# Unit 1: Object Oriented Databases

## Outline

- Overview
- Complex Data Types
- Structure Types and Inheritance in SQL
- Table Inheritance
- Arrays and Multiset Types in SQL
- Object-Identity and Reference Types in SQL
- Implementing O-R Features
- Persistent Programming Languages
- Object-Relational Mapping
- Object-Oriented versus Object-Relational

## Overview

- Traditional database applications consist of data-processing tasks with relatively simple data types that are well suited to the relational data model.
- As database systems were applied to a wider range of applications, such as computer-aided design and geographical information systems, limitations imposed by the relational model emerged as an obstacle.
- **The first obstacle** faced by programmers using the relational data model was the **limited type system supported** by the relational model.
- **The second obstacle** was the difficulty in accessing database data from programs written in programming languages such as C++ or Java.

# Complex Data Types

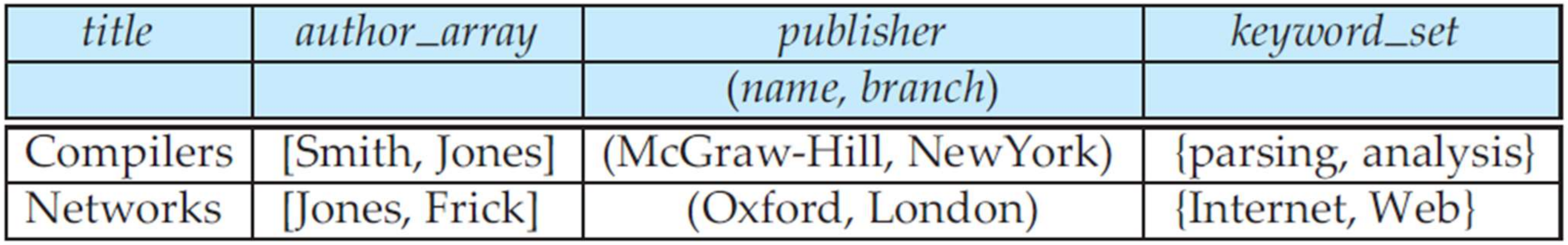
- Traditional database applications have conceptually simple data types. The basic data items are records whose fields are **atomic** (atomic ≡ indivisible).
- In recent years, **demand has grown** for ways to deal with more complex data types.
- Example:- Address
  - Entire address could be viewed as an atomic data item of type string.
  - An address were represented by breaking it into the components (street address, city, state, and postal code).
- **A better alternative is to allow structured data types** that allow a type *address with subparts street address, city, state, and postal code.*

## Example

- For example, a library application, and suppose we wish to store the following information for each book:
  - Book title.
  - List of authors.
  - Publisher.
  - Set of keywords.
- **Authors:-** A book may have a list of authors, which we can represent as an array. Nevertheless, we may want to find all books of which Jones was one of the authors. Thus, we are interested in a subpart of the domain element "authors."
- **Keywords:-** If we store a set of keywords for a book, we expect to be able to retrieve all books whose keywords include one or more specified keywords. Thus, we view the domain of the set of keywords as nonatomic.

## Example(cont…)

- **Publisher:-** Unlike *keywords and authors, publisher does not have a set-valued* domain. However, we may view *publisher as consisting of the subfields name* and *branch. This view makes the domain of publisher nonatomic.*



| *title* | *author_array* | *publisher* | *keyword_set* |
|---|---|---|---|
| | | *(name, branch)* | |
| Compilers | [Smith, Jones] | (McGraw-Hill, NewYork) | {parsing, analysis} |
| Networks | [Jones, Frick] | (Oxford, London) | {Internet, Web} |

Fig:- Non-1NF books relation, books

## Example(cont…)

- Suppose for simplicity that title uniquely identifies a book
  - In real world ISBN is a unique identifier
- Decompose *books* into 4NF using the schemas:
  - (*title, author, position*)
  - (*title, keyword* )
  - (*title, pub-name, pub-branch* )
- 4NF design requires users to include joins in their queries.

| *title* | *author* | *position* |
|---|---|---|
| Compilers | Smith | 1 |
| Compilers | Jones | 2 |
| Networks | Jones | 1 |
| Networks | Frick | 2 |

*authors*

| *title* | *keyword* |
|---|---|
| Compilers | parsing |
| Compilers | analysis |
| Networks | Internet |
| Networks | Web |

*keywords*

| *title* | *pub_name* | *pub_branch* |
|---|---|---|
| Compilers | McGraw-Hill | New York |
| Networks | Oxford | London |

Fig:- 4NF version of the relation books.

## Structured Types and Inheritance in SQL

- **Structured types** (a.k.a. **user-defined types**) can be declared and used in SQL

```
create type Name as
        (firstname      varchar(20),
         lastname       varchar(20))
                final

    create type Address as
        (street     varchar(20),
         city       varchar(20),
         zipcode    varchar(20))
                not final
```

- Note: **final -** subtypes may not be created from the given type

  **not final -** subtypes may be created.

## Structured Types and Inheritance in SQL (Cont...)

- Structured types can be used to create tables with composite attributes

```
create table person (
            name        Name,
            address     Address,
            dateOfBirth date)
```

- Dot notation used to reference components:- *name.firstname*

## Structured Types and Inheritance in SQL (Cont...)

- **User-defined row types**

```
create type PersonType as (
        name Name,
        address Address,
        dateOfBirth date
    )
        not final
```

- Can then create a table whose rows are a user-defined type

  **create table** *person* **of** *PersonType*

Alternative using **Unnamed row types**.

```
create table person_r(

name        row(firstname  varchar(20),
                lastname   varchar(20)),

address     row(street    varchar(20),
                city      varchar(20),
                zipcode   varchar(20)),

dateOfBirth date)
```

The following query illustrates how to access component attributes of a composite attribute. The query finds the last name and city of each person.

**select** ***name.lastname, address.city*** **from** ***person;***

## Structured Types and Inheritance in SQL (Cont…)

- A structured type can have **methods** defined on it. We declare methods as part of the type definition of a structured type:

```
create type PersonType as (
        name Name,
        address Address,
        dateOfBirth date)
        not final
method ageOnDate(onDate date)
        returns interval year;
```

- We create the method body separately:

```
create instance method ageOnDate (onDate date)
        returns interval year
        for PersonType
begin
        return onDate − self.dateOfBirth;
end
```

## Structured Types and Inheritance in SQL (Cont...)

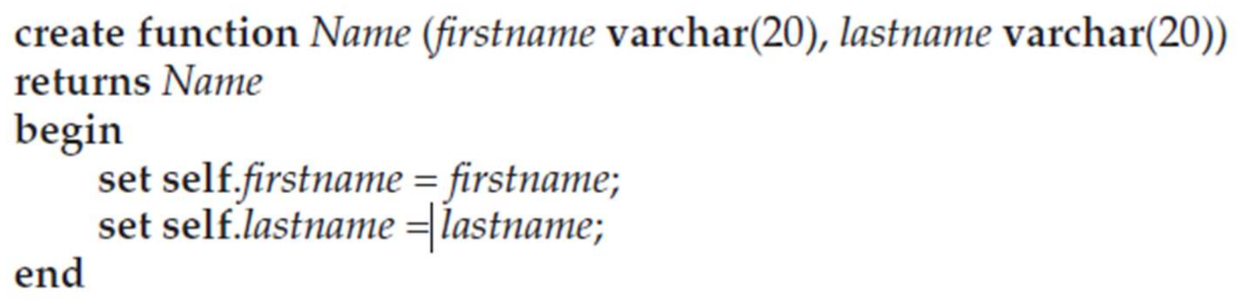
- **for** clause indicates which type this method is for.
- The keyword **instance** indicates that this method executes on an instance of the *Person* type.
- The variable **self** refers to the *Person instance on which the method is* invoked.
- Methods can be invoked on instances of a type. If we had created a table *person of type PersonType, we could invoke the method ageOnDate() as illustrated* below, to find the age of each person.

**Select *name.lastname, ageOnDate(current date)***

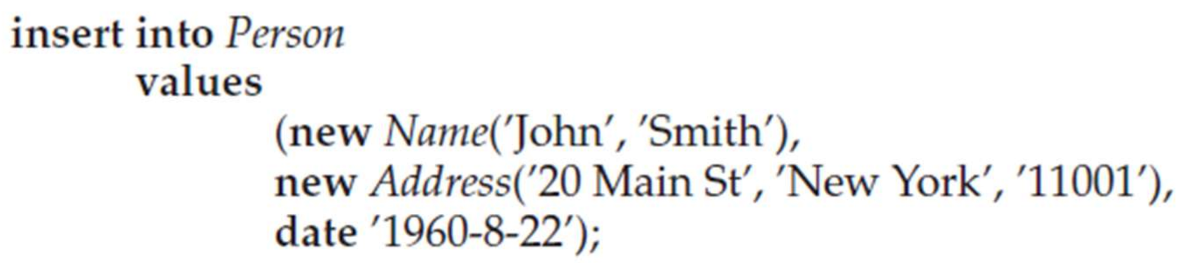
**from *person;***

## Structured Types and Inheritance in SQL (Cont...)

- In SQL:1999, **constructor functions** are used to create values of structured types.
- A function with the same name as a structured type is a constructor function for the structured type.



```
create function Name (firstname varchar(20), lastname varchar(20))
returns Name
begin
    set self.firstname = firstname;
    set self.lastname = lastname;
end
```

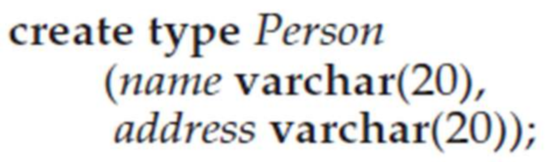


```
insert into Person
    values
        (new Name('John', 'Smith'),
        new Address('20 Main St', 'New York', '11001'),
        date '1960-8-22');
```

## Structured Types and Inheritance in SQL (Cont...)

- **Type Inheritance**

  Suppose that we have the following type definition for people:



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

- We may want to store extra information in the database about people who are students, and about people who are teachers. Since students and teachers are also people, we can use inheritance to define the student and teacher types in SQL:

## Structured Types and Inheritance in SQL (Cont...)

```
create type Student
    under Person
    (degree varchar(20),
     department varchar(20));

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

Both Student and Teacher inherit the attributes of Person—namely, name and address. Student and Teacher are said to be subtypes of Person, and Person is a supertype of Student, as well as of Teacher.

## Structured Types and Inheritance in SQL (Cont...)

- **Methods** of a structured type are inherited by its subtypes, just as attributes are.
- However, a subtype can redefine the effect of a method by declaring the method again, using **overriding method** in place of method in the method declaration.
- Suppose that we want to store information about **teaching assistants**, who are simultaneously students and teachers, perhaps even in different departments.
- We can do this if the type system supports **multiple inheritance**, where a type is declared as a subtype of multiple types. Note that the **SQL standard does not support multiple inheritance**.

## Structured Types and Inheritance in SQL (Cont...)

- For instance, if our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

```
create type TeachingAssistant
    under Student, Teacher;
```

- There is a problem since the attributes **name, address, and department are present in Student, as well as in Teacher.**
- A teaching assistant may be a student of one department and a teacher in another department. **To avoid a conflict between** the two occurrences of department, we can rename them by using an as clause, **as** in this definition of the type TeachingAssistant:

```
create type TeachingAssistant
    under Student with (department as student_dept),
        Teacher with (department as teacher_dept);
```

## Table Inheritance

- Subtables in SQL correspond to the E-R notion of specialization/generalization.
- For instance, suppose we define the people table as follows:

```
create table people of Person;
```

- We can then define tables students and teachers as subtables of people, as follows:

```
create table students of Student
    under people;

create table teachers of Teacher
    under people;
```

## Table Inheritance

- When we declare students and teachers as subtables of people**, every tuple present in students or teachers becomes implicitly present in people**.
- If a query uses the table people, it will find not only tuples directly inserted into that table, but also tuples inserted into its subtables, namely students and teachers.
- SQL permits us to find tuples that are in people but not in its sub tables by using **“only people”** in place of people in a query. The **only** keyword can also be used in delete and update statements.
- Conceptually, multiple inheritance is possible with tables, just as it is possible with types. For example, we can create a table of type TeachingAssistant:

```
create table teaching_assistants
of TeachingAssistant
    under students, teachers;
```

## Table Inheritance

- There are some consistency requirements for subtables.
- Definition: we say that tuples in a subtable and parent table **correspond** if they have the same values for all inherited attributes. Thus, corresponding tuples represent the same entity.
- The consistency requirements for subtables are:
    1. Each tuple of the supertable can correspond to at most one tuple in each of its immediate subtables.
    2. SQL has an additional constraint that all the tuples corresponding to each other must be derived from one tuple (inserted into one table).

## Array and Multiset Types in SQL

- SQL supports two collection types: **arrays** and **Multiset**; array types were added in SQ L:1999, while Multiset types were added in SQL:2003.
- ***Multiset*** *is* an **unordered collection**, where an element may occur multiple times and ***Arrays*** are ordered collection.
- Suppose we wish to record information about books, including a set of keywords for each book.

```
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 varchar(20) multiset);

create table books of Book;
```

## Array and Multiset Types in SQL(cont...)

- **Creating and Accessing Collection Values**
- An array of values can be created in SQL:1999 in this way:

**array**['Silberschatz', 'Korth', 'Sudarshan']

- Similarly, a multiset of keywords can be constructed as follows:

**multiset**['computer', 'database', 'SQL']

- Thus, we can create a tuple of the type defined by the *books relation as:*

('Compilers', **array**['Smith', 'Jones'], **new** *Publisher*('McGraw-Hill', 'New York'), **multiset**['parsing', 'analysis'])

## Array and Multiset Types in SQL(cont…)

- If we want to insert the preceding tuple into the relation *books, we could* execute the statement:

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

- We can access or update elements of an array by specifying the array index, for example ***author array[1].***

## Array and Multiset Types in SQL(cont…)

- **Querying Collection-Valued Attributes**
- We use the table *books that we defined earlier.*
- If we want to find all books that have the word “**database**” as one of their keywords, we can use this query:

```
select title
from books
where 'database' in (unnest(keyword_set));
```

- Suppose that we want a relation containing pairs of the form “title, author name” for each book and each author of the book. We can use this query:

```
select B.title, A.author
from books as B, unnest(B.author_array) as A(author);
```

## Array and Multiset Types in SQL(cont...)

- When unnesting an array, the previous query loses information about the ordering of elements in the array. The **unnest with ordinality clause can be used** to get this information.

```
select title, A.author, A.position
from books as B,
            unnest(B.author_array) with ordinality as A(author, position);
```

## Array and Multiset Types in SQL(cont…)

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

| *title* | *author* | *pub_name* | *pub_branch* | *keyword* |
|---|---|---|---|---|
| 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 |

*Fig:- flat books: result of unnesting attributes author array and keyword set of relation books.*

```
select title, A.author, publisher.name as pub_name, publisher.branch
        as pub_branch, K.keyword
from books as B, unnest(B.author_array) as A(author),
        unnest (B.keyword_set) as K(keyword);
```

## Array and Multiset Types in SQL(cont…)

- The reverse process of transforming a 1NF relation into a nested relation is called **nesting. Nesting can be carried out by an extension of grouping in** SQL.
- The **collect function returns the** multiset of values, so instead of creating a single value, we can create a nested relation.

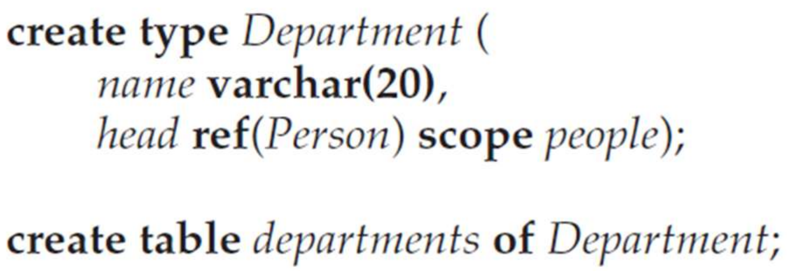
| *title* | *author* | *publisher* | *keyword_set* |
|---|---|---|---|
| | | (*pub_name,pub_branch*) | |
| Compilers | Smith | (McGraw-Hill, New York) | {parsing, analysis} |
| Compilers | Jones | (McGraw-Hill, New York) | {parsing, analysis} |
| Networks | Jones | (Oxford, London) | {Internet, Web} |
| Networks | Frick | (Oxford, London) | {Internet, Web} |

```
select title, author, Publisher(pub_name, pub_branch) as publisher,
       collect(keyword) as keyword_set
from flat_books
group by title, author, publisher;
```

Fig:- A partially nested version of the *flat books relation.*

## Object-Identity and Reference Types in SQL

- Object-oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type.
- For example, in SQL we can define a type *Department with a field name and a field head that is a reference* to the type *Person, and a table departments of type Department, as follows:*



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

create table departments of Department;
```

## Object-Identity and Reference Types in SQL

- 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);
```

- The referenced table must have an attribute that stores the identifier of the tuple. We declare this attribute, called the **self-referential attribute, by adding a ref is clause to the create table statement:**

```
create table people of Person
ref is person_id system generated;
```

## Object-Identity and Reference Types in SQL

- In order to initialize a reference attribute, we need to get the identifier of the tuple that is to be referenced. We can get the identifier value of a tuple by means of a query.

```
insert into departments
    values ('CS', null);

update departments
    set head = (select p.person_id
                from people as p
                where name = 'John')
where name = 'CS';
```

## Object-Identity and Reference Types in SQL

- An alternative to system-generated identifiers is to allow users to generate identifiers. The type of **the self-referential attribute** 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:**

```
create type Person
      (name varchar(20),
       address varchar(20))
      ref using varchar(20);

create table people of Person
      ref is person_id user generated;
```

- When inserting a tuple in *people, we must then provide a value for the identifier:*

```
insert into people (person_id, name, address) values
      ('01284567', 'John', '23 Coyote Run');
```

## Object-Identity and Reference Types in SQL

- No other tuple for *people or its supertables or subtables can have the same* identifier. We can then use the identifier value when inserting a tuple into *departments,* without the need for a separate query to retrieve the identifier:

```
insert into departments
    values ('CS', '01284567');
```

- It is even possible to use an existing primary-key value as the identifier, by including the **ref from clause in the type definition:**

```
create type Person
    (name varchar(20) primary key,
     address varchar(20))
    ref from(name);

create table people of Person
    ref is person_id derived;
```

## Object-Identity and Reference Types in SQL

- References are dereferenced in SQL:1999 by the –> *symbol. Consider the departments table defined earlier. We can use this query to 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.***
- We can use the operation **deref to return the tuple pointed to by a reference,** and then access its attributes, as shown below:

```
select deref(head).name
from departments;
```

## Implementing O-R Features

- Object-relational database systems are basically extensions of existing relational database systems. Changes are clearly required at many levels of the database system.
- However, to minimize changes to the storage-system code the complex data types supported by object-relational systems can be translated to the simpler type system of relational databases.
- To understand how to do this translation, we need look only at how some features of the E-R model are translated into relations.
- For instance, multivalued attributes in the E-R model correspond to multiset-valued attributes in the object relational model.

## Implementing O-R Features

- Composite attributes roughly correspond to structured types.
- ISA hierarchies in the E-R model correspond to table inheritance in the object relational model.
- To translate object-relational data to relational data at the storage level there are two ways
  - Each table stores the primary key and the attributes that are defined locally. Inherited attributes do not need to be stored, and can be derived by means of a join with the supertable, based on the primary key.
  - Each table stores all inherited and locally defined attributes. When a tuple is inserted, it is stored only in the table in which it is inserted, and its presence is inferred in each of the supertables. Access to all attributes of a tuple is faster, since a join is not required.

# Persistent Programming Languages

- Database languages differ from traditional programming languages in that they directly manipulate data that are **persistent**—that is, data that continue to exist even after the program that created it has terminated.
- A **persistent programming language is a programming language extended** with constructs to handle persistent data. Persistent programming languages can be distinguished from languages with embedded SQL in at least two ways:
  - With an embedded language, **the type system of the host language** usually **differs** from the type system of the **data-manipulation language.**
  - The **programmer** using an embedded query language is **responsible** for writing **explicit code to fetch data from the database into memory**.
- In contrast, in a **persistent programming language**, the programmer can manipulate persistent data without writing code explicitly to fetch it into memory or store it back to disk.

## Persistent Programming Languages(cont…)

- **Persistence of Objects**
- Object-oriented programming languages already have a concept of objects, a type system to define object types, and constructs to create objects.
- These objects are ***transient***—*they vanish when the program terminates, just as* variables in a Java or C program vanish when the program terminates.
- If we wish to turn such a language into a database programming language, the first step is to provide a way to make objects persistent.
- Several approaches have been proposed.

## Persistent Programming Languages(cont…)

- **Persistence by class**
- The simplest, but least convenient, way is to declare that a class is persistent.
- **All objects of the class are then persistent** objects **by default**. Objects of nonpersistent classes are all transient.
- This approach is not flexible, since it is often useful to have both transient and persistent objects in a single class.

## Persistent Programming Languages(cont...)

- **Persistence by creation**

  In this approach, new syntax is introduced to create persistent objects, by extending the syntax for creating transient objects. Thus, an object is either persistent or transient, depending on how it was created. Several object-oriented database systems follow this approach.

- **Persistence by marking**

  A variant of the preceding approach is to mark objects as persistent after they are created. All objects are created as transient objects, but, if an object is to persist beyond the execution of the program, it must be marked explicitly as persistent before the program terminates

## Persistent Programming Languages(cont...)

- **Persistence by reachability**

  One or more objects are explicitly declared as (root) persistent objects. All other objects are persistent if (and only if) they are reachable from the root object through a sequence of one or more references.

  A benefit of this scheme is that it is easy to make entire data structures persistent by merely declaring the root of such structures as persistent.

## Persistent Programming Languages(cont…)

- **Object Identity and Pointers**
- In an object-oriented programming language when an object is created, the system returns a transient object identifier.
- Transient object identifiers are valid only when the program that created them is executing; after that program terminates, the objects are deleted, and the identifier is meaningless.
- When a persistent object is created, it is assigned a persistent object identifier.
- The **object identity** has an interesting relationship to **pointers** in programming languages.
- A simple way to achieve built-in identity is through pointers to physical locations in storage.

## Persistent Programming Languages(cont…)

- There are several degrees of permanence of identity:
- **Intraprocedure:-** Identity persists only during the execution of a single procedure. Examples of intraprogram identity are local variables within procedures.
- **Intraprogram:-** Identity persists only during the execution of a single program or query. Examples of intraprogram identity are global variables in programming languages. Main-memory or virtual-memory pointers offer only intraprogram identity.
- **Interprogram:-** Identity persists from one program execution to another. Pointers to file-system data on disk offer interprogram identity, but they may change if the way data is stored in the file system is changed.

## Persistent Programming Languages(cont…)

- **Persistent:-** Identity persists not only among program executions, but also among structural reorganizations of the data. It is the persistent form of identity that is required for object-oriented systems.
- In persistent extensions of languages such as C++, object identifiers for persistent objects are implemented as "**persistent pointers**." A *persistent pointer is* a type of pointer that, unlike in-memory pointers, remains valid even after the end of a program execution.

## Persistent Programming Languages(cont…)

- **Storage and Access of Persistent Objects**
- The data part of an object has to be stored individually for each object.
- Logically, the code that implements methods of a class should be stored in the database as part of the database schema, along with the type definitions of the classes.
- Many implementations simply store the code in files outside the database, to avoid having to integrate system software such as compilers with the database system.
- There are several ways to find objects in the database. One way is to give names to objects, just as we give names to files.
- This approach works for a relatively small number of objects, but does not scale to millions of objects.

## Persistent Programming Languages(cont….)

- A second way is to expose object identifiers or persistent pointers to the objects, which can be stored externally.
- A third way is to store collections of objects, and to allow programs to iterate over the collections to find required objects. Collections of objects can themselves be modeled as objects of a *collection type.*
- Most object-oriented database systems support all three ways of accessing persistent objects. They give identifiers to all objects.

## Object-Relational Mapping

- **Object-relational mapping** systems provide a third approach to integration of object-oriented programming languages and databases.
- Object-relational mapping systems are built on top of a traditional relational database, and allow a programmer to define a mapping between tuples in database relations and objects in the programming language.
- An object, or a set of objects, can be retrieved based on a selection condition on its attributes; relevant data are retrieved from the underlying database based on the selection conditions.
- The program can optionally update such objects, create new objects, or specify that an object is to be deleted, and then issue a save command.

## Object-Relational Mapping(cont….)

- The primary goal of object-relational mapping systems is to ease the job of programmers who build applications, by providing them an object-model, while retaining the benefits of using a robust relational database underneath.
- Object-relational mapping systems also provide query languages that allow programmers to write queries directly on the object model; such queries are translated into SQL queries on the underlying relational database, and result objects created from the SQL query results.
- On the negative side, object-relational mapping systems can suffer from significant overheads for bulk database updates, and may provide only limited querying capabilities.

## Object-Oriented versus Object-Relational

- Object-relational databases built on top of the relation model and object-oriented databases built around persistent programming languages, and object-relational mapping systems.
- Each of these approaches targets a different market.
- Object-relational systems aim at making data modeling and querying easier by using complex data types. Typical applications include storage and querying of complex data, including multimedia data.
- SQL, however, imposes a significant performance penalty for certain kinds of applications that run primarily in main memory, and that perform a large number of accesses to the database.
- Persistent programming languages target such applications that have high performance requirements. They provide low-overhead access to persistent data.