# Advanced Management of Data

Object-Relational Database Systems (1)

# Extensions of ORDBMSs

SQL:2011 - Extensions of ORDBMSs to overcome some limitations of RDBMS:

- row types and reference types
- user-defined types
- supertype / subtype relationships
- user-defined procedures, functions, and methods
- table inheritance
- collection types (arrays, sets, lists, multisets)
- large objects

Note that some of these extensions may be implemented differently in some ORDBMSs or not implemented at all.

# Extensions of ORDBMSs

**Row type**

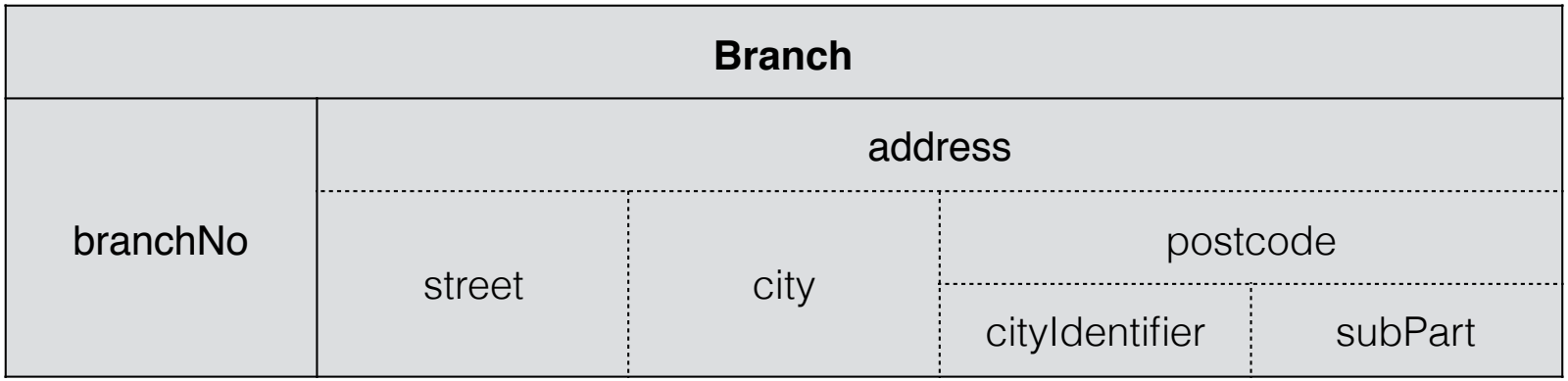
- sequence of attribute definitions that provides a new data type
- enables rows to be
  - stored in variables
  - passed as arguments to routines
  - returned as return values from function calls

  Additionally, a column of a table can contain row values. That is, each attribute component of a row type definition can be a row type itself.

  The keyword `ROW` defines a row type or a value of row type.

# Example (1)

```
CREATE TABLE Branch (
    branchNo  CHAR(4),
    address   ROW(street   VARCHAR(25),
                  city     VARCHAR(15),
                  postcode ROW(cityIdentifier VARCHAR(4),
                               subPart VARCHAR(4))));
```



# Example (2)

```
INSERT INTO Branch
VALUES (‘B005’, ROW(‘23 Deer Rd’, ‘London’, ROW(‘SW1’, '4EH')));

UPDATE Branch
SET address = ROW (‘23 Deer Rd’, ‘London’, ROW (‘SW1’, ‘4EH’))
WHERE address = ROW (‘23 Deer Rd’, ‘London’, ROW (‘SW1’, ‘4EH’));
```

| **Branch** | | | | |
|---|---|---|---|---|
| branchNo | address | | | |
| | street | city | postcode | |
| | | | cityIdentifier | subPart |
| B005 | 23 Deer Rd | London | SW1 | 4EH |

# Extensions of ORDBMSs

**User-Defined Types**

ORDBMS allow the definition of user-defined types (UDTs), which may be used in the same way as the predefined types.

UDTs are subdivided into two categories:

- Distinct Types:

  Allow differentiation between the same underlying base types. It is not possible to treat an instance of one type as an instance of the other type.

- Structured Types

  Consist of attribute definitions and routine declarations (methods), which can also be operator declarations.

  Structured attributes can be accessed using the common dot notation (.)

# Examples

Distinct Types:

```
CREATE TYPE OwnerNumberType AS VARCHAR(5) FINAL;
CREATE TYPE StaffNumberType AS VARCHAR(5) FINAL;
```

Structured Type:

```
CREATE TYPE PersonType AS (
    fName VARCHAR(15),
    lName VARCHAR(15),
    sex   CHAR);
```

Access on an instance `p` of `PersonType`:

```
p.fName = 'John'
```

# User-Defined Types

**Observer and Mutator functions**

SQL provides a pair of built-in routines for each attribute of a structured type:

- an observer (get) function, which returns the current value of the attribute
- a mutator (set) function, which sets the value of the attribute to a value specified as a parameter

Observer and mutator functions can be redefined.

**Encapsulation**

These functions are invoked whenever a user attempts to access an attribute.

# Example

Observer and mutator functions for the `fName` attribute of `PersonType`:

observer function:

```
FUNCTION fName(p PersonType) RETURNS VARCHAR(15)
  RETURN p.fName;
```

mutator function:

```
FUNCTION fName(p PersonType RESULT, newValue VARCHAR(15))
  RETURNS PersonType
  BEGIN
    p.fName = newValue;
    RETURN p;
  END;
```

# User-Defined Types

**Constructor function**

- is automatically defined to create new instances of a structured type
- has the same name and type as the UDT
- takes zero arguments
- returns a new instance of the type with the attributes set to their default value

**Constructor methods**

- can be provided by the user to initialize a newly created instance of a structured type
- must be different from the system-supplied constructor
- must differ in the number of parameters or in the data types of the parameters

# Example

**Constructor method for type *PersonType*:**

```
CREATE CONSTRUCTOR METHOD PersonType (fN VARCHAR(15),
                                      lN VARCHAR(15),
                                      sx CHAR)
  RETURNS PersonType SELF AS RESULT

  BEGIN
    SET SELF.fName = fN;
    SET SELF.lName = lN;
    SET SELF.sex = sx;
    RETURN SELF;
  END;
```

# User-Defined Types

**The `NEW` Expression**

- invokes the system-supplied constructor function:

  Example

```
SET p NEW PersonType();
```

- can be parameterized to invoke user-defined constructor methods:

  Example

```
SET p NEW PersonType('John', 'White', 'M');
```

# Example

```
CREATE TYPE PersonType AS (
    dateOfBirth DATE,
    fName       VARCHAR(15),
    lName       VARCHAR(15),
    sex         CHAR)

INSTANTIABLE   /* Instances can be created for this type */
NOT FINAL      /* subtypes can be created */
REF IS SYSTEM GENERATED
INSTANCE METHOD age () RETURNS INTEGER,            /* virtual attribute */
INSTANCE METHOD age (dob DATE) RETURNS PersonType; /* overloading */

CREATE INSTANCE METHOD age () RETURNS INTEGER
    FOR PersonType
    BEGIN
        RETURN /* age calculated from SELF.dateOfBirth */;
    END;

CREATE INSTANCE METHOD age (dob DATE) RETURNS PersonType
    FOR PersonType
    BEGIN
        SELF.dateOfBirth /* code to set dateOfBirth from dob*/;
        RETURN SELF;
    END;
```

# Extensions of ORDBMSs

**Subtypes and Supertypes**

- using the `UNDER` clause, UDTs can participate in a subtype / supertype hierarchy
- a subtype inherits all the attributes and methods of its supertype
- a subtype can define additional attributes and methods like any other UDT and it can override inherited methods

**Example**

```
CREATE TYPE StaffType UNDER PersonType AS (
    staffNo  VARCHAR(5),
    position VARCHAR(10),
    salary   DECIMAL(7,2),
    branchNo CHAR(4))
INSTANTIABLE
NOT FINAL
INSTANCE METHOD isManager () RETURNS BOOLEAN;

CREATE INSTANCE METHOD isManager() RETURNS BOOLEAN
FOR StaffType
BEGIN
    IF SELF.position = 'Manager' THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END IF
END;
```

# Subtypes and Supertypes

**Substitutability**

An instance of a subtype is considered an instance of all its supertypes. Whenever an instance of a supertype is expected, an instance of the subtype can be used instead.

**Type Test**

The type of a UDT can be tested using the `TYPE` predicate, e.g.

- Test, whether a given UDT Udt1 is the *PersonType* or any of its subtypes

```
TYPE Udt1 IS OF (PersonType)
```

- Test, whether Udt1 is the *PersonType*

```
TYPE Udt1 IS OF (ONLY PersonType)
```

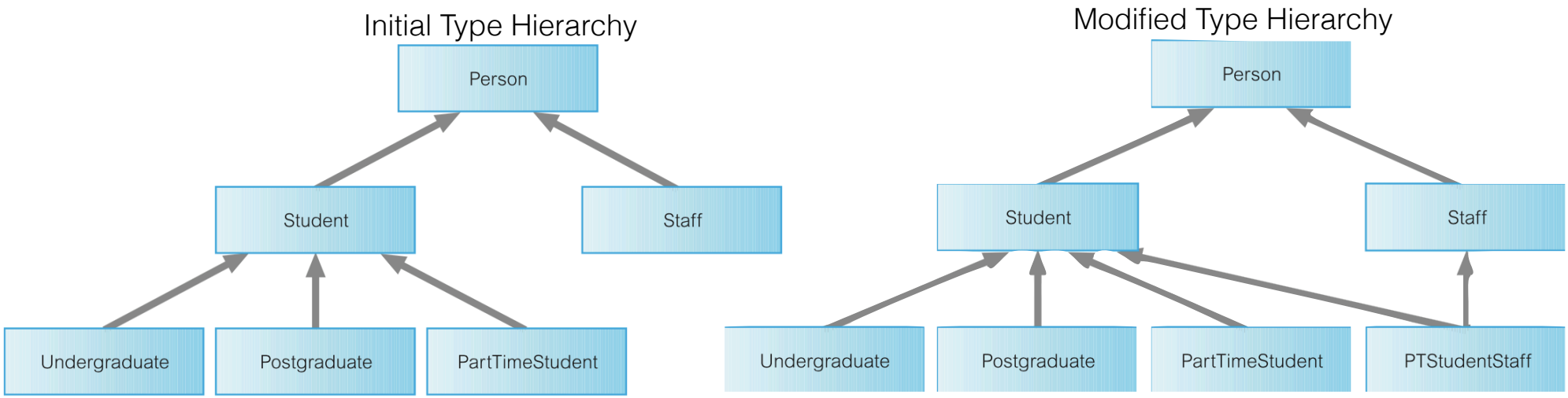
# Subtypes and Supertypes

**Most Specific Type**

Every instance of a UDT must be associated with exactly *one most specific type*, which corresponds to the lowest subtype assigned to the instance.

If the UDT has more than one direct supertypes, then there must be a single type to which the instance belongs, and that single type must be a subtype of all the types to which the instance belongs. In some cases, this can require the creation of a large number of types.

**Example -** Associating an instance to both *PartTimeStudent* and *Staff* by creating a new type




[Connolly & Begg]

# Extensions of ORDBMSs

**User-Defined Routines (UDRs)**

- define methods for manipulating data
- provide required behavior for UDTs
- can return simple and complex values and should support overloading
- may be defined as part of a UDT or separately as part of a schema
- may be externally provided in a standard programming language such as C, C++, or Java, or defined in SQL (when SQL has been extended to be computationally complete)
- may be SQL-invoked or external. An SQL-invoked routine may be a
  - procedure
  - function
  - method

# User-Defined Routines

**Procedure**

- may have zero or more parameters, each of which may be
  - an input parameter (`IN`),
  - an output parameter (`OUT`)
  - both an input and output parameter (`INOUT`)

**Function**

- An SQL-invoked function returns a value
- any parameters must be input parameters
- One input parameter can be designated as the result (using the `RESULT` keyword), in which case the parameter's data type must match the type of the `RETURN` type.

# User-Defined Routines

**Method**

- An SQL-invoked method is similar to a function but has some important differences:
  - a method is associated with a single UDT
  - the signature of every method associated with a UDT must be specified in that UDT
- There are three types of methods
  - constructor methods
  - instance methods
  - static methods

# User-Defined Routines

**`SELF`**

constructor methods and instance methods include an additional implicit first parameter called `SELF` whose data type is that of the associated UDT.

**Method invocation**

- constructor methods are invoked using the `NEW` expression
- instance methods are invoked using the standard dot notation

  examples: `p.fName`
  `(p AS StaffType).fName()` // generalized invocation format

- static methods are invoked using `::`

  example: `StaffType::totalStaff()` // if *totalStaff* is a static method of *StaffType*

# User-Defined Routines

**External Routine**

- is defined by specifying an external clause that identifies the corresponding “compiled code” in the operating system’s file storage.

**Example**

```
CREATE FUNCTION thumbnail(IN myImage ImageType) RETURNS BOOLEAN
EXTERNAL NAME '/usr/dreamhome/bin/images/thumbnail'
LANGUAGE C
PARAMETER STYLE GENERAL
DETERMINISTIC      /* A routine is deterministic if it always returns the same return
                      value(s) for a given set of inputs. */
NO SQL;            /* this function contains no SQL statements  */
```

# Polymorphism

**Overloading**

Different routines may have the same name.

Following constraints must hold:

- No two functions in the same schema are allowed to have the same signature.
- No two procedures in the same schema are allowed to have the same name and the same number of parameters.

**Example**

```
INSTANCE METHOD age () RETURNS INTEGER,
INSTANCE METHOD age (dob DATE) RETURNS PersonType;
```

# Polymorphism

**Instance method invocation**

Phase 1 - Static Analysis

- Identification of all routines with the appropriate name
- Elimination of all
  - procedures/functions/methods for which the user does not have **`EXECUTE`** privilege
  - methods that are not associated with the declared type (or subtype) of the implicit **`SELF`** argument
  - methods whose parameters are not equal to the number of arguments in the method invocation
- For the remaining methods, the system checks that the data type of each parameter matches the precedence list of the corresponding argument, eliminating those methods that do not match.
- If there are no candidate methods remaining a syntax error occurs.

# Polymorphism

**Instance method invocation**

Phase 2 - Runtime Execution

If the most specific type of the runtime value of the implicit argument to the method invocation has a type definition

- that includes one of the candidate methods, then that method is selected for execution.
- that does not include one of the candidate methods, then the method selected for execution is the candidate method whose associated type is the nearest supertype of all supertypes having such a method.

# Extensions of ORDBMSs

**Object identity (OID)**

- is that aspect of an object that never changes and that distinguishes the object from all other objects
- an object's identity is independent of its name, structure, and location
- ideally an OID persists even after the object has been deleted, so that it may never be confused with the identity of any other object
- other objects can use an object's identity as a unique way of referencing it



# Extensions of ORDBMSs

**Reference Type**

Can be used to uniquely identify a row within a table and to define relationships between row types

A value of a reference type can be stored in a table and used as a direct reference to a specific row in some base table that has been defined to be this type.

`REF IS SYSTEM GENERATED` in a `CREATE TYPE` statement indicates that the actual values of the associated `REF` type are provided by the system.

Other columns can be specified for the table but at least one column must be specified, namely a column of the associated `REF` type, using the clause

```
REF IS <columnName> SYSTEM GENERATED.
```

This column is used to contain unique identifiers for the rows of the associated base table. The identifier for a given row is assigned when the row is inserted into the table and remains associated with that row until it is deleted.

# Creating Tables

### Instances of UDTs

To maintain upwards compatibility with the SQL2 standard, it is still necessary to use the `CREATE TABLE` statement to create a table, even if the table consists of a single UDT.

### Example

```
CREATE TABLE Person (
    info PersonType
    CONSTRAINT DOB_Check CHECK(dateOfBirth > DATE '1900-01-01'));
```

The columns of a *Person* table can be accessed using a path expression such as

```
Person.info.fName
```

# Creating Tables

### Typed Tables

Typed tables are tables that have been constructed with the `CREATE TABLE ... OF` statement.

The rows of a typed table are considered to be objects.

A typed table has a column for every attribute of the structured type on which it is based.

For each row of the table there is a self-referencing column that contains a unique OID (reference).

Objects are inserted into a typed table using the normal `INSERT` statement.

Apart from the self-referencing column no additional columns can be added to the table definition.

# Example

**Typed Table**

```
CREATE TABLE Person OF PersonType (
    dateOfBirth WITH OPTIONS
    CONSTRAINT DOB_Check CHECK (dateOfBirth > DATE '1900-01-01')
    REF IS PersonID SYSTEM GENERATED);
```

Columns of Person can be accessed using a path expression such as

```
Person.fName
```

# Creating Tables

**OIDs**

- The OID is generated automatically when a new row is inserted into the typed table.
- To gain access to the OIDs stored in the self-referencing column, we have to give the column name explicitly using the **`REF IS`** clause.
- A typed table definition must also repeat a reference generation specification that is consistent with the corresponding specification in the UDT

```
CREATE TABLE Person OF PersonType (
    dateOfBirth WITH OPTIONS
    CONSTRAINT DOB_Check CHECK (dateOfBirth > DATE '1900-01-01')
    REF IS PersonID SYSTEM GENERATED);
```

# Creating Tables

**Example - Using a primary key / foreign key relationship to define a relationship**

```
CREATE TABLE PropertyForRent(
  propertyNo VARCHAR(5)     NOT NULL,
  street     VARCHAR(25)    NOT NULL,
  city       VARCHAR(15)    NOT NULL,
  postcode   VARCHAR(8),
  type       CHAR(1)        NOT NULL DEFAULT 'F',
  rooms      SMALLINT       NOT NULL DEFAULT 4,
  rent       DECIMAL(6,2)   NOT NULL DEFAULT 600,

  staffNo    VARCHAR(5)     NOT NULL,
  PRIMARY KEY (propertyNo),
  FOREIGN KEY (staffNo)     REFERENCES Staff ON DELETE SET NULL
                                             ON UPDATE CASCADE
);
```

# Creating Tables

**Example - Using a reference type to define a relationship**

```
CREATE TABLE PropertyForRent(
  propertyNo VARCHAR(5)      NOT NULL,
  street     VARCHAR(25)     NOT NULL,
  city       VARCHAR(15)     NOT NULL,
  postcode   VARCHAR(8),
  type       CHAR(1)         NOT NULL DEFAULT 'F',
  rooms      SMALLINT        NOT NULL DEFAULT 4,
  rent       DECIMAL(6,2)    NOT NULL DEFAULT 600

  staffID    REF(StaffType) SCOPE Staff
                            REFERENCES ARE CHECKED
                            ON DELETE CASCADE,
  PRIMARY KEY (propertyNo)
);
```