contents

Composite Types in PostgreSQL (1)

- The following defines a new type "Instructor" with four attributes/components:

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

- This is conforming to the SQL-2003 syntax.
- `NOT NULL` or any other constraints cannot be used.

  E.g., one cannot specify a `CHECK`-constraint that the value of `NO` must be positive. Constraints can only be specified on the level of tables.

Composite Types in PostgreSQL (2)

- For each `CREATE TABLE` statement, PostgreSQL automatically defines a type with the same name.
- Therefore, in stored procedures, entire rows can be assigned to variables or arguments.

  And a row can also be the result of a function.
- In PostgreSQL, values of such "composite types" can be seen as "objects".
- Tables are understood in PostgreSQL as classes, e.g. they are documented in the data dictionary table `pg_class`.

Composite Types in PostgreSQL (3)

- The defined type can be used as a column type like the standard SQL types:

```
CREATE TABLE INST_COL_TAB(
        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:

```
CREATE TABLE INST_TYPE_TAB OF INSTRUCTOR;
```

This is similar to a “one-column table”. The user-defined type is used here as row type of the table.

Composite Types in PostgreSQL (4)

- It is possible to define CHECK-Constraints that refer to parts of the composed type:

```
CREATE TABLE INST_COL_TAB(
        SCHOOL VARCHAR(20),
        INST INSTRUCTOR,
        CHECK((INST).NO > 0));
```

The parenthesis are necessary in `(INST).NO`. Otherwise PostgreSQL thinks that `INST` is a tuple variable and one gets the error message "`missing FROM-clause entry for table "inst"`". If one tries three parts such as `INST_COL_TAB.INST.NO`, PostgreSQL thinks that the first part is a schema name and complains that the schema does not exist.

- Keys can only be direct columns of the table.

It is possible to declare the entire `INST`-value as a key, but not the `NO` inside the `INST`-value.

Composite Types in PostgreSQL (5)

- The table that directly contains objects of the type is basically a standard table, one can add the usual constraints (in "table constraint syntax"):

```
CREATE TABLE INST_TYPE_TAB OF INSTRUCTOR
        ((CHECK(NO > 0),
         PRIMARY KEY(NO));
```

  One even can add further columns.

- `SELECT * FROM INST_TYPE_TAB` lists the four columns as usual.

- An `INSERT`-Statement requires the four single values for the four attributes as usual, not a single value of type `INSTRUCTOR`.

Composite Types in PostgreSQL (6)

- When inserting into a table with a column of a composed type ("row type"), one must write a corresponding value:

```
INSERT INTO INST_COL_TAB
VALUES('SIS',
       ROW(1,'Stefan','Brass',NULL));
```

Null values are allowed for all columns. It is possible to insert `(NULL,NULL,NULL,NULL)`. One can use `CHECK`-constraints with `IS NOT NULL` in the table declaration.

- The keyword "`ROW`" is not necessary if the composite value has more than one column:

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

Composite Types in PostgreSQL (7)

- One can also write an explicit type conversion:

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

- Or a type conversion with the standard SQL `CAST`:

```
INSERT INTO INST_COL_TAB
VALUES('SIS',
       CAST((1,'Stefan','Brass',NULL)
            AS INSTRUCTOR));
```

Composite Types in PostgreSQL (8)

- PostgreSQL automatically converts String literals with a specific format to the required composite types:

```
INSERT INTO INST_COL_TAB
VALUES('SIS',
       '(1,"Stefan","Brass",NULL)');
```

I.e. string constants for the components of the composite value are written in double quotes inside the main string constant. In this syntax, the keyword `NULL` is not needed, one can leave the field empty (but the comma must be written).

Composite Types in PostgreSQL (9)

- The query "SELECT * FROM INST_COL_TAB" gives

SCHOOL	INST
SIS	(1, 'Stefan', 'Brass', NULL)

- As explained already for CHECK-constraints above, when single components of a composite type are selected, the column of the composite type must be written in (...)

```
SELECT INST.NO -- Error: No tuple variable INST
FROM   INST_COL_TAB
```

PostgreSQL complains about a "missing FROM-clause entry".

- This works:

```
SELECT (INST).NO
FROM   INST_COL_TAB
```

Also `(INST).*` is possible to get all components as result columns.

Methods in PostgreSQL (1)

- PostgreSQL has no methods in the `CREATE TYPE`.
- However, one can declare functions that take a parameter of the composed type:

```
CREATE FUNCTION FULL_NAME(I INSTRUCTOR)
                RETURNS VARCHAR
LANGUAGE SQL
AS $$
    SELECT I.FIRST_NAME || ' ' || I.LAST_NAME;
$$ IMMUTABLE;
```

- This function can be called like an attribute access:

```
SELECT (INST).FULL_NAME
FROM   INST_COL_TAB
```

Methods in PostgreSQL (2)

- Unfortunately, this special syntax does not work with additional arguments.

  Only a function with a single argument can be called in this "dot-notation".
  By the way, `SELECT (1).INC` also works (the parentheses are not only necessary because `1.` is a valid number, e.g. `1 .inc` gives a syntax error).

- Of course, a standard function call is always possible:

```
SELECT FULL_NAME(INST)
FROM   INST_COL_TAB
```

- It is also possible to define type casts for the new types.

  This would permit a different syntax than `(...,...)`.
  [https://www.postgresql.org/docs/current/sql-createcast.html]

Contents

Inheritance in PostgreSQL (1)

- Example for “superclass table” (“parent table”):

```
CREATE TABLE INST(
    NO         NUMERIC(5) NOT NULL,
    FIRST_NAME VARCHAR(30) NOT NULL,
    LAST_NAME  VARCHAR(30) NOT NULL,
    EMAIL      VARCHAR(80),
    PRIMARY KEY(NO));
```

- Example for “subclass table” (“child table”):

```
CREATE TABLE PROF(
    TENURE CHAR(1),
    CHECK(TENURE IN ('Y','N')))
INHERITS (INST);
```

Inheritance in PostgreSQL (2)

- The subclass/child table `PROF` has automatically all columns of the superclass/parent table `INST`.

  Multiple inheritance is possible. Columns with the same name are merged if their data type is equal. Otherwise, one gets an error message.

- Table rows from the subclass/child table `PROF` appear also in the superclass/parent table" `INST`.

  As usual, objects of the subclass are also objects of the superclass.

- Therefore, the following query lists also professors (subclass):

```
SELECT * FROM INST
```

  Of course, only the columns of `INST` are shown.

- The keyword `ONLY` excludes rows from subclasses:

```
SELECT * FROM ONLY INST
```

Inheritance in PostgreSQL (3)

- With the system column "`tableoid`" it is possible to query the real class/table to which a row belongs:

```
SELECT p.*, p.tableoid::regclass
FROM   PROF p
```

  The type conversion to `regclass` shows the name of the table, otherwise `tableoid` is an integer. Alternatively, `tableoid` can be joined with the column `oid` of `pg_class`. The column `relname` of `pg_class` is the table name.

- `CHECK`-constraints and `NOT NULL` constraints are automatically inherited from the "superclass table" to all "subclass tables".

  Unfortunately, this does not hold for keys. See next slide. Foreign keys are not inherited, too, but they can be declared for each subclass/child table.

Inheritance in PostgreSQL (4)

- Indexes belong to a single table only.
- I.e. an index created for the superclass/parent table can enforce a key for all rows that directly belong to that table.
- However, rows from subclass/child tables are not inserted into that index. They can violate the key constraint on the superclass/parent table.
- Of course, one can declare a key also on the subclass/child table, but it enforces uniqueness only for rows that directly belong to that table.
- Thus, there can be a general instructor (object of the superclass) and a professor (object of the subclass) with the same number (similar to Subclass Translation Method 2).

Inheritance in PostgreSQL (5)

- In the same way, foreign keys referencing the superclass/parent table `INST` are a problem:
  - Only direct objects of the superclass can be referenced.

    Not objects of the subclass/child table, that belong indirectly to the superclass/parent table, but are not included in the index for the key of the superclass/parent table.
- The "superclass" table cannot be dropped as long as there are "subclass tables". If necessary, use:

```
DROP TABLE INST CASCADE;
```

This deletes the table including all dependent objects.

Contents

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 fuctions”) 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 the 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`.

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')

Contents

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
```

Contents

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]
- PostgreSQL Documentation: `CREATE TYPE`.
  [https://www.postgresql.org/docs/current/sql-createtype.html]
- PostgreSQL Documentation: Composite Types.
  [https://www.postgresql.org/docs/current/rowtypes.html]
- PostgreSQL Documentation: Inheritance.
  [https://www.postgresql.org/docs/current/ddl-inherit.html]
- 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]