Object-Relational Databases

Elisa Bertino
CS Department and CERIAS
Purdue University

RDBMS

- They manage and manipulate simple data (tabular data)
- They are characterized by a query language (SQL) which is simple, declarative, and standard
- Tools are available supporting application development (Oracle Developer, Sybase Power Builder, Microsoft Visual Basic)

RDBMS: current situation

- Portable on a large number of platforms
- Examples of RDBMS: IBM DB2, Oracle, Sybase, Informix, Microsoft SQL Server
- Very good performance
- Reliability, transaction management
- Based on client-server architecture efficiently support a large number of users
- Provide access control mechanism

RDBMS: current situation

- Today the worldwide market for RDBMS is over 50 billions a year, but... RDBMS have also some limitations
RDBMS: problems

- Mainly related to the intrinsic characteristics of the relational model:
  - SQL-92 provides only a limited set of data types
  - Table have a flat structure and do not provide good support for nested data structures, such as the ones obtained by using the set and array constructors
  - It is not possible to specify sub-typing relations among the objects in a database

- The relationships among entities are modeled by value (through the mechanism of foreign keys) and in the case of complex relationships several additional tables and columns may need to be created
- RDBMS do not exploit current object-oriented software designe methodologies that are today widely used

OODBMS: overview

- They directly model complex objects and their relationships
- The object-orientated paradigm to software engineering and programming: uniqueness of the paradigm
- Good performance for navigational applications
- Limited support for concurrence, parallelism and distribution

OODBMS: problems

- Lack of a data model and query language standard and fully accepted
- Lack of a declarative query language (SQL-like)
- Simple transaction models
- Limited access control capabilities
- Cover a small application market, characterized by applications requiring efficient navigational accesses (like chip design)
RDBMS vs. OODBMS

- RDBMS provide an efficient and effective support for applications manipulating simple data.
- OODBMS provide an efficient support for some application classes that have to manage complex data, but without many of the positive aspects of RDBMS.

The object-relational data model

- The object-relational DBMS have been developed in order to provide all functions and features that RDBMS provide to the management of complex data, typical of OODBMS.

ORDBMS: main features

- **New data types**
  - text, images, audio/video, GIS data, etc.
  - user-defined data types
  - collection types
- **Methods** to model the operations on the user-defined types (ex. Java, C)
- New approaches to represent **relationships**
- The data management approach is however still the relational approach:
  - All data accesses are through SQL.
  - All entities of interest are modeled through table.

ORDBMS: current situation

- Today all main producers DBMS (Oracle, Informix, DB2,..) have extended their DBMS with object-relational features.
- Such extensions require extending the SQL language.
- Currently each RDBMS has a proprietary object-relational extension.
ORDBMS: current situation

- The extensions differ for:
  - The supported functions
  - The approach used to implement such functions
  - The extensions to SQL
- And such differences exist even though there is SQL-99...

The SQL-99 standard

- As part SQL-99 an attempt has been made to standardize the object-relational extension of the relational model
- When SQL-99 was defined the major producers RDBMS already had their versions of object-relational extensions
- SQL-99 does not standardize all object-relational features present in commercial DBMS

The SQL-99 standard

- It is thus a bit early for understanding when and to which extent the standard will be adopted by the commercial products
- It looks like that a further standard will be required to mediate among the various proprietary extensions

In what follows we will...

- Discuss the general features of an ORDBMS
- Discuss how such features are defined by the standard
  - If not otherwise specified, we will use the SQL-99 syntax
- Introduce the object-relational features Oracle
Type system of SQL92

- In SQL-92 the types of relation attributes can be:
  - Numeric (integers, decimals, etc.)
  - String (fixed length or variable length strings, single characters)
  - Temporal (date, time, datetime, interval)
  - Boolean (true, false)
  - Non structured (BYTE, TEXT, BLOB, CLOB)

For each built-in data type there is a fixed and predefined set of operations that can be defined on the values of this data type.

Such limitations often make difficult the representation of real-world information.

Extension to the type system

- Simple type
- Abstract data types
  - User-defined types
- Reference types
- Complex types:
  - record types and collection types
The simple types (also called distinct types) are the simplest form of extension to the set of types provided by an ORDBMS.

They allow the applications to create some new data types, based on a single type (built-in or user-defined).

We refer to the type, on which a distinct type is defined, as base type.

Comparisons with the base type or with other simple types defined on the same base type require the use of cast operations.

The ORDBMS automatically creates a cast function when a new simple type is created.

No inheritance or subtyping mechanisms are provided for the simple types.

The simple types are typically used to define data types that require operations that are different with respect to those available for the types on which the simple types are defined.

The simple types are considered by the DBMS as totally distinct from their base types.

The values of a simple type cannot be directly compared with the values of its base type (strong typing).

Example:

Suppose to create a new data type id_employee based on the integer type.

As the integer type, id_employee is used for recording numeric values but the DBMS will handle these two types as distinct.

For these two types different operations can be defined (for example the sum of two identifiers does not make sense, whereas a comparison operation could be useful).

Simple types in SQL-99:

SQL-99 allows the definition of simple types only on top of built-in types.

```
CREATE TYPE <name> AS <built-in type>
FINAL
```

We will see in what follows the meaning of the FINAL clause.
CREATE TYPE id_employee AS INTEGER FINAL;

CREATE TABLE Employees (id id_employee, name VARCHAR(50), age INTEGER, id_manager id_employee);

SELECT name FROM Employees WHERE id_manager = 123;

CREATE TYPE Euro AS Decimal(8,2) FINAL;
CREATE TYPE USA_Dollar AS Decimal(8,2) FINAL;

CREATE TABLE European_Sales (cust# INTEGER, order# INTEGER, total Euro);
CREATE TABLE USA_Sales (cust# INTEGER, order# INTEGER, total USA_Dollar);

SELECT n_cliente,n_ordine FROM Vendite_Europee ERP, Vendite_USA USA WHERE ERP.n_ordine = USA.n_ordine AND ERP.totale > USA.totale;
Casting in SQL-99

- The DBMS defines two cast functions for each new simple type:
  - One to translate from the distinct type to the built-in type
  - One to translate from the built-in type to the distinct type

Cast functions in SQL-99

CREATE CAST (<source type> AS <target type>)
WITH <function signature>
[AS ASSIGNMENT]

- <source type>: input type
- <target type>: output type
- At least one between <source type> and <target type> must be a type defined by the application
- The other can be any type (built-in or simple type)

Cast functions in SQL-99

- <function signature> is the signature of the function implementing the conversion
- The function must be defined as follows:

  FUNCTION <name> (<source type>) RETURNS <target type>
  ... code ...

Cast functions in SQL-99

- If the AS ASSIGNMENT clause is specified, the casting is implicitly executed when needed
- For each pair of types only a single casting function can be defined by the user
- The cast functions for the simple types are automatically created by the system with AS ASSIGNMENT clause
Casting in SQL-99

- The cast function can be invoked:
  - explicitly
    \[
    \text{CAST(<source type> as <target type>)}
    \]
  - implicitly, without invoking the CAST function

Example

SELECT name
FROM Employees
WHERE id_manager = CAST(123 AS id_employee);

SELECT name
FROM Employees
WHERE id_manager = 123;

Example

SELECT cust#, order#
FROM European_Sales ERP, USA_Sales USA
WHERE ERP.order# = USA.order#
AND CAST(ERP.total AS Decimal(8,2)) >
CAST(USA.total AS Decimal(8,2));

Example - alternative

- To convert from Euro to USA_Dollar one can also defined a new cast function

```sql
CREATE FUNCTION f(e Euro) RETURNS USA Dollar
BEGIN
DECLARE g DECIMAL(8,2);
SET g = e*0.9;
RETURN g;
END;
CREATE CAST(Euro AS Dollaro_USA)
WITH FUNCTION f(Euro);
```
Is this correct?

SELECT cust#, order#
FROM European_Sales ERP, USA_Sales USA
WHERE ERP.order# = USA.order#
AND ERP.total > USA.total;

It is correct only if the cast function is defined with
the AS ASSIGNMENT clause

ADT

- An abstract data type (ADT) includes:
  - One or more attributes
  - One or more methods
- The attributes of an ADT can be specified very much like the table attributes
  - Default values can be specified
  - It is not possible to specify the NOT NULL constraint
- An ADT can be instantiable or not
  - We will see better in what follows

ADT in SQL-99

- If there are only attributes (we will complete the
definition later on):
  CREATE TYPE <type name>
  AS <attribute definition list>
  [{INSTANTIABLE|NOT INSTANTIABLE}]
  {FINAL|NOT FINAL}
- INSTANTIABLE is the default

Example

- Suppose to have to represent the addresses of
  the employees in a RDBMS
- There are two options:
  - address: VARCHAR(n)
  - Representing each component of address as a
    separate attribute
Example

CREATE TYPE address_t AS
  number INTEGER,
  street VARCHAR(50),
  city CHAR(20),
  state CHAR(2),
  zip INTEGER
NOT FINAL;

Address_t is a complex type the attributes of which have predefined types.

ADT

- The ADT can also be nested:

  CREATE TYPE employee_t AS
    emp# id_employee,
    name CHAR(20),
    resume TEXT,
    address address_t
  NOT FINAL;

ADT

- An ADT can be used as:
  - Type of columns in tables (that is, as domains)
  - Type of the tuples in one or more table (row type)

ADT as column types

- The ADT can be used as types of columns

  CREATE TABLE Employees (
    emp# id_employee,
    name CHAR(20),
    resume TEXT,
    address address_t);

  ...
### ADT as column types

Table Employees

<table>
<thead>
<tr>
<th>empid</th>
<th>name</th>
<th>resume</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>number street</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>city state zip</td>
</tr>
</tbody>
</table>

### Methods

- On ADT we can define (signatures of) methods as part of the type definition:

```sql
CREATE TYPE book_t AS
    title CHAR(20),
    sale_price DECIMAL(9,2),
    buying_price DECIMAL(9,2)
NOT FINAL
METHOD revenue() RETURNS DECIMAL(9,2);
```

### Methods

- The methods are functions defined by the users/applications and associated with the types
- They can be written in languages that are proprietary of the DBMS or in programming languages (ex. Java)
- The syntax changes a lot depending on the DBMS used
- The definition is similar to that of functions
  - difference: the methods have an implicit parameter denoting the object on which the method is invoked

### Methods in SQL-99

- They are created with the command CREATE METHOD

  ```sql
  CREATE METHOD <method name>
  (parameter list)
  RETURNS <output data type>
  FOR <UDT name>
  <method body>
  ```
**Example**

```sql
CREATE METHOD revenue() 
RETURNS DECIMAL(9,2) 
FOR book_t 
RETURN (SELF.sale_price - SELF.buying_price);
```

```sql
CREATE FUNCTION review(book_t) 
RETURNS DECIMAL(9,2) 
RETURN (l.sale_price - l.buying_price);
```

**Metodi in SQL-99**

- The methods can be:
  - methods for instances (INSTANCE)
    - Can be invoked on an instance of a type
  - Type methods (STATIC)
    - Can be invoked on a type

- the default is INSTANCE

**Encapsulation**

- The ADT can be encapsulated

- In such case, they can only be manipulated through specific functions created by the DBMS upon their creation
Encapsulation in SQL-99

- Strict encapsulation
- Three predefined methods
  - Constructor methods: to create a new ADT instance
  - Observer methods: to query ADT
  - Mutator methods: to modify the values of ADT instances
- TYPE = CLASS

Constructor methods

- With each ADT a method (constructor) is associated with has the same name of the type
- The constructor creates an instance of the type
- The values to be assigned to the components of the instance to be created are passed as input to the constructor

Constructors in SQL-99

- for any ADT T, a constructor T() exists
  - address_t() \to address_t
  - address_t(1715, 'Ashbury Court', 'Lafayette', 'IN', 47907)

Example - insertion 1

<table>
<thead>
<tr>
<th>Emp#</th>
<th>Name</th>
<th>Resume</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Smith</td>
<td>NULL</td>
<td>14 First Attica IN 47890</td>
</tr>
</tbody>
</table>
Mutator methods

- The are used to modify the instances of an ADT

- It holds also for SQL-99

Example

- Suppose we would like to make the following insertion in the Employees table:

<table>
<thead>
<tr>
<th>emp#</th>
<th>name</th>
<th>resume</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Smith</td>
<td>NULL</td>
<td>14 First Attica IN 47890</td>
</tr>
</tbody>
</table>

- Two steps:
  - The tuple is created by initializing to null the address column
  - Then the components of the address column are updated

Example - insertion 2

```
INSERT INTO Employees
VALUES(123, 'Smith', NULL, address_t());

UPDATE Employees
SET address = address.number(14)
WHERE emp# = 123;

UPDATE Employees
SET address = address.street('First')
WHERE emp# = 123;
```

Example - insertion 3

```
BEGIN
  DECLARE i address_t;
  SET i = address_t();
  SET i = i.number(14);
  SET i = i.street('First');
  SET i = i.city('Attica');
  SET i = i.state('IN');
  SET i = i.zip(47890);
  INSERT INTO Employees VALUES ('123', 'Smith', NULL, i);
END;
```
Observer methods

- For each component of an ADT the system automatically creates an observer method with the same name of the component:
  
  - number() ----> INTEGER
  - street() ----> VARCHAR(50)
  - city() ----> CHAR(20)
  - state() ----> CHAR(2)
  - zip() ----> INTEGER

- Also in SQL-99

Examples of selection with observer methods

SELECT name
FROM Employees
WHERE address.city() = 'Milan'
   OR address.city() = 'Rome';

SELECT address.city()
FROM Employees
WHERE name = 'Smith';

Selection

- The selection of an ADT column returns an instance of the ADT

  SELECT address
  FROM Employees
  WHERE emp# = 123

  The result is
  address_t(14,'First','Attica','IN',47890)

Deletion

DELETE FROM Employees
WHERE address =
  address_t(14,'First','Attica','IN',47890);
Update

UPDATE Employees
SET address = address.number(18)
WHERE emp# = 123;

UPDATE Employees
SET address =
address_t(22,'University','West Lafayette','IN',47906);

Use of methods in queries

CREATE TYPE book_t AS
  title            CHAR(20),
  sale_price       DECIMAL(9,2),
  buying_price     DECIMAL(9,2)
NOT FINAL
METHOD revenue() RETURNS DECIMAL(9,2);
CREATE TABLE bookstore
(codB# INTEGER,
  book book_t);
SELECT b.book.revenue()
FROM bookstore b

Integrity constraints

- It is not possible to define the PRIMARY KEY, UNIQUE, FOREIGN KEY constraints on a column that has as domain an ADT
- Motivation
  - Conceptually everything is OK
  - There are efficiency problems

Operations

- Casting defined by the applications for converting from the ADT to other types
- Possibility of defining ordering and comparison functions
  - We do not see them
Type deletion and update

DROP TYPE <type_name> {CASCADE|RESTRICT};

ALTER TYPE <type_name> <modification_operation>;

< modification_operation >::=
ADD ATTRIBUTE <attribute_name>|
DROP ATTRIBUTE <attribute_name>

Row types

- An ADT can also be used to specify the type of the tuples of a table (row type)
- The tuples of the table are thus instances of the row type, whereas the columns of the table are the same as the attributes of the row type

Row types

- The row types support:
  - The definition of a set of tables that share the same structure (typed tables)
  - A direct and simple modeling mechanism for relationships among different tables (referenceable tables)
  - The definition of hierarchies among tables
- TUPLE OF A TYPED TABLE = OBJECT
- each tuple has an un identifier representing an additional column for each table; the values of such column are unique in the entire database
- by default, the identifiers are generated by the system
  - There are other possibilities, we do not see them

Typed tables in SQL-99

CREATE TABLE <table_name> OF <complex_type_name>
[REF IS <TID_column_name>]

- The REF IS clause denotes the name of an attribute (distinct with respect to the other attribute names) in which the TID (TID - tuple identifier) will be inserted
- The TID column is always the first column in the table schema
- If such clause is missing, the column recording the TID’s exists; it is generated by the system but it transparent to the user (that is, the user cannot select it)
Example

Suppose that we wish to represent information about the projects on which employees work

```
CREATE TYPE project_t AS
  proj# INTEGER,
  name VARCHAR(20),
  description VARCHAR(50),
  budget INTEGER
NOT FINAL;
```

```
CREATE TABLE Projects OF project_t
(REF IS my_TID);
```

```
<table>
<thead>
<tr>
<th>my_TID</th>
<th>proj#</th>
<th>name</th>
<th>description</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>16454</td>
<td>12</td>
<td>Oracle</td>
<td>ORDBMS</td>
<td>10,000,000</td>
</tr>
</tbody>
</table>
```

Typed tables in SQL-99

When defining a typed table it is possible to add integrity constraints on the attributes of the ADT on which the table is defined

```
CREATE TABLE <table_name>
OF <complex_type_name>
[(REF IS <TID_column_name>,
  <constraints>)]
```

Example

```
CREATE TYPE project_t AS
  proj# INTEGER,
  name VARCHAR(20),
  description VARCHAR(50),
  budget INTEGER
NOT FINAL;
```

```
CREATE TABLE projects OF project_t
OK

CREATE TABLE projects OF project_t
NO

CREATE TABLE projects OF project_t
(PRIMARY KEY (proj#));
```

```
CREATE TABLE projects OF project_t
(REF IS my_TID);
```

```
CREATE TABLE projects OF project_t
OK

CREATE TABLE projects OF project_t
NO

CREATE TABLE projects OF project_t
(REF IS my_TID);
```
Row type

- The row types are not encapsulated
- There is encapsulation only when an ADT is used domain of a column
- The attributes of the row type are seen as columns of the table (included the TID column, that can be selected)
- The queries are executed according to the usual approach

Selection

```sql
SELECT proj#
FROM Projects
WHERE budget > 1,000,000;
```

```sql
SELECT my_TID
FROM Projects
WHERE budget > 1,000,000;
```

Insertion

```sql
INSERT INTO Projects(Proj#,Name,Description,Budget)
VALUES(14,'DB development','DB development in Oracle',20000000);
```

no value needs to be specified for the identifier column

Reference types

- The row types can be combined with reference types (REF type)
- The REF types directly support the representation of relationships among type instances
- Such a type allows a column in a tuple in a given table to reference a tuple in another table
- Each tuple in a table is identified through its TID
Example

- Suppose we wish to record information about which employee works in which project
- In a RDBMS one would have two tables Employees and Projects
- The table Employees would include a column indicating the project where each employee works (foreign key)

Example

<table>
<thead>
<tr>
<th>Employees</th>
<th>Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>emp#</td>
<td>proj#</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>123</td>
<td>12</td>
</tr>
</tbody>
</table>

Reference types

- In an ORDBMS there are two additional options:
  - define an ADT project and use this ADT as domain of a column of the Employees relation (however we have data redundancy because the same project may be stored several times)
  - define a table based on a new complex type and refer the tuples of this table from the table Employees
    - Reference type

Reference types in SQL-99

REF (<ADT_type>)
[SCOPE <table_name> [reference_scope_check]]

- The SCOPE specifies a typed table defined on the type <ADT_type>; it specifies that the admitted values for the reference type are references (that is, pointers) to the tuples stored in table <table_name>
- If the SCOPE clause is not specified, the scope is implicit and is represented by all references to tuples instances of the type <ADT_type> (independently from the tables where these tuples are stored)
Reference types in SQL-99

- The SCOPE clause represents is similar to the foreign key constraint in the relational model.
- Also in this context there is the referential integrity problem.
- `<reference_scope_check>` is a clause specifying how integrity has to be maintained, like to what we have seen for the foreign keys.

```
<reference_scope_check> = REFERENCES ARE [NOT] CHECKED [ON DELETE {CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION}]
```

- The meaning is the same as of the relational model.
- Because the TID is considered immutable, there is no ON UPDATE clause.
- default: RESTRICT

Example

```
CREATE TABLE Employees(
  emp# id_employee,
  name VARCHAR(50),
  address address_t,
  assignment REF(project_t) SCOPE Projects REFERENCES ARE CHECKED ON DELETE CASCADE);
```

We associated an employee with a project.
The same project can be associated with several employees.
If a project is deleted, all employees assigned to this project are also deleted.

```
CREATE TABLE Employees(
  emp# id_employee,
  name VARCHAR(50),
  address address_t,
  assignment REF(project_t) SCOPE Projects REFERENCES ARE CHECKED ON DELETE RESTRICT);
```

A project can be deleted if there are no employees assigned to it.
CREATE TYPE employee_t AS
emp# id_employee,
name CHAR(20),
resume TEXT,
address t_indirizzo,
department REF(department_t)
NOT FINAL;

CREATE TABLE Employees OF employee_t (REF IS my_tid);

CREATE TYPE department_t AS
dep# INTEGER,
name VARCHAR(30),
manager REF (employee_t)
NOT FINAL;

CREATE TABLE Departments OF department_t (REF IS my_tid);

Example

- The department column for a tuple of Employees references a tuple of the Departments table (that is, the tuple corresponding to the department where the employee works)
- The manager column for a tuple Departments references a tuple of the Employees table (that is, the tuple corresponding to the employee which is the manager of the department)
Reference types in SQL-99

- Possibility of extending the definition of a typed table with further attributes of REF type

Example

```sql
CREATE TABLE Projects OF project_t
    (proj_ref REF(project_t));
```

<table>
<thead>
<tr>
<th>Projects</th>
<th>proj_ref</th>
<th>name</th>
<th>description</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>Oracle</td>
<td>ORDBMS</td>
<td></td>
<td>10,000,000</td>
</tr>
</tbody>
</table>

Reference types in SQL-99 manipulation

- The values of reference types can only be compared using the = and <> comparison operators
- = denotes the identity equality
- The casting can be defined between a reference type and the target ADT or built-in type

Dereferentiation function DEREF:
- It receives in input an expression denoting a value (pointer) for a reference type with a non empty scope
- It returns the value pointed by the reference (that is, the referenced tuple)

REF function (denoted by →)
- It receives in input an expression that denotes a reference and attribute of the ADT to which the reference points
- It returns the value for that attribute for the referenced tuple
Example

SELECT manager
FROM Departments
WHERE name = "DVD";

it returns a pointer to an employee (that is, the oid of
the employee who is the manager of the DVD
department)

Example

SELECT deref(manager)
FROM Departments
WHERE name = "DVD";

it returns information on the manager of the DVD
department (that is, an entire row of the Employees
table)

Example

SELECT deref(manager).name
FROM Departments
WHERE name = "DVD";

It returns the name of the manager of the DVD
department

Example

SELECT manager -> name
FROM Departments
WHERE name = "DVD";

It returns the name of the manager of the DVD
department
It is equivalent to the previous query
Referential integrity

- The identifiers are generated and assigned by the system
- The user does not them beforehand
- Problem:
  - How to assure referential integrity for a table that includes a reference type?
- Solution:
  - A subquery is used to determine the identifiers to assign to the new tuples

**Example**

```
CREATE TABLE Employees(
  emp# id_employee,
  name VARCHAR(50),
  address address_t,
  assignment REF(project_t) SCOPE Projects REFERENCES ARE CHECKED ON DELETE RESTRICT);
CREATE TABLE Projects OF project_t
  (REF IS My_TID,
   prog_ref REF(project_t));
```

Referential integrity

- When we insert a tuple in the Employees table, we need to specify a project identifier for the attribute assignment; such identifier must reference a tuple of the Projects table
- Two steps:
  - The tuple is inserted by assigning a NULL value to the column of reference type
  - Such column is then modified by the UPDATE command

**Example**

```
INSERT INTO Employees
VALUES (2,'James', address_t( ),NULL);
UPDATE Employees
SET assignment = (SELECT my_tid
                  FROM Projects
                  WHERE name = 'Oracle')
WHERE emp# = 2;
```
Reference types in SQL-99 restrictions

- PRIMARY KEY, UNIQUE, FOREIGN KEY cannot be specified

Collection types and row types

Collection types

- The collection types can be seen as “containers” for objects with a similar structure
- There is no set of collection types that is supported by all ORDBMS
  - set
  - bag
  - list
  - array

Collection types in SQL-99

- The only collection type included in SQL-99 is ARRAY
  - `<attribute_name> <type> ARRAY<dimension>`
- `<dimension>` is an integer value
- Constructor:
  - `ARRAY<value_1>,...,<value_n>`
- access:
  - `<attribute_name>[i]` where i is an integer value between 1 and n
Collection types in SQL-99

- The number of elements in an array can be any number between 0 (ARRAY[]) and the maximum number of elements specified in the array declaration.
- Implicitly, there exists a “length” parameter directly managed by the system.

Example

```sql
CREATE TABLE Employees(
    emp# id_employee,
    name VARCHAR(50),
    skills VARCHAR(20) ARRAY[3];
)

INSERT INTO Employees
VALUES (2,'James',ARRAY['Oracle','Unix','Java']);

SELECT *
FROM Employees
WHERE skills[2] = 'Unix';
```

Example

```sql
CREATE TYPE employee_t AS
    emp# id_employee,
    name VARCHAR(30),
    address address_t,
    skills VARCHAR(20) ARRAY[3],
    manager REF(employee_t),
    projects REF(project_t) ARRAY[10],
    children REF(person_t) ARRAY[10],
    hobbies VARCHAR(20) ARRAY[5],
    NOT FINAL;

CREATE TABLE Employees OF employee_t;
```
Example

UPDATE Employees
SET skills = ARRAY[‘Oracle’,’Unix’];

UPDATE Employees
SET skills = ARRAY[‘SQL Server’];

The new array contains a single element (the length changes)

Collection type in SQL-99 manipulation

- Casting
  - cast on the type of the array elements and reduction in the number of elements
- assignment:
  - usual
  - If the value is truncated an error is generated
- comparison:
  - =, <>

Collection type in SQL-99 functions

- concatenation
  - CONCATENATE (<array_expression> WITH <array_expression>)
- cardinality
  - CARDINALITY(<array_expression>)

Collection type in SQL-99 restrictions

- For the fields defined as array the UNIQUE, PRIMARY KEY, FOREIGN KEY constraints cannot be specified
Tuple types in SQL-99

- SQL-99 provides the possibility of a nested definition for the row types (we refer to these types of values as tuple types)
- They do not require the definition of an ADT but their definition can be directly embedded in the definition of a table or another type

Row types in SQL-99

- Type
  \[ \text{ROW} (<\text{def component}_1>, \ldots, <\text{def component}_n>) \]
- example:
  \[ \text{ROW}(\text{number INTEGER, street VARCHAR(50), city CHAR(20), state CHAR(2), zip INTEGER}) \]

Example

```sql
CREATE TABLE Employees ( 
  emp# id_employee, 
  name CHAR(20), 
  resume TEXT, 
  address ROW( number INTEGER, street VARCHAR(50), city CHAR(20), state CHAR(2), zip INTEGER) );
```

Tuple types in SQL-99

- Values:
  - ROW(<value typo_1>,...,<value typo_n>)
- Example:
  - ROW(3, 'University', 'West Lafayette', 'IN', 47906)
- Also the tuples returned by a query are seen as value of the tuple type
- The components of a tuple can be accessed by using the dot notation
Example

INSERT INTO Employees
VALUES (3,'James',NULL,
ROW(3,'University','West Lafayette','IN',47906))

Example

CREATE TABLE Addresses
(
  lid INTEGER,
  street VARCHAR(20),
  city VARCHAR(20),
  state VARCHAR(20),
  zip INTEGER);

UPDATE Employees
SET Address = (SELECT t FROM Addresses t WHERE lid = 3)
WHERE name = 'James';

Example

SELECT Name
FROM Employees
WHERE address.city = 'West Lafayette';

Tuple types in SQL-99 manipulation

- Assignment:
  - The same number of components
  - The types must be compatible
- Comparison:
  - =, <>, <, <=, >, >=
  - Lexicographic ordering, based on the types of components
  - The values must have the same number of components
  - The presence of NULL can generate UNKNOWN
## Example

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW(1,1,1) = ROW(1,1,1)</td>
<td>TRUE</td>
</tr>
<tr>
<td>ROW(1,1,1) = ROW(1,2,1)</td>
<td>FALSE</td>
</tr>
<tr>
<td>ROW(1,NULL,1) = ROW(2,2,1)</td>
<td>FALSE</td>
</tr>
<tr>
<td>ROW(1,NULL,1) = ROW(1,2,1)</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>ROW(1,1,1) &lt;&gt; ROW(1,2,1)</td>
<td>TRUE</td>
</tr>
<tr>
<td>ROW(2,NULL,2) &lt;&gt; ROW(2,2,1)</td>
<td>TRUE</td>
</tr>
<tr>
<td>ROW(2,2,1) &lt;&gt; ROW(2,2,1)</td>
<td>FALSE</td>
</tr>
<tr>
<td>ROW(1,NULL,1) &lt;&gt; ROW(1,2,1)</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>ROW(1,1,1) &lt; ROW(1,2,0)</td>
<td>TRUE</td>
</tr>
<tr>
<td>ROW(1,NULL,1) &lt; ROW(2,NULL,0)</td>
<td>TRUE</td>
</tr>
<tr>
<td>ROW(1,1,1) &lt; ROW(1,1,1)</td>
<td>FALSE</td>
</tr>
<tr>
<td>ROW(3,NULL,1) &lt; ROW(2,NULL,0)</td>
<td>FALSE</td>
</tr>
<tr>
<td>ROW(1,NULL,1) &lt; ROW(1,2,0)</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>ROW(NULL,1,1) &lt; ROW(2,1,0)</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

## Tuple types in SQL-99

- The tuple types cannot have PRIMARY KEY, UNIQUE, FOREIGN KEY constraints.

## Inheritance

- The inheritance mechanism allows one to define supertype/subtype relationships.
- The inheritance mechanism supports the specialization of existing types based on the application requirements.
Inheritance

- A subtype inherits all attributes, methods, and constraints defined for its supertypes
- A subtype can extend the supertype by adding new attributes and methods
- A subtype can also re-define the inherited methods and attributes

Type inheritance

Consider the following entities:

**Truck:**
- model: CHAR(20),
- plate#: INTEGER,
- last_revision: DATE,
- weight: INTEGER,
- next_revision(): DATE

**Bus:**
- model: CHAR(20),
- plate#: INTEGER,
- last_revision: DATE,
- seats#: INTEGER,
- next_revision(): DATE

We can distinguish two forms of inheritance:
- Type inheritance
- Table inheritance

In the relational model to represent those entities we need two table and two procedures

In a ORDBMS, truck and bus can be considered specializations of a common type of entity: Vehicle

One can thus define the a type vehicle including all features that are common to the types truck and bus

Truck and bus are defined as subtype of vehicle; each one adds some special features
Type inheritance in SQL-99

- For ADT
- Single inheritance
  
  ```sql
  CREATE TYPE <type_name>
  UNDER <supertype_name>
  AS ... additional attributes
  [NOT] FINAL;
  ```

  - The supertype must have been declared with the NOT FINAL clause

- FINAL clause:
  - If a type has been defined as FINAL, no subtypes can be generated from it

- NOT FINAL clause:
  - Subtypes can be defined
  - The NOT FINAL clause is mandatory if no supertype is specified in the definition
  - Otherwise one can choose

Example

```sql
CREATE TYPE vehicle AS
  model CHAR(20),
  plate# INTEGER,
  last_revision DATE,
  METHOD next_revision() RETURNS DATE
NOT FINAL;
```

```sql
CREATE TYPE truck_t
  UNDER vehicle_t
  AS
  weight INTEGER
  NOT FINAL;
```

```sql
CREATE TYPE bus_t
  UNDER vehicle_t
  AS
  seats# INTEGER
  NOT FINAL;
```
CREATE TYPE t_persona AS
nome CHAR(20),
id INTEGER,
data_di_nascita DATE,
indirizzo t_indirizzo,
METHOD età() RETURNS INTEGER
NOT FINAL;

CREATE TYPE t_insegnante
UNDER t_persona AS
stipendio DECIMAL(9,2),
data_assunzione DATE,
corso t_corso
NOT FINAL;

CREATE TABLE Insegnanti OF t_insegnante;
SELECT nome, I.età()
FROM Insegnanti I
WHERE stipendio > 3000;

It is possible to redefine an inherited method
It is not possible to redefine the attributes
Consider a type teacher_t; suppose that to this type a
method is associated called age that determines the
years of service as teacher (overriding)
CREATE TYPE teacher_t
UNDER person_t AS
salary DECIMAL(8,2),
hiringdate DATE,
course t_course
OVERRIDING METHOD age() RETURNS INTEGER
NOT FINAL;

The declaration of a type specifies whether the
type can be instantiated (as thus have its own
instances) or not
CREATE TYPE <type_name>
AS <attribute_definition_list>
[[INSTANTIABLE|NOT INSTANTIABLE]]
(FINAL|NOT FINAL)
The default is INSTANTIABLE
A type that cannot be instantiated corresponds to an
abstract class: it only useful for code reuse
Substitutability

- In the OODBMS the substitutability principle holds
- An instance of a type can be used everywhere we can use an instance of an instance of one of its supertypes
- Such principle does not hold for current ORDBMS
- To support substitutability:
  - CAST functions must be used

Table inheritance

- The typed tables can be organized according to inheritance hierarchies
- However this is possible only if the types on which the tables are defined are related by the inheritance relation
- The inheritance among tables is useful in order to extend SQL operations (such as queries) to the instances of a table and to the instances of all the sub-tables of this table

Example

CREATE TABLE People OF person_t;
CREATE TABLE Teachers of teacher_t UNDER people;

- An hierarchy has been created between the People and the Teachers tables

Queries

- The inheritance relationships holding among tables influences query results
- A query issued on a table automatically propagates to the sub-tables
- The same applies to the delete and update operations, whereas an insert operation only applies to a single table
- If the operation has to be restricted to the instances of a given table, the ONLY option has to be specified
Example

- **People**
  - name | id  | birth_date   | address
  - Smith | 74  | 16/8/68      | 
  - John  | 86  | 3/2/48       | 

- **Teachers**
  - name | id  | birth_date | address | salary
  - Allen | 82  | 9/7/67     |         | 30K
  - Mark  | 81  | 3/5/58     |         | 60K

Example

- SELECT name FROM People WHERE birth_date > 1/1/1967;
  - The result is: Smith and Allen

- SELECT name FROM ONLY People WHERE birth_date > 1/1/1967;
  - The result is: Smith

Example

- DELETE FROM People WHERE id > 80;
  - It deletes John from the People table and Allen and Mark from the Teachers table

Object-relational Features of Oracle 9i
The Oracle type system

- No distinct type
- Object types (corresponding to ADT)
- Reference types
- Collection types
- Inheritance

Object types

- Definition of ADT:
  - Specification:
    - Attribute declaration
    - Method specification
  - Body:
    - Code of methods

Example

CREATE TYPE Complex AS OBJECT
  r_part FLOAT,
  i_part FLOAT,
MEMBER FUNCTION sum(x Complex) RETURNS Complex,
MEMBER FUNCTION minus(x Complex) RETURNS Complex,
MEMBER FUNCTION product(x Complex) RETURNS Complex,
MEMBER FUNCTION division(x Complex) RETURNS Complex;

Example

CREATE TYPE BODY Complex AS
  MEMBER FUNCTION sum(x Complex)
    RETURN Complex IS
      BEGIN
      RETURN Complex(r_part + x. r_part, i_part + x. i_part);
      END sum;
  MEMBER FUNCTION difference(x Complex)
    RETURN Complex IS
      BEGIN
      RETURN Complex(r_part - x. r_part, i_part - x. i_part);
      END difference;
      END;
Object types

- Everything we have said for SQL-99 applies here with the following differences:
  - The notion of body
  - There is no encapsulation: the attributes can be directly accessed through the dot notation

Methods

- They can be functions or procedures
- Two types:
  - MEMBER
    - Defined for instances
    - Implicit parameter: SELF
  - STATIC
    - Defined for the type

Example

```sql
CREATE TYPE Rational AS OBJECT
(num INTEGER,
den INTEGER,
MEMBER PROCEDURE normalize,
...);
```

```sql
CREATE TYPE BODY Rational AS
MEMBER PROCEDURE normalize IS
  g INTEGER;
BEGIN
  g := gcd(SELF.num, SELF.den);
  g := gcd(num, den); -- equivalent to previous line
  num := num / g;
  den := den / g;
  END normalize;
... END;
```

Special methods

- Constructors
  - as in SQL-99
- MAP methods
- ORDER methods
MAP methods

- They allow one to compare ADT instances by mapping
  them to values of built-in types (DATE, NUMBER,
  VARCHAR)
- They thus represent a casting between an ADT and
  one of the built-in types
- If for a given ADT, a MAP method exists, the
  comparisons among instances of this ADT are
  executed by first converting the instances into the
  values of the considered built-in type

Example

```sql
CREATE TYPE Rectangle_typ AS OBJECT (  
  length NUMBER,  
  width NUMBER,  
  MAP MEMBER FUNCTION area RETURN NUMBER,  
  ...  
);
CREATE TYPE BODY Rectangle_typ AS
  MAP MEMBER FUNCTION area RETURN NUMBER IS
    BEGIN
      RETURN length * width;
    END area;
  ...  
END;
```

ORDER methods

- These methods implement an order relation on the set
  of instances of a given type
- They have always a parameter with the same type
  of the type for which the method is defined
- They are useful for comparing complex data types that
  cannot be easily compared with a MAP method
- If an ORDER method exists, the method is
  automatically invoked when the instances of the
  considered type are compared

Example

- Let o1 and o2 be instances rectangle_typ type:
  - o1 < o2 is equivalent to o1.area() < o2.area()
  - The comparison relation is established on two
    instances of the NUMBER type
ORDER methods

- The output of such a method is always an integer which can have one of the following values:
  - -1: SELF < parameter
  - 0: SELF = parameter
  - +1: SELF > parameter
- For an ADT, at most one MAP or ORDER method can be defined
- If none of the two is defined, the system supports only = and <>

Example

```sql
CREATE TYPE Customer_t AS OBJECT (
  id NUMBER,
  name VARCHAR2(20),
  address VARCHAR2(30),
ORDER MEMBER FUNCTION match (c Customer_t) RETURN INTEGER IS
BEGIN
  IF id < c.id THEN
    RETURN -1;
  ELSIF id > c.id THEN
    RETURN 1;
  ELSE
    RETURN 0;
  END IF;
END;
END;
)
```

Customers are compared with respect to their identifiers

Typed tables

- Also in Oracle an ADT type can be used according to two different modes:
  - as type (i.e. domain) for a table attribute
  - as base type for the definition of a typed table
- It is not possible to specify a column for the identifier (no REF IS clause)
  - The Identifier is not managed as an attribute but as a function
Object types: access

- Access through notation to attributes and methods
- If the dot notation is used, it is always necessary to use a relation alias for the accessed table

Example

CREATE TYPE Person_t AS OBJECT (ssn# VARCHAR(20));
CREATE TABLE ptab1 OF Person_t;
CREATE TABLE ptab2 (c1 Person_t);

SELECT ssn# FROM ptab1; OK
SELECT c1.ssn# FROM ptab2; Error
SELECT ptab2.c1.ssn# FROM ptab2; Error
SELECT p.c1.ssn# FROM ptab2 p; OK

Syntax of commands

- CREATE TYPE typename AS OBJECT (attrname datatype [, attrname datatype]);
- CREATE OR REPLACE TYPE BODY typename AS metodo {metodo};
- CREATE TABLE tablename OF typename [attrname NOT NULL] [,attrname NOT NULL] [,PRIMARY KEY (attrname [,attrname ])];
- DROP TYPE typename;
- DROP TABLE tablename;
- ALTER TYPE typename REPLACE AS OBJECT (nuova definizione tipo);
- CREATE OR REPLACE TYPE BODY typename IS metodo {metodo};

Reference types

- Vale quanto visto per SQL-99
  - cambia un minimo la sintassi
  - REF <nome tipo> SCOPE IS <nome_tavella>
- Nessuna possibilita’ di specificare come gestire l’integrita’
- Predicato IS DANGLING per stabilire se un valore di tipo REF punta ad una tupla non piu’ disponibile
Reference types - manipulation

- Three main functions:
  - referencing `ref()`: given an object of a given type, it returns the identifier of this object
  - dereferencing `deref()`: given an identifier, it returns the object
  - `value()`: given a relation alias, it returns the tuple object associated (by using the proper constructor)

Reference types

CREATE TYPE person_t AS OBJECT(
  fname VARCHAR2(10),
  lname VARCHAR2(15),
  birth_date DATE,
  address address_t,
  mother REF person_t,
  father REF person_t);

CREATE TABLE People OF person_t;

Reference types

INSERT INTO People
VALUES('Thomas','Smith',...,NULL,NULL);

INSERT INTO People
VALUES(person_t('Joanna','White',...,NULL,NULL));

INSERT INTO People
VALUES('Robert','Smith',...,NULL,NULL);
People

<table>
<thead>
<tr>
<th>fname</th>
<th>lname</th>
<th>...</th>
<th>mother</th>
<th>father</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thomas</td>
<td>Smith</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>Joanna</td>
<td>White</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>Robert</td>
<td>Smith</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

The mother of Robert Smith is Joanna White

UPDATE People p
SET p.mother = (SELECT ref(d1) FROM People d1
WHERE fname = 'Joanna'
and lname = 'White')
WHERE fname = 'Robert' and lname = 'Smith';

We obtain:

person_t('Thomas', 'Smith', NULL, NULL)
person_t('Joanna', 'White', NULL, NULL)
person_t('Robert', 'Smith', xxxyyywww, NULL)

where xxxyyywww is the identifier of the tuple of Joanna White
Reference types

SELECT deref(p.mother)
FROM People p
WHERE fname = 'Robert' and lname='Smith';

It returns all information contained in the People table related to the mother of Robert Smith

It returns

person_t('Joanna','White',NULL,NULL)

Reference types

DELETE FROM People
WHERE fname = 'Joanna' AND lname = 'Smith';

After such operation the tuple corresponding to Robert Smith has a dangling pointer; there is a special predicate to retrieve the tuples with dangling pointers

SELECT fname, lname
FROM People
WHERE mother IS DANGLING;

It returns Robert Smith

Collection types

- Oracle provides two collection types:
  - nested table
  - varray
- The collection types may have elements instances of object types
- Also an object type can have an attribute the type of which is a collection type

Collection types

- The nested tables can be considered as a table with a single column
- Main differences between arrays and nested tables:
  - The arrays have a fixed dimension whereas the nested tables have a variable dimension
  - The arrays are stored inside the table in which are used or as BLOB (the storage strategy is decided by the system)
  - The nested tables are stored in separated tables, with an additional column that identifies the tuple of the table to which they belong
Collection types - varray

<table>
<thead>
<tr>
<th>Data1</th>
<th>Data2</th>
<th>Data3</th>
<th>Data4</th>
<th>VARRAY_Data[5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>[A11, A12, A13]</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>[B21, B22, B23, B24, B25]</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>[C31, C32]</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>[D41]</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>[E51, E52, E53, E54]</td>
</tr>
</tbody>
</table>

Collection types – nested tables

Collection types

- They cannot be directly used in the definition of an attribute
- It is always necessary to assign a name to the collection type before using it
- For each collection type, a constructor exists
- To create an instance, it is necessary to pass to the constructor a set of elements of the type on which the collection type is based

Varrays - creation

```sql
CREATE TYPE project_t AS OBJECT(
    id INTEGER,
    title VARCHAR2(25),
    cost NUMBER(7,2));
CREATE TYPE project_list AS VARRAY(50) OF project_t;
CREATE TYPE department_t AS OBJECT(
    dept# INTEGER,
    name VARCHAR2(15),
    budget NUMBER(11,2),
    projects project_list);
CREATE TABLE Departments OF department_t;
```
**Varray - insertion**

```sql
INSERT INTO Departments
VALUES(30,'R&D',1000000000,
       project_list(project_t(1,'DBMS',10000000),
                   project_t(3,'C++',20000000)));
```

```sql
INSERT INTO Departments
VALUES(32,'Marketing',1000000000,
       project_list (project_t(1,'New Ads',10000000),
                     project_t(3,'Personnel Bonus',20000000)));
```

**Nested tables - creation**

```sql
CREATE TYPE course_t as object(
    id number(4),
    name varchar2(25),
    credits number(1))
```

```sql
CREATE TYPE course_list as TABLE course_t;
```

**Nested tables - creation**

A) CREATE TABLE University_departments (  
   name VARCHAR2(20),
   chair VARCHAR2(20),
   courses course_list  

   NESTED TABLE courses STORE AS course_tab;

B) CREATE TYPE university_department_t AS OBJECT(  
   name VARCHAR2(20),
   chair VARCHAR2(20),
   course course_list  

   CREATE TABLE University_departments OF university_department_t;

   NESTED TABLE courses STORE AS course_tab;
```

**Nested Table - creation**

- Notice the NESTED TABLE clause in the previous slide
- such clause is necessary because the columns defined as nested table are stored as separated tables
- The general format of the clause is
  NESTED TABLE colname STORE AS tablename
- The table tablename is called “child-table” of the table within which is defined (called “parent-table”)
- A child-table can only be accessed through the parent-table
- The names of the child-tables must be unique in the schema
- During the manipulation and query operations, the system executes a join between the parent and child tables
  - The nested tables are less efficient than the varray’s but more flexible
Nested tables: insertion

```
INSERT INTO University_departments
VALUES('Computer Science', 'Brown',
    course_list(courset(1000, 'Programming I', 2),
    courset(1001, 'Mathematical Logics', 1),
    courset(1002, 'Database Systems', 2),
    courset(1003, 'Computer Graphics', 1)));
```

Collection types - queries

- Two possibilities
  - Select the nested collection
    - a tuple for each collection
  - Select the unnested collection
    - a tuple for each element of the collection
    - TABLE function
- They apply to both VARRAY and NESTED TABLE

Nested selection - example

```
SELECT courses
FROM University_departments;
```

The result is

```
course_list(courset(1000, 'Programming I', 2),
    courset(1001, 'Mathematical Logics', 1),
    courset(1002, 'Database Systems', 2),
    courset(1003, 'Computer Graphics', 1))
```

Nested selection - example

```
SELECT t.*
FROM University_departments d, TABLE(d.courses) t;
```

The results is given by a table containing tuples of type `course_t`

```
<table>
<thead>
<tr>
<th>course_id</th>
<th>name</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Programming I</td>
<td>2</td>
</tr>
<tr>
<td>1001</td>
<td>Mathematical Logics</td>
<td>1</td>
</tr>
<tr>
<td>1002</td>
<td>Database Systems</td>
<td>2</td>
</tr>
<tr>
<td>1003</td>
<td>Computer Graphics</td>
<td>1</td>
</tr>
</tbody>
</table>
**TABLE function**

- The TABLE function receives a value of type collection and allows one to use it as a table.
- The previous query performs a join for each tuple of the University_departments table with each element of the collection object.
- The function can also be applied to queries returning a single value of type collection.

**Example**

```sql
SELECT d.name, t.*
FROM University_departments d, TABLE(d.courses) t;
```

The result is:

<table>
<thead>
<tr>
<th>Department</th>
<th>Course</th>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Science</td>
<td>Programming I</td>
<td>2</td>
</tr>
<tr>
<td>Computer Science</td>
<td>Mathematical Logics</td>
<td>1</td>
</tr>
<tr>
<td>Computer Science</td>
<td>Database Systems</td>
<td>2</td>
</tr>
<tr>
<td>Computer Science</td>
<td>Computer Graphics</td>
<td>1</td>
</tr>
</tbody>
</table>

**Example**

```sql
SELECT * FROM TABLE(SELECT courses FROM University_departments
WHERE name = 'Psychology');
```

returns all information about the courses of the Psychology department.

```sql
SELECT credits FROM TABLE(SELECT courses FROM University_departments
WHERE name = 'Psychology');
```

returns the credits of all the courses of the Psychology department.

The queries are correct if the query on which the TABLE function is applied returns a single element.

**Collection types - DML**

- Nested tables:
  - The TABLE function can be used to modify a collection.
- VARRAY's:
  - Unlike the nested tables, the elements of a varray cannot be manipulated through SQL instructions.
  - To select or update a given element of a VARRAY, PL/SQL must be used.
  - There is a large number of functions for manipulating arrays.
Nested tables

```sql
UPDATE
  TABLE(SELECT courses FROM University_departments
       WHERE name = 'Psychology')
SET credits = credits + 1
WHERE id IN (2200,3540);

DELETE FROM TABLE(SELECT courses FROM University_departments
                  WHERE name = 'English') p
WHERE p.credits = 2;
```

Varray

```sql
DECLARE my_projects project_list;

SELECT projects INTO my_projects
  FROM Departments.
  WHERE id = 30;

IF my_projects(i).Title = 'DBMS' …
```

Inheritance

- Only for types
- It applies what we said for SQL-99
- In addition:
  - Both types and methods can be defined as FINAL/NOT FINAL
    - FINAL: overriding is not possible
  - Substitutability
  - Late binding

Substitutability

- It refers to the possibility of using an instance of a type in each context in which an instance of a supertype of this type can be used
- Substitutability at
  - Attribute level
  - Tuple level
Substitutability at attribute level

- An attribute of type REF(<type1>) can be assigned values of type REF(<type2>) if <type2> is a subtype of <type1>
- An attribute of type ADT <type1> can be assigned values of type ADT <type2> if <type2> is a subtype of <type1>
- An attribute of type collection defined on type <type1> can include elements of type <type2> if <type2> is a subtype of <type1>

Substitutability at tuple level

- A typed table defined on the type < type 1> can contain instances of the type < type 2> if <type2> is a subtype of <type1>

Example

```sql
CREATE TYPE Person_t AS OBJECT
  (ssn NUMBER,
   name VARCHAR2(30),
   address VARCHAR2(100)) NOT FINAL;
CREATE TYPE Student_t UNDER Person_t
  (deptid NUMBER,
   major VARCHAR2(30)) NOT FINAL;
CREATE TYPE PartTimeStudent_t UNDER Student_t
  (numhours NUMBER);
```

Example (on attributes)

```sql
CREATE TYPE Person_t AS OBJECT
  (ssn NUMBER,
   name VARCHAR2(30),
   address VARCHAR2(100)) NOT FINAL;
CREATE TYPE Student_t UNDER Person_t
  (deptid NUMBER,
   major VARCHAR2(30)) NOT FINAL;
CREATE TYPE PartTimeStudent_t UNDER Student_t
  (numhours NUMBER);
CREATE TABLE Departments
  (id INTEGER,
   name VARCHAR(20),
   manager Person_t);
INSERT INTO Departments
  VALUES (1,'research', Person_t(1243, 'Bob', '121 Front St'));
INSERT INTO Departments
  VALUES (2,'development',Student_t(3456, 'Joe', '34 View', 12, 'HISTORY'));
INSERT INTO Departments
  VALUES (3, 'testing', PartTimeStudent_t(5678, 'Tim', '14 Back St', 13, 'PHYSICS', 20));
```
Example (on tuples)

CREATE TABLE People OF Person_t;
INSERT INTO People
VALUES (Person_t(1243, 'Bob', '121 Front St'));
INSERT INTO People
VALUES (Student_t(3456, 'Joe', '34 View', 12, 'HISTORY'));
INSERT INTO People
VALUES (PartTimeStudent_typ(5678, 'Tim', '14 Back St', 13, 'PHYSICS', 20));

Limiting substitutability

CREATE TABLE Departments
( id    INTEGER,
  name VARCHAR(20),
  manager Person_t)
COLUMN manager NOT SUBSTITUTABLE AT ALL LEVELS;
Manager can only be an object of type Person_t (instances of subtypes not admitted)
CREATE TABLE People OF Person_t NOT SUBSTITUTABLE AT ALL LEVELS;
The table People can only include instances of the type Person_t (and not of its subtypes)

Limiting substitutability

CREATE TABLE Departments
( id    INTEGER,
  name VARCHAR(20),
  manager Person_t)
COLUMN manager IS OF (ONLY Student_t);
Only one subtype can be specified

Queries

- Useful functions:
  - REF
  - DEREF
  - VALUE
  - Expr IS OF type: it determines if the type of Expr is 'type'
  - TREAT(Expr AS type): if modifies the type of Expr into 'type'; 'type' must be subtype of the type of Expr
  - ...
- REF, DEREF, VALUE già viste
Example – IS OF

- Si consideri la tabella Dipartimenti presentata in precedenza (senza limitazioni di sostituibilità)

```sql
SELECT * 
FROM Dipartimenti p
WHERE p.manager IS OF (Student_typ, PartTimeStudent_typ)
```

restituisce i dipartimenti i cui manager sono studenti o studenti part-time

---

Example – IS OF

- Consider the People table (without the substitutability limitations)

```sql
SELECT * 
FROM People p
WHERE VALUE(p) IS OF (Student_t, PartTimeStudent_t)
```

It returns the people that are students or part-time students

---

Example - TREAT

- È utile per accedere ai campi di uno subtype del tipo di una tabella o un attributo

```
CREATE TABLE People OF Person_t;

INSERT INTO People
VALUES (Person_t(1243, 'B b', '121 F r', St));

INSERT INTO People
VALUES (Student_t(3456, 'Joe', '34 View', 12, 'HISTORY'));

INSERT INTO People
VALUES (PartTimeStudent_t(5678, 'Tim', 13, ****, 'PHYSICS', 20));

SELECT TREAT(value(p) as PartTimeStudent_t).numhours 
FROM People p;
```

SELECT p.numhours ERROR 
FROM People p 
WHERE value(p) IS OF PartTimeStudent_t;

---

Main differences with respect to SQL-99

- Collection types
  - SQL-99: only ARRAY
  - Oracle: VARRAY e NESTED TABLE
- inheritance
  - SQL-99: on types and tables, no substitutability
  - Oracle: on only types, substitutability
Use of ORDB functions from JDBC

- We consider JDBC 3 (but several extensions are already present in DBC 2)
- New interfaces to map the object relational types into Java types

Creation of new types

- Because it is a DDL command DDL, the `executeUpdate` method is used

```java
String type = "CREATE TYPE address AS
    number INTEGER,
    street VARCHAR(50),
    city CHAR(20),
    state CHAR(2),
    zip INTEGER";

Statement st = con.createStatement();
st.executeUpdate(type);
```

Manipulation of new type values from Java

- New interfaces (only for the standard types):
  - `STRUCT` To map ADT's
  - `REF` To map values of REF types
  - `ARRAY` To map values of array types
  - `SQLDATA` To simplify the mapping of ADT

Manipulation of new type values from Java

- For each interface, the chosen driver specify a class that implements the interface
- In such way, it is possible to handle the differences among DBMS
Simple types

- No new interfaces
- They are manipulated by using the methods of the base type
- **Example**
  - `CREATE TYPE id_employee AS INTEGER FINAL;`
  - The `getInt` and `setInt` methods are used to read and write fields of type `id_employee`

ADT

- The instances of ADT types are mapped into instances of the `Struct` class
- An object of type `Struct` contains a value for each attribute of the ADT to which it corresponds
- The methods for reading and writing values of ADT types are
  - `ResultSet - getObject(int):` to access attributes if type ADT
  - `Struct - getAttributes():` to access the components of an instance
  - `PreparedStatement - setObject(int, Struct):` to set the parameters of type ADT

Example

```java
CREATE TABLE Employees (
    emp#       INTEGER PRIMARY KEY,
    name       CHAR(20),
    address    address_t);

We would like to assign to Green the address of employee # 12
```

Example

```java
String query = "select address from Employees where emp# = 12";
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(query);
rs.next();
Struct address = (Struct) rs.getObject(1);  //the instance is obtained
Object[] addr_attr = address.getAttributes();  //the attributes are accessed
```
Example

String up = "update Employees set address = ? where name = 'Green'";
PreparedStatement pst = con.prepareStatement(up);
pst.setObject(1, address);
pst.executeUpdate();

Reference types

- The instances of REF types are mapped into instance of the Ref class
- The methods to read and write attributes of REF types are
  - ResultSet - getRef(int): to access attributes of REF types
  - Ref - getObject(): given a value of REF type, it returns the referenced object (only in JDBC 3)
  - Ref - setObject(Struct): the parameter becomes the new referenced object (only in JDBC 3)
  - PreparedStatement - setRef(int, Ref): to set parameters of REF types

Example

CREATE TYPE project_t AS
  proj# INTEGER,
  name VARCHAR(20),
  description VARCHAR(50),
  budget INTEGER;

CREATE TABLE Projects of project_t;

CREATE TABLE Employees (emp# INTEGER PRIMARY KEY, name VARCHAR(50), address address_t, assignment REF(project_t));

Example

- We want to assign the project of employee #12 to Green and print the information concerning such project
Example

String query = "select assignment from Employees where emp# = 12";
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(query);
rs.next();
Ref assegn_ref = rs.getRef(1); //the identifier is obtained
String update = "update Employees set assignment = ? where name = "Green";"
PreparedStatement pst = con.prepareStatement(update);
pst.setRef(1, assegn_ref);
pst.executeUpdate();

Example

Struct project = (Struct) assegn_ref.getObject();
Object [] proj_attr = project.getAttributes();

Array types

- The instances of the array types are mapped onto instances of the Array class
- They represent pointers to the database (the content is not copied)
- the methods to read and write values of attributes of type array are:
  - ResultSet - getArray(): to access attributes of type array (it returns a pointer to the instance)
  - Array - getArray(): to copy the data contained in the array into structures local to the program
  - Array - getResultSet(): it returns a result set that contains a tuple for each element of the array, with two columns. The first contains the index (starting from 1), the second the value
  - PreparedStatement - setArray(int,Array): to set parameters of type Array

Example

CREATE TABLE Employees{
  emp# INTEGER PRIMARY KEY,
  name VARCHAR(50),
  address address_t,
  skills VARCHAR(20) ARRAY[10];
}

We want to print all skills of employee #12
**Example**

```java
String query = "select skills from Employees where emp# = 12";
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(query);
rs.next();
Array comp = rs.getArray(1); //returns a pointer to the array
String[] comp_arr = (String[]) comp.getArray();
for (int i = 0; i < comp_arr.length(); i++)
    System.out.print(comp_arr[i])
```

**Collection types**

- As we have seen, the various systems do not conform to the standard for what concerns the collection types
- the driver can eventually use the Array class also to manage other collection types
  - in JDBC the nested tables of Oracle are mapped onto objects of type Array

**SQL Data**

- Interface
- each class that implement SQLData allows one to define an explicit mapping between an SQL type and a Java class
  - The casting is thus explicitly defined by the user
- It simplifies the access to structured objects
  - Reading an object returns an instance of the class associated by the mapping
  - It is not any longer to use Struct and Object
- We do not see this topic