

Αντικείμενα στις ΒΔ



- Object-Oriented Databases
 - Need for Complex Data Types
 - The O-O Data Model
 - O-O Languages
 - Persistent Programming Languages
 - Persistent C++ Systems
- Object-Relational Databases
 - Nested Relations
 - Complex Types and Object Orientation
 - Querying with Complex Types
 - Creation of Complex Values and Objects
 - Comparison of O-O and O-R Databases

Βασική πηγή διαφανειών: Silberschatz et al., "Database System Concepts", 4/e
Εργαστήριο Πληροφοριακών Συστημάτων, Παν/μιο Πειραιώς (<http://infolab.cs.unipi.gr/>)
έκδοση: Νοέμβριος 2009

Object-Oriented Databases



- Need for Complex Data Types
- The O-O Data Model
- O-O Languages
- Persistent Programming Languages
- Persistent C++ Systems

Need for Complex Data Types



- Traditional database applications in data processing had conceptually simple data types
 - Relatively few data types, first normal form holds
- Complex data types have grown more important in recent years
 - E.g. Addresses can be viewed as a
 - Single string, or
 - Separate attributes for each part, or
 - Composite attributes (which are not in first normal form)
 - E.g. it is often convenient to store multivalued attributes as-is, without creating a separate relation to store the values in first normal form
- Applications
 - computer-aided design, computer-aided software engineering
 - multimedia and image databases, and document/hypertext databases.

Object-Oriented Data Model



- Loosely speaking, an **object** corresponds to an entity in the E-R model.
- The *object-oriented paradigm* is based on *encapsulating* code and data related to an object into single unit.
- The object-oriented data model is a logical data model (like the E-R model).
- Adaptation of the object-oriented programming paradigm (e.g., Smalltalk, C++) to database systems.

Object Structure



- An object has associated with it:
 - A set of **variables** that contain the data for the object. The value of each variable is itself an object.
 - A set of **messages** to which the object responds; each message may have zero, one, or more *parameters*.
 - A set of **methods**, each of which is a body of code to implement a message; a method returns a value as the *response* to the message
- The physical representation of data is visible only to the implementor of the object
- Messages and responses provide the only external interface to an object.
- The term message does not necessarily imply physical message passing. Messages can be implemented as procedure **invocations**.

Messages and Methods



- Methods are programs written in general-purpose language with the following features
 - only variables in the object itself may be referenced directly
 - data in other objects are referenced only by sending *messages*.
- Methods can be **read-only** or **update** methods
 - **Read-only** methods do not change the value of the object
- Strictly speaking, every attribute of an entity must be represented by a variable and two methods, one to read and the other to update the attribute
 - e.g., the attribute *address* is represented by a variable *address* and two messages *get-address* and *set-address*.
 - For convenience, many object-oriented data models permit direct access to variables of other objects.

Object Classes



- Similar objects are grouped into a **class**; each such object is called an **instance** of its class
- All objects in a class have the same
 - Variables, with the same types
 - message interface
 - methods

They may differ in the values assigned to variables
- Example: Group objects for people into a *person* class
- Classes are analogous to entity sets in the E-R model

Class Definition Example



```
class employee {
    /*Variables */
    string name;
    string address;
    date start-date;
    int salary;
    /* Messages */
    int annual-salary();
    string get-name();
    string get-address();
    int set-address(string new-address);
    int employment-length();
};
```

- Methods to read and set the other variables are also needed with strict encapsulation
- Methods are defined separately
 - E.g. `int employment-length() { return today() – start-date; }`
`int set-address(string new-address) { address = new-address; }`

Inheritance

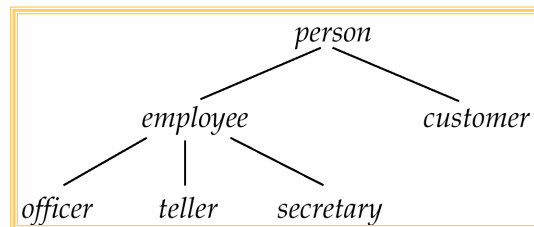


- E.g., class of bank customers is similar to class of bank employees, although there are differences
 - both share some variables and messages, e.g., *name* and *address*.
 - But there are variables and messages specific to each class e.g., *salary* for employees and *credit-rating* for customers.
- Every employee is a person; thus *employee* is a specialization of *person*
- Similarly, *customer* is a specialization of *person*.
- Create classes *person*, *employee* and *customer*
 - variables/messages applicable to all persons associated with class *person*.
 - variables/messages specific to employees associated with class *employee*; similarly for *customer*

Inheritance (Cont.)



- Place classes into a specialization/IS-A hierarchy
 - variables/messages belonging to class *officer* are *inherited* by class *employee* as well as *person*
- Result is a **class hierarchy**



Note analogy with ISA Hierarchy in the E-R model

Class Hierarchy Definition



```
class person{
    string name;
    string address;
};
class customer isa person {
    int credit-rating;
};
class employee isa person {
    date start-date;
    int salary;
};
class officer isa employee {
    int office-number,
    int expense-account-number,
};
:
```

Αντικείμενα στις ΒΔ

11

ΠΑ.ΠΕΙ. – Γιάννης Θεοδωρίδης

Class Hierarchy Example (Cont.)



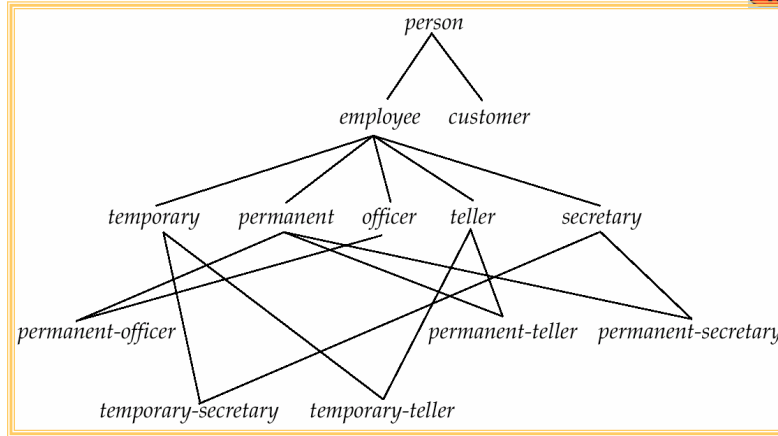
- Full variable list for objects in the class *officer*:
 - *office-number, expense-account-number*: defined locally
 - *start-date, salary*: inherited from *employee*
 - *name, address*: inherited from *person*
- Methods inherited similar to variables.
- **Substitutability** — any method of a class, say *person*, can be invoked equally well with any object belonging to any subclass, such as subclass *officer* of *person*.
- **Class extent**: set of all objects in the class. Two options:
 1. Class extent of *employee* includes all *officer, teller* and *secretary* objects.
 2. Class extent of *employee* includes only employee objects that are not in a subclass such as *officer, teller, or secretary*
 - ☞ This is the usual choice in OO systems
 - ☞ Can access extents of subclasses to find all objects of subtypes of employee

Αντικείμενα στις ΒΔ

12

ΠΑ.ΠΕΙ. – Γιάννης Θεοδωρίδης

Example of Multiple Inheritance



Class DAG for banking example.

Multiple Inheritance

- With multiple inheritance a class may have more than one superclass.
 - The class/subclass relationship is represented by a **directed acyclic graph (DAG)**
 - Particularly useful when objects can be classified in more than one way, which are independent of each other
 - E.g. temporary/permanent is independent of Officer/secretary/teller
 - Create a subclass for each combination of subclasses
 - Need not create subclasses for combinations that are not possible in the database being modeled
- A class inherits variables and methods from all its superclasses
- There is potential for ambiguity when a variable/message N with the same name is inherited from two superclasses A and B
 - Otherwise, do one of the following
 - flag as an error,
 - rename variables (A.N and B.N)
 - choose one.

More Examples of Multiple Inheritance



- Conceptually, an object can belong to each of several subclasses
 - A *person* can play the roles of *student*, a *teacher* or *footballPlayer*, or any combination of the three
 - E.g., student teaching assistant who also play football
- Can use multiple inheritance to model “roles” of an object
 - That is, allow an object to take on any one or more of a set of types
- But many systems insist an object should have a **most-specific class**
 - That is, there must be one class that an object belongs to which is a subclass of all other classes that the object belongs to
 - Create subclasses such as *student-teacher* and *student-teacher-footballPlayer* for each combination
 - When many combinations are possible, creating subclasses for each combination can become cumbersome

Object Identity



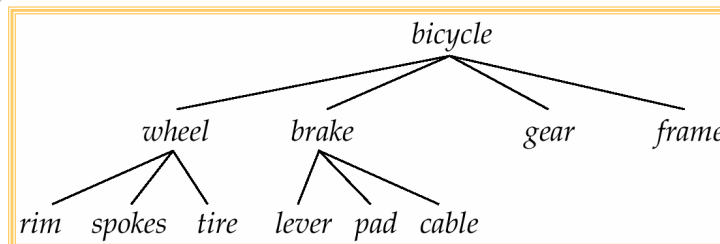
- An object retains its identity even if some or all of the values of variables or definitions of methods change over time.
- Object identity is a stronger notion of identity than in programming languages or data models not based on object orientation.
 - **Value** – data value; e.g. primary key value used in relational systems.
 - **Name** – supplied by user; used for variables in procedures.
 - **Built-in** – identity built into data model or programming language.
 - no user-supplied identifier is required.
 - Is the form of identity used in object-oriented systems.

Object Identifiers



- **Object identifiers** used to uniquely identify objects
 - Object identifiers are **unique**:
 - no two objects have the same identifier
 - each object has only one object identifier
 - E.g., the *spouse* field of a *person* object may be an identifier of another *person* object.
 - can be stored as a field of an object, to refer to another object.
 - Can be
 - system generated (created by database) or
 - external (such as social-security number)
 - System generated identifiers:
 - Are easier to use, but cannot be used across database systems
 - May be redundant if unique identifier already exists

Object Containment



- Each component in a design may contain other components
- Can be modeled as containment of objects. Objects containing other objects are called **composite** objects.
- Multiple levels of containment create a **containment hierarchy**
 - links interpreted as **is-part-of**, not **is-a**.
- Allows data to be viewed at different granularities by different users.

Object-Oriented Languages



- Object-oriented concepts can be used in different ways
 - Object-orientation can be used as a design tool, and be encoded into, for example, a relational database
 - analogous to modeling data with E-R diagram and then converting to a set of relations)
 - The concepts of object orientation can be incorporated into a programming language that is used to manipulate the database.
 - **Object-relational systems** – add complex types and object-orientation to relational language.
 - **Persistent programming languages** – extend object-oriented programming language to deal with databases by adding concepts such as persistence and collections.

Persistent Programming Languages



- Persistent Programming languages allow objects to be created and stored in a database, and used directly from a programming language
 - allow data to be manipulated directly from the programming language
 - No need to go through SQL.
 - No need for explicit format (type) changes
 - format changes are carried out transparently by system
 - Without a persistent programming language, format changes becomes a burden on the programmer
 - More code to be written
 - More chance of bugs
 - allow objects to be manipulated in-memory
 - no need to explicitly load from or store to the database
 - Saved code, and saved overhead of loading/storing large amounts of data

Persistent Prog. Languages (Cont.)



- Drawbacks of persistent programming languages
 - Due to power of most programming languages, it is easy to make programming errors that damage the database.
 - Complexity of languages makes automatic high-level optimization more difficult.
 - Do not support declarative querying as well as relational databases

Persistence of Objects



- Approaches to make transient objects persistent include establishing
 - **Persistence by Class** – declare all objects of a class to be persistent; simple but inflexible.
 - **Persistence by Creation** – extend the syntax for creating objects to specify that that an object is persistent.
 - **Persistence by Marking** – an object that is to persist beyond program execution is marked as persistent before program termination.
 - **Persistence by Reachability** - declare (root) persistent objects; objects are persistent if they are referred to (directly or indirectly) from a root object.
 - Easier for programmer, but more overhead for database system
 - Similar to garbage collection used e.g. in Java, which also performs reachability tests

Object Identity and Pointers



- A persistent object is assigned a persistent object identifier.
- Degrees of permanence of identity:
 - **Intraprocedure** – identity persists only during the executions of a single procedure
 - **Intraprogram** – identity persists only during execution of a single program or query.
 - **Interprogram** – identity persists from one program execution to another, but may change if the storage organization is changed
 - **Persistent** – identity persists throughout program executions and structural reorganizations of data; required for object-oriented systems.

Object Identity and Pointers (Cont.)



- In O-O languages such as C++, an object identifier is actually an in-memory pointer.
- **Persistent pointer** – persists beyond program execution
 - can be thought of as a pointer into the database
 - E.g. specify file identifier and offset into the file
 - Problems due to database reorganization have to be dealt with by keeping **forwarding pointers**

Storage and Access of Persistent Objects



How to find objects in the database:

- Name objects (as you would name files)
 - Cannot scale to large number of objects.
 - Typically given only to class extents and other collections of objects, but not objects.
- Expose object identifiers or persistent pointers to the objects
 - Can be stored externally.
 - All objects have object identifiers.
- Store collections of objects, and allow programs to iterate over the collections to find required objects
 - Model collections of objects as **collection types**
 - **Class extent** - the collection of all objects belonging to the class; usually maintained for all classes that can have persistent objects.

Persistent C++ Systems



- C++ language allows support for persistence to be added without changing the language
 - Declare a class called **Persistent_Object** with attributes and methods to support persistence
 - **Overloading** – ability to redefine standard function names and operators (i.e., +, -, the pointer dereference operator ->) when applied to new types
 - **Template classes** help to build a type-safe type system supporting collections and persistent types.
- Providing persistence without extending the C++ language is
 - relatively easy to implement
 - but more difficult to use
- Persistent C++ systems that add features to the C++ language have been built, as also systems that avoid changing the language

ODMG C++ Object Definition Language



- The Object Database Management Group is an industry consortium aimed at standardizing object-oriented databases
 - in particular persistent programming languages
 - includes standards for C++, Smalltalk and Java
 - ODMG-93
 - ODMG-2.0 and 3.0 (which is 2.0 plus extensions to Java)
 - Our description is based on ODMG-2.0
- ODMG C++ standard avoids changes to the C++ language
 - provides functionality via template classes and class libraries

ODMG Types



- Template class **d_Ref<class>** used to specify references (persistent pointers)
- Template class **d_Set<class>** used to define sets of objects.
 - Methods include **insert_element(e)** and **delete_element(e)**
- Other collection classes such as **d_Bag** (set with duplicates allowed), **d_List** and **d_Varray** (variable length array) also provided.
- **d_** version of many standard types provided, e.g. **d_Long** and **d_string**
 - Interpretation of these types is platform independent
 - Dynamically allocated data (e.g. for **d_string**) allocated in the database, not in main memory

ODMG C++ ODL: Example



```
class Person : public d_Object {
public:
    d_String  name;    // should not use String!
    d_String  address;
};

class Account : public d_Object {
private:
    d_Long    balance;
public:
    d_Long    number;
    d_Set <d_Ref<Person>> owners;

    int       find_balance();
    int       update_balance(int delta);
};
```

Object-Relational Databases



- Nested Relations
- Complex Types and Object Orientation
- Querying with Complex Types
- Creation of Complex Values and Objects
- Comparison of O-O and O-R Databases

Nested Relations



- Motivation:
 - Permit non-atomic domains (atomic \equiv indivisible)
 - Example of non-atomic domain: set of integers, or set of tuples
 - Allows more intuitive modeling for applications with complex data
- Intuitive definition:
 - allow relations whenever we allow atomic (scalar) values — relations within relations
 - Retains mathematical foundation of relational model
 - Violates first normal form.

Αντικείμενα στις ΒΔ

31

ΠΑ.ΠΕΙ. – Γιάννης Θεοδωρίδης

Example of a Nested Relation



- Example: library information system
- Each book has
 - title,
 - a set of authors,
 - Publisher, and
 - a set of keywords
- Non-1NF relation *books*

<i>title</i>	<i>author-set</i>	<i>publisher</i> (<i>name, branch</i>)	<i>keyword-set</i>
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

Αντικείμενα στις ΒΔ

32

ΠΑ.ΠΕΙ. – Γιάννης Θεοδωρίδης

1NF Version of Nested Relation



<i>title</i>	<i>author</i>	<i>pub-name</i>	<i>pub-branch</i>	<i>keyword</i>
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

flat-books

4NF Decomposition of Nested Relation



- Remove awkwardness of *flat-books* by assuming that the following multivalued dependencies hold:
 - $title \twoheadrightarrow author$
 - $title \twoheadrightarrow keyword$
 - $title \twoheadrightarrow pub-name, pub-branch$
- Decompose *flat-doc* into 4NF using the schemas:
 - $(title, author)$
 - $(title, keyword)$
 - $(title, pub-name, pub-branch)$

4NF Decomposition of *flat-books*



<i>title</i>	<i>author</i>
Compilers	Smith
Compilers	Jones
Networks	Jones
Networks	Frick

authors

<i>title</i>	<i>keyword</i>
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

<i>title</i>	<i>pub-name</i>	<i>pub-branch</i>
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

Problems with 4NF Schema



- 4NF design requires users to include joins in their queries.
- 1NF relational view *flat-books* defined by join of 4NF relations:
 - eliminates the need for users to perform joins,
 - but loses the one-to-one correspondence between tuples and documents.
 - And has a large amount of redundancy
- Nested relations representation is much more natural here.

Complex Types and SQL:1999



- Extensions to SQL to support complex types include:
 - Collection and large object types
 - Nested relations are an example of collection types
 - Structured types
 - Nested record structures like composite attributes
 - Inheritance
 - Object orientation
 - Including object identifiers and references
- Our description is mainly based on the SQL:1999 standard
 - Not fully implemented in any database system currently
 - But some features are present in each of the major commercial database systems
 - Read the manual of your database system to see what it supports
 - We present some features that are not in SQL:1999
 - These are noted explicitly

Collection Types



- Set type (not in SQL:1999)


```
create table books (
    ....
    keyword-set setof(varchar(20))
    ....
)
```
- Sets are an instance of collection types. Other instances include
 - Arrays (are supported in SQL:1999)
 - E.g. *author-array* **varchar**(20) **array**[10]
 - Can access elements of array in usual fashion:
 - E.g. *author-array*[1]
 - Multisets (not supported in SQL:1999)
 - I.e., unordered collections, where an element may occur multiple times
 - Nested relations are sets of tuples
 - SQL:1999 supports arrays of tuples

Large Object Types



- Large object types
 - **clob**: Character large objects
book-review **clob**(10KB)
 - **blob**: binary large objects
image **blob**(10MB)
movie **blob** (2GB)
- JDBC/ODBC provide special methods to access large objects in small pieces
 - Similar to accessing operating system files
 - Application retrieves a **locator** for the large object and then manipulates the large object from the host language

Structured and Collection Types



- Structured types can be declared and used in SQL


```
create type Publisher as
  (name      varchar(20),
   branch    varchar(20))
create type Book as
  (title      varchar(20),
   author-array varchar(20) array [10],
   pub-date   date,
   publisher   Publisher,
   keyword-set setof(varchar(20)))
```

 - Note: **setof** declaration of keyword-set is not supported by SQL:1999
 - Using an array to store authors lets us record the order of the authors
- Structured types can be used to create tables


```
create table books of Book
```

 - Similar to the nested relation books, but with array of authors instead of set

Structured and Collection Types (Cont.)



- Structured types allow composite attributes of E-R diagrams to be represented directly.
- Unnamed row types can also be used in SQL:1999 to define composite attributes
 - E.g.** we can omit the declaration of type *Publisher* and instead use the following in declaring the type *Book*

```
publisher row (name varchar(20),
               branch varchar(20))
```
- Similarly, collection types allow multivalued attributes of E-R diagrams to be represented directly.

Structured Types (Cont.)



- We can create tables without creating an intermediate type
 - For example, the table *books* could also be defined as follows:

```
create table books
  (title varchar(20),
   author-array varchar(20) array[10],
   pub-date date,
   publisher Publisher
   keyword-list setof(varchar(20)))
```
- Methods can be part of the type definition of a structured type:

```
create type Employee as (
  name varchar(20),
  salary integer)
method giveraise (percent integer)
```
- We create the method body separately

```
create method giveraise (percent integer) for Employee
begin
  set self.salary = self.salary + (self.salary * percent) / 100;
end
```

Creation of Values of Complex Types



- Values of structured types are created using constructor functions
 - E.g. *Publisher*('McGraw-Hill', 'New York')
 - Note: a value is **not** an object
- SQL:1999 constructor functions
 - E.g.


```
create function Publisher (n varchar(20), b varchar(20))
returns Publisher
begin
  set name=n;
  set branch=b;
end
```
 - Every structured type has a default constructor with no arguments, others can be defined as required
- Values of **row** type can be constructed by listing values in parantheses
 - E.g. given row type **row** (*name* varchar(20), *branch* varchar(20))
 - We can assign ('McGraw-Hill', 'New York') as a value of above type

Αντικείμενα στις ΒΔ

43

ΠΑ.ΠΕΙ. – Γιάννης Θεοδωρίδης

Creation of Values of Complex Types



- Array construction


```
array ['Silberschatz', 'Korth', 'Sudarshan']
```
- Set value attributes (not supported in SQL:1999)
 - **set**(*v1*, *v2*, ..., *vn*)
- To create a tuple of the *books* relation


```
('Compilers', array['Smith', 'Jones'],
  Publisher('McGraw-Hill', 'New York'),
  set('parsing', 'analysis'))
```
- To insert the preceding tuple into the relation *books*

```
insert into books
values
('Compilers', array['Smith', 'Jones'],
  Publisher('McGraw Hill', 'New York'),
  set('parsing', 'analysis'))
```

Αντικείμενα στις ΒΔ

44

ΠΑ.ΠΕΙ. – Γιάννης Θεοδωρίδης

Inheritance

- Suppose that we have the following type definition for people:



```
create type Person
(name varchar(20),
address varchar(20))
```

- Using inheritance to define the student and teacher types

```
create type Student
under Person
(degree varchar(20),
department varchar(20))
create type Teacher
under Person
(salary integer,
department varchar(20))
```

- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration

Multiple Inheritance



- SQL:1999 does not support multiple inheritance
- If our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

```
create type Teaching Assistant
under Student, Teacher
```

- To avoid a conflict between the two occurrences of *department* we can rename them

```
create type Teaching Assistant
under
Student with (department as student-dept),
Teacher with (department as teacher-dept)
```

Reference Types



- Object-oriented languages provide the ability to create and refer to objects.
- In SQL:1999
 - References are to tuples, and
 - References must be scoped,
 - I.e., can only point to tuples in one specified table
- We will study how to define references first, and later see how to use references

Reference Declaration in SQL:1999



- E.g. define a type *Department* with a field *name* and a field *head* which is a reference to the type *Person*, with table *people* as scope

```
create type Department(
    name varchar(20),
    head ref(Person) scope people)
```

- We can then create a table *departments* as follows

```
create table departments of Department
```

- We can omit the declaration **scope** *people* from the type declaration and instead make an addition to the create table statement:

```
create table departments of Department
    (head with options scope people)
```


Initializing Reference Typed Values



- In Oracle, to create a tuple with a reference value, we can first create the tuple with a null reference and then set the reference separately by using the function **ref(p)** applied to a tuple variable

- E.g. to create a department with name CS and head being the person named John, we use

```
insert into departments
values (`CS`, null)
update departments
set head = (select ref(p)
             from people as p
             where name=`John`)
where name = `CS`
```

Initializing Reference Typed Values (Cont.)



- SQL:1999 does not support the **ref()** function, and instead requires a special attribute to be declared to store the object identifier
- The self-referential attribute is declared by adding a **ref is** clause to the create table statement:

```
create table people of Person
ref is oid system generated
```

- Here, *oid* is an attribute name, not a keyword.
- To get the reference to a tuple, the subquery shown earlier would use

```
select p.oid
instead of select ref(p)
```

User Generated Identifiers



- SQL:1999 allows object identifiers to be user-generated
 - The type of the object-identifier must be specified as part of the type definition of the referenced table, and
 - The table definition must specify that the reference is user generated
 - E.g.

```
create type Person
(name varchar(20)
 address varchar(20))
ref using varchar(20)
create table people of Person
ref is oid user generated
```

- When creating a tuple, we must provide a unique value for the identifier (assumed to be the first attribute):

```
insert into people values
('01284567', 'John', '23 Coyote Run')
```

Αντικείμενα στις ΒΔ

51

ΠΑ.ΠΕΙ. – Γιάννης Θεοδωρίδης

User Generated Identifiers (Cont.)



- We can then use the identifier value when inserting a tuple into *departments*
 - Avoids need for a separate query to retrieve the identifier:

```
E.g. insert into departments
values('CS', '02184567')
```

- It is even possible to use an existing primary key value as the identifier, by including the **ref from** clause, and declaring the reference to be **derived**

```
create type Person
(name varchar(20) primary key,
 address varchar(20))
ref from(name)
create table people of Person
ref is oid derived
```

- When inserting a tuple for *departments*, we can then use

```
insert into departments
values('CS', 'John')
```

Αντικείμενα στις ΒΔ

52

ΠΑ.ΠΕΙ. – Γιάννης Θεοδωρίδης

Path Expressions



- Find the names and addresses of the heads of all departments:

```
select head->name, head->address  
from departments
```
- An expression such as "head->name" is called a **path expression**
- Path expressions help avoid explicit joins
 - If department head were not a reference, a join of *departments* with *people* would be required to get at the address
 - Makes expressing the query much easier for the user

Querying with Structured Types



- Find the title and the name of the publisher of each book.

```
select title, publisher.name  
from books
```

Note the use of the dot notation to access fields of the composite attribute (structured type) *publisher*

Collection-Value Attributes



- Collection-valued attributes can be treated much like relations, using the keyword **unnest**
 - The *books* relation has array-valued attribute *author-array* and set-valued attribute *keyword-set*
- To find all books that have the word "database" as one of their keywords,


```
select title
from books
where 'database' in (unnest(keyword-set))
```

 - Note: Above syntax is valid in SQL:1999, but the only collection type supported by SQL:1999 is the array type
- To get a relation containing pairs of the form "title, author-name" for each book and each author of the book

```
select B.title, A
from books as B, unnest (B.author-array) as A
```

Collection Valued Attributes (Cont.)



- We can access individual elements of an array by using indices
 - E.g. If we know that a particular book has three authors, we could write:

```
select author-array[1], author-array[2], author-array[3]
from books
where title = 'Database System Concepts'
```

Unnesting



- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called **unnesting**.
- E.g.

```
select title, A as author, publisher.name as pub_name,
       publisher.branch as pub_branch, K as keyword
from books as B, unnest(B.author-array) as A, unnest
(B.keyword-list) as K
```

Nesting



- **Nesting** is the opposite of unnesting, creating a collection-valued attribute
- NOTE: SQL:1999 does not support nesting
- Nesting can be done in a manner similar to aggregation, but using the function set() in place of an aggregation operation, to create a set
- To nest the *flat-books* relation on the attribute *keyword*:

```
select title, author, Publisher(pub_name, pub_branch) as publisher,
       set(keyword) as keyword-list
from flat-books
groupby title, author, publisher
```

- To nest on both authors and keywords:

```
select title, set(author) as author-list,
       Publisher(pub_name, pub_branch) as publisher,
       set(keyword) as keyword-list
from flat-books
groupby title, publisher
```

Nesting (Cont.)



- Another approach to creating nested relations is to use subqueries in the select clause.

```
select title,
  ( select author
    from flat-books as M
    where M.title=O.title) as author-set,
  Publisher(pub-name, pub-branch) as publisher,
  (select keyword
    from flat-books as N
    where N.title = O.title) as keyword-set
from flat-books as O
```

- Can use **orderby** clause in nested query to get an ordered collection
 - Can thus create arrays, unlike earlier approach

Functions and Procedures



- SQL:1999 supports functions and procedures
 - Functions/procedures can be written in SQL itself, or in an external programming language
 - Functions are particularly useful with specialized data types such as images and geometric objects
 - E.g. functions to check if polygons overlap, or to compare images for similarity
 - Some databases support **table-valued functions**, which can return a relation as a result
- SQL:1999 also supports a rich set of imperative constructs, including
 - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

SQL Functions



- Define a function that, given a book title, returns the count of the number of authors (on the 4NF schema with relations *books4* and *authors*).

```
create function author-count(name varchar(20))
returns integer
begin
  declare a-count integer;
  select count(author) into a-count
  from authors
  where authors.title=name
  return a=count;
end
```

- Find the titles of all books that have more than one author.

```
select name
from books4
where author-count(title)> 1
```

SQL Methods



- Methods can be viewed as functions associated with structured types
 - They have an implicit first parameter called **self** which is set to the structured-type value on which the method is invoked
 - The method code can refer to attributes of the structured-type value using the **self** variable
 - E.g. **self.a**

SQL Functions and Procedures (cont.)



- The *author-count* function could instead be written as procedure:

```
create procedure author-count-proc (in title varchar(20),
                                     out a-count integer)
begin
    select count(author) into a-count
    from authors
    where authors.title = title
end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the call statement.
 - E.g. from an SQL procedure

```
declare a-count integer;
call author-count-proc(`Database systems Concepts`, a-count);
```

- SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ

External Language Functions/Procedures



- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

```
create procedure author-count-proc(in title varchar(20),
                                     out count integer)
```

```
language C
external name '/usr/avi/bin/author-count-proc'
```

```
create function author-count(title varchar(20))
returns integer
language C
external name '/usr/avi/bin/author-count'
```


External Language Routines (Cont.)



- Benefits of external language functions/procedures:
 - more efficient for many operations, and more expressive power
- Drawbacks
 - Code to implement function may need to be loaded into database system and executed in the database system's address space
 - risk of accidental corruption of database structures
 - security risk, allowing users access to unauthorized data
 - There are alternatives, which give good security at the cost of potentially worse performance
 - Direct execution in the database system's space is used when efficiency is more important than security

Procedural Constructs



- SQL:1999 supports a rich variety of procedural constructs
- Compound statement
 - is of the form **begin ... end**,
 - may contain multiple SQL statements between **begin** and **end**.
 - Local variables can be declared within a compound statements
- While and repeat statements


```

declare  $n$  integer default 0;
while  $n < 10$  do
    set  $n = n + 1$ 
end while

repeat
    set  $n = n - 1$ 
until  $n = 0$ 
end repeat
            
```

Procedural Constructs (Cont.)



- For loop
 - Permits iteration over all results of a query
 - E.g. find total of all balances at the Perryridge branch

```
declare n integer default 0;
for r as
  select balance from account
  where branch-name = 'Perryridge'
do
  set n = n + r.balance
end for
```

Αντικείμενα στις ΒΔ

67

ΠΑ.ΠΕΙ. – Γιάννης Θεοδωρίδης

Procedural Constructs (cont.)



- Conditional statements (if-then-else)
E.g. To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)

```
if r.balance < 1000
  then set l = l + r.balance
elseif r.balance < 5000
  then set m = m + r.balance
else set h = h + r.balance
end if
```

- SQL:1999 also supports a **case** statement similar to C case statement
- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_stock condition
declare exit handler for out_of_stock
begin
  ...
  .. signal out-of-stock
end
```

- The handler here is **exit** -- causes enclosing begin..end to be exited
- Other actions possible on exception

Αντικείμενα στις ΒΔ

68

ΠΑ.ΠΕΙ. – Γιάννης Θεοδωρίδης

Comparison of O-O and O-R Databases



- Summary of strengths of various database systems:
- **Relational systems**
 - simple data types, powerful query languages, high protection.
- **Persistent-programming-language-based OODBs**
 - complex data types, integration with programming language, high performance.
- **Object-relational systems**
 - complex data types, powerful query languages, high protection.
- Note: Many real systems blur these boundaries
 - E.g. persistent programming language built as a wrapper on a relational database offers first two benefits, but may have poor performance.