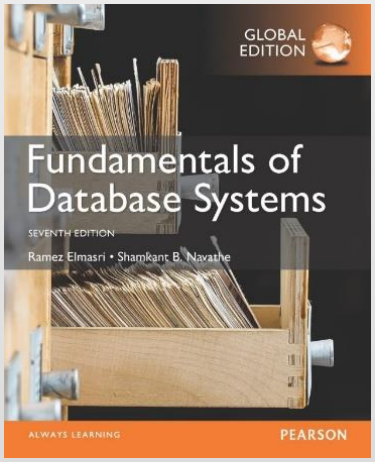
CS 703
Advanced Database Systems

Chapter 12: Object and Object-Relational Databases Part I



Introduction

- Weak Points of the Relational Model:

1. Efficiency
   - information retrieved by addressing multiple tables (join operation is very slow)
2. Data semantics
   - the relational model lacks semantics
     - cannot distinguish between different types of relationships (association, aggregation, specialization)
     - a column can be either attribute or relationship.
3. Model extension
   - relations cannot be used as built-in data types (1-NF prevents nested relations)

(a)
EMP_PROJ

		Projs	
Ssn	Ename	Pnumber	Hours

(b)
EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

Introduction

- Weak Points of the Relational Model:

4. Program interface and impendence mismatch problem

  - differences between the database model and the programming language model.

  - mismatch in the data types supported by the two systems. More on impedance mismatch (Section 10.1.2)

    - Binding for each host programming language that specifies for each attribute type the compatible programming language types. A different binding is needed for each programming language because different languages have different data types.
    - Another problem occurs because the results of most queries are sets or multisets of tuples (rows), and each tuple is formed of a sequence of attribute values. In the program, it is often necessary to access the individual data values within individual tuples for printing or processing. Hence, a binding is needed to map the query result data structure, which is a table, to an appropriate data structure in the programming language. A mechanism is needed to loop over the tuples in a query result in order to access a single tuple at a time and to extract individual values from the tuple. The extracted attribute values are typically copied to appropriate program variables for further processing by the program.

    Or

    - Special database programming language is designed that uses the same data model and data types as the database Model such as PL\SQL or SQL\PSM.

The object data model is quite similar to the data model of the Java programming language, so the impedance mismatch is greatly reduced when Java is used as the host language for accessing a Java-compatible object database.

Introduction

- **Object databases (ODB)**
  - Reason for the creation of object-oriented databases is the vast increase in the use of object-oriented programming languages for developing software applications.
  - Meet some of the needs of more complex applications (CAD\CAM, biological and other sciences, telecommunications, geographic information systems, and multimedia).
  - Object-oriented databases have adopted many of the concepts that were developed originally for object-oriented programming languages.
  - Specify:
    - **Structure of complex objects**
    - **Operations that can be applied to these objects**
  - These include object identity, object structure and type constructors, encapsulation of operations, and the definition of methods as part of class declarations, mechanisms for storing objects in a database by making them persistent, and type and class hierarchies and inheritance.
- **Relational DBMS (RDBMS)** vendors have also recognized the need for incorporating features that were proposed for object databases, and newer versions of relational systems have incorporated many of these features.
  - This has led to database systems that are characterized as object-relational or ORDBMs.

Overview of Object Database Concepts

Introduction to Object-Oriented Concepts and Features

- Origins in OOPL

- An object typically has two components: state (value) and behavior (operations). It can have a complex data structure as well as specific operations defined by the programmer.

- Objects in an OOPL exist only during program execution; therefore, they are called transient objects. An OO database can extend the existence of objects so that they are stored permanently in a database, and hence the objects become persistent objects that exist beyond program termination and can be retrieved later and shared by other programs.

- The internal structure of an object in OOPLs includes the specification of instance variables, which hold the values that define the internal state of the object. An instance variable is similar to the concept of an attribute in the relational model, except that instance variables may be encapsulated within the object and thus are not necessarily visible to external users. Instance variables may also be of arbitrarily complex data types.

- Object-oriented systems allow definition of the operations or functions (behavior) that can be applied to objects of a particular type.

  – **the object behavior** …

    - methods (operations) that can be executed to

      – create/destroy an object

      – update an object state

      – retrieve object state

      – compute new values based on object state

    - the names and parameters of methods

      – define the object interface

Introduction to Object-Oriented Concepts and Features

- Inheritance
  - Permits specification of new types or classes that inherit much of their structure and/or operations from previously defined types or classes. This makes it easier to develop the data types of a system incrementally and to reuse existing type definitions when creating new types of objects.

- Operator overloading (polymorphism)
  - Operation's ability to be applied to different types of objects
  - Operation name may refer to several distinct implementations
  - For example, an operation to calculate the area of a geometric object may differ in its method (implementation), depending on whether the object is of type triangle, circle, or rectangle.

- The concept of encapsulation is one of the main characteristics of OO languages and systems. It is also related to the concepts of abstract data types and information hiding in programming languages.
  - Define **behavior** of a class of object based on operations that can be externally applied
  - External users only aware of interface of the operations
  - Can divide structure of object into visible and hidden attributes

Overview of Object Database Concepts

--The **object identifier (OID)**

- Is a unique system-wide identifier. Every object must have an object identifier.
- Object has Unique identity
  - Implemented via a unique, system-generated object identifier (OID)
  - Not visible to the user but is used internally by the system to identify each object uniquely and to create and manage interobject references.
  - Immutable (unchangeable for the same object and is not assigned to other objects).
  - OIDs does not depend on attribute values.

RDB: each relation must have a primary key attribute whose value identifies each tuple uniquely. If the value of the primary key is changed, the tuple will have a new identity,

- Some early OO data models required that everything—from a simple value to a complex object—was represented as an object; hence, every basic value, such as an integer, string, or Boolean value, has an OID. Although useful as a theoretical model, this is not very practical, since it leads to the generation of too many OIDs.
- Hence, most ODBs allow for the representation of both objects and literals (or values). Every object must have an immutable OID, whereas a literal value has no OID and its value just stands for itself. Thus, a literal value is typically stored within an object and cannot be referenced from other objects.
- In many systems, complex structured literal values can also be created without having a corresponding OID if needed.

Objects versus Literals

- Most OO database systems allow for the representation of both objects and literals (simple or complex values)
- Objects and literals are the basic building blocks of the object model. The main difference between the two is that an object has both an object identifier and a **state** (or current value), whereas a literal has a value (state) but *no object identifier.*
- In either case, the value can have a complex structure.

Complex Type Structures for Objects and Literals

- Structure of arbitrary complexity
  - Contain all necessary information that describes object or literal

RDB: Information about a complex data structure is often scattered over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation.

- A complex type may be constructed from other types by nesting of type constructors. The three most basic constructors are atom, struct (or tuple), and collection.

1. One type constructor has been called the **atom** constructor. This includes the basic built-in data types of the object model, which are similar to the basic types in many programming languages: integers, strings, floating-point numbers, Booleans, and so on. These basic data types are called **single valued** or **atomic** types, since each value of the type is considered an atomic (indivisible) single value.
2. A second type constructor is referred to as the **struct** (or **tuple**) constructor. This can create standard structured types, such as the tuples (record types) in the basic relational model. A structured type is made up of several components and is also sometimes referred to as a *compound* or *composite* type.

   For example, two different structured types that can be created are:

   struct Name<FirstName: string, MiddleInitial: char, LastName: string>

   struct CollegeDegree<Major: string, Degree: string, Year: date>.

RDB: the type constructors *atom* and *struct* are the only ones available in the original (basic) relational model.
.

Complex Type Structures for Objects and Literals

3. To create complex nested type structures in the object model, the *collection* type constructors are needed.

Collection (or multivalued) type constructors include the set(T), list(T), bag(T), array(T), and dictionary(K,T) type constructors. These allow part of an object or literal value to include a collection of other objects or values when needed. These constructors are also considered to be type generators because many different types can be created. For example, set(string), set(integer), and set(Employee) are three different types that can be created from the set type constructor. All the elements in a particular collection value must be of the same type. For example, all values in a collection of type set(string) must be string values.

- Collection types:
  - set constructor will create objects or literals that are a set of distinct elements {i1, i2,… ,in}, all of the same type.
  - The bag constructor (also called a multiset) is similar to a set except that the elements in a bag need not be distinct.
  - The list constructor will create an ordered list [i1, i2, … , in] of OIDs or values of the same type. A list is similar to a bag except that the elements in a list are ordered, and hence we can refer to the first, second, or jth element.
  - The array constructor creates a single-dimensional array of elements of the same type. The main difference between array and list is that a list can have an arbitrary number of elements whereas an array typically has a maximum size.
  - Finally, the dictionary constructor creates a collection of key-value pairs (K, V), where the value of a key K can be used to retrieve the corresponding value V.

Encapsulation of Object Attributes

RDB: Attributes are visible to users and applications according to the predefined external view.

- Divide the structure of an object into visible and hidden attributes (instance variables).
- Visible attributes can be seen by and are directly accessible to the database users and programmers via the query language.
- The hidden attributes of an object are completely encapsulated and can be accessed only through predefined operations.
- Most ODMSs employ high-level query languages for accessing visible attributes
- The term class is often used to refer to a type definition, along with the definitions of the operations for that type.

Encapsulation of Operations

RDB: Operations are generic and visible to users and applications. The operations for selecting, inserting, deleting, and modifying tuples are generic and may be applied to *any relation* in the database. The relation and its attributes are visible to users and to external programs that access the relation by using these operations.

- The concept of encapsulation is applied to database objects in ODBs by defining the behavior of a type of object based on the operations that can be Externally applied to objects of that type.

Typical operations include:

- **Constructor** operation
  - Used to create a new object
- **Destructor** operation
  - Used to destroy (delete) an object
- **Modifier** operations
  - Modify the state of an object
- **Retrieve** operation
  - Retrieve parts of the object state
- *Dot notation* to apply operations to object
  - An operation is typically applied to an object by using the **dot notation**. For example, if d is a reference to a DEPARTMENT object, we can invoke an operation such as no_of_emps by writing d.no_of_emps. Similarly, by writing d.destroy_dept, the object referenced by d is destroyed (deleted).
- The external users of the object are only made aware of the interface of the operations, which defines the name and arguments (parameters) of each operation. The implementation is hidden from the external users.

Persistence of Objects

- **Transient objects:** Exist in executing program. Disappear once program terminates
- **Persistent objects:** Stored in database, persist after program termination

- **The typical mechanisms for making an object persistent are naming and reachability.**
  - **Naming mechanism:** object assigned a unique name in object base, user finds object by its name through which users and applications can start their database access
  - **Reachability:** object referenced from other persistent objects, object located through references**.** An object B is said to be reachable from an object A if a sequence of references in the database lead from object A to object B.

- In the OO approach, a class declaration of EMPLOYEE specifies only the type and operations for a class of objects. The user must separately define a persistent object of type set(EMPLOYEE) whose value is the collection of references (OIDs) to all persistent EMPLOYEE objects, if this is desired

- The ODMG ODL standard gives the schema designer the option of naming an extent as part of class definition.

RDB: all objects are assumed to be persistent. Hence, when a table such as EMPLOYEE is created in a relational database, it represents both the type declaration for EMPLOYEE and a persistent set of all EMPLOYEE records (tuples).

Type (Class) Hierarchies and Inheritance (cont'd.)

- Inheritance
  - Definition of new types based on other predefined types
  - Leads to **type** (or **class**) **hierarchy**
- **Subtype**
  - Useful when creating a new type that is similar but not identical to an already defined type
  - Subtype inherits functions
  - Additional (local or specific) functions in subtype

Other Object-Oriented Concepts

- **Polymorphism** of operations
  - Also known as **operator overloading**
  - Allows same operator name or symbol to be bound to two or more different implementations
  - Type of objects determines which operator is applied
- **Multiple inheritance**
  - Subtype inherits functions (attributes and operations) of more than one supertype
  - Example: subtype ENGINEERING_MANAGER that is a subtype of both MANAGER and ENGINEER.

Summary of Object Database Concepts

- **Object identity.** Objects have unique identities that are independent of their attribute values and are generated by the ODB system.
- **Type constructors.** Complex object structures can be constructed by applying in a nested manner a set of basic type generators/constructors, such as tuple, set, list, array, and bag.
- **Encapsulation of operations.** Both the object structure and the operations that can be applied to individual objects are included in the class/type definitions.
- **Programming language compatibility.** Both persistent and transient objects are handled seamlessly. Objects are made persistent by being reachable from

  a persistent collection (extent) or by explicit naming (assigning a unique name by which the object can be referenced/retrieved).
- **Type hierarchies and inheritance.** Object types can be specified by using a type hierarchy, which allows the inheritance of both attributes and methods (operations) of previously defined types. Multiple inheritance is allowed in some models.
- **Extents.** All persistent objects of a particular class/type C can be stored in an extent, which is a named persistent object of type set(C). Extents corresponding to a type hierarchy have set/subset constraints enforced on their collections of persistent objects.
- **Polymorphism and operator overloading.** Operations and method names can be overloaded to apply to different object types with different implementations.

Object-Relational Features: Object DB Extensions to SQL

Object-Relational Features: Object DB Extensions to SQL

- **Type constructors (generators)**
  - Specify complex types using user defined types UDT such as row type, array types, set, list and bag.
- Mechanism for specifying **object identity** using *reference type* operator**.**
- **Encapsulation of operations**
  - Provided through user-defined types (UDTs) that may include operations as part of their declaration. In addition, the concept of user-defined routines (UDRs) allows the definition of general methods (operations).
- **Inheritance** mechanisms
  - Provided using keyword UNDER

User-Defined Types (UDTs) and Complex Structures for Objects

- **UDT** syntax:
  - `CREATE TYPE <type name> AS (<component declarations>);`
  - Can be used to create a complex type for an attribute
- Array type – to specify collections
  - Reference array elements using []

```
CREATE TYPE USA_ADDR_TYPE AS (
    STREET_ADDR     ROW (   NUMBER          VARCHAR (5),
                            STREET_NAME     VARCHAR (25),
                            APT_NO          VARCHAR (5),
                            SUITE_NO        VARCHAR (5) ),
    CITY            VARCHAR (25),
    ZIP             VARCHAR (10)
    );
```

```
(a) CREATE TYPE STREET_ADDR_TYPE AS (
        NUMBER          VARCHAR (5),
        STREET          NAME VARCHAR (25),
        APT_NO          VARCHAR (5),
        SUITE_NO        VARCHAR (5)
    );
    CREATE TYPE USA_ADDR_TYPE AS (
        STREET_ADDR     STREET_ADDR_TYPE,
        CITY            VARCHAR (25),
        ZIP             VARCHAR (10)
    );
    CREATE TYPE USA_PHONE_TYPE AS (
        PHONE_TYPE      VARCHAR (5),
        AREA_CODE       CHAR (3),
        PHONE_NUM       CHAR (7)
    );
```

Figure 12.4a Illustrating some of the object features of SQL. Using UDTs as types for attributes such as Address and Phone.

Object Identifiers Using Reference Types

- **Reference type**
  - Create unique object identifiers (OIDs)
  - Can specify system-generated object identifiers
  - Alternatively can use primary key as OID as in traditional relational model
  - Examples:
    - `REF IS SYSTEM GENERATED`
    - `REF IS <OID_ATTRIBUTE> <VALUE_GENERATION_METHOD> ;`

SYSTEM GENERATED
Or
DERIVED

```
(b) CREATE TYPE PERSON_TYPE AS (
        NAME            VARCHAR (35),
        SEX             CHAR,
        BIRTH_DATE      DATE,
        PHONES          USA_PHONE_TYPE ARRAY [4],
        ADDR            USA_ADDR_TYPE
    INSTANTIABLE
    NOT FINAL
    REF IS SYSTEM GENERATED
    INSTANCE METHOD AGE() RETURNS INTEGER;
    CREATE INSTANCE METHOD AGE() RETURNS INTEGER
        FOR PERSON_TYPE
        BEGIN
            RETURN /* CODE TO CALCULATE A PERSON'S AGE FROM
                      TODAY'S DATE AND SELF.BIRTH_DATE */
        END;
    );
```

indicates that whenever a new PERSON_TYPE object is created, the system will assign it a unique system-generated identifier.

Figure 12.4b Illustrating some of the object features of SQL. Specifying UDT for PERSON_TYPE.

Creating Tables Based on the UDTs

- **INSTANTIABLE**
  - Specify that UDT is instantiable
  - The user can then create one or more tables based on the UDT
  - If keyword INSTANTIABLE is left out, can use UDT only as attribute data type – not as a basis for a table of objects

```
(b) CREATE TYPE PERSON_TYPE AS (
        NAME            VARCHAR (35),
        SEX             CHAR,
        BIRTH_DATE      DATE,
        PHONES          USA_PHONE_TYPE ARRAY [4],
        ADDR            USA_ADDR_TYPE
    INSTANTIABLE
    NOT FINAL
    REF IS SYSTEM GENERATED
    INSTANCE METHOD AGE() RETURNS INTEGER;
    CREATE INSTANCE METHOD AGE() RETURNS INTEGER
        FOR PERSON_TYPE
        BEGIN
            RETURN /* CODE TO CALCULATE A PERSON'S AGE FROM
                      TODAY'S DATE AND SELF.BIRTH_DATE */
        END;
    );
```

Figure 12.4b Illustrating some of the object features of SQL. Specifying UDT for PERSON_TYPE.

Encapsulation of Operations

- User-defined type
  - Specify methods (or operations) in addition to the attributes

```
INSTANCE METHOD <NAME> (<ARGUMENT_LIST>) RETURNS
<RETURN_TYPE>;
```

```
(b)  CREATE TYPE PERSON_TYPE AS (
         NAME            VARCHAR (35),
         SEX             CHAR,
         BIRTH_DATE      DATE,
         PHONES          USA_PHONE_TYPE ARRAY [4],
         ADDR            USA_ADDR_TYPE
     INSTANTIABLE
     NOT FINAL
     REF IS SYSTEM GENERATED
     INSTANCE METHOD AGE() RETURNS INTEGER;
     CREATE INSTANCE METHOD AGE() RETURNS INTEGER
         FOR PERSON_TYPE
         BEGIN
             RETURN /* CODE TO CALCULATE A PERSON'S AGE FROM
                       TODAY'S DATE AND SELF.BIRTH_DATE */
         END;
     );
```

Figure 12.4b Illustrating some of the object features of SQL. Specifying UDT for PERSON_TYPE.

Specifying Type Inheritance

- NOT FINAL:
  - The keyword NOT FINAL indicates that subtypes can be created for that type
- UNDER
  - The keyword UNDER is used to create a subtype
- Type inheritance rules:
  - All attributes/operations are inherited
  - Order of supertypes in UNDER clause determines inheritance hierarchy
  - Instance (object) of a subtype can be used in every context in which a supertype instance used
  - Subtype can redefine any function defined in supertype

Specifying Type Inheritance

```
(b)  CREATE TYPE PERSON_TYPE AS (
         NAME            VARCHAR (35),
         SEX             CHAR,
         BIRTH_DATE      DATE,
         PHONES          USA_PHONE_TYPE ARRAY [4],
         ADDR            USA_ADDR_TYPE
     INSTANTIABLE
     NOT FINAL
     REF IS SYSTEM GENERATED
     INSTANCE METHOD AGE() RETURNS INTEGER;
     CREATE INSTANCE METHOD AGE() RETURNS INTEGER
         FOR PERSON_TYPE
         BEGIN
             RETURN /* CODE TO CALCULATE A PERSON'S AGE FROM
                       TODAY'S DATE AND SELF.BIRTH_DATE */
         END;
     );
```

Figure 12.4b Illustrating some of the object features of SQL. Specifying UDT for PERSON_TYPE.

```
(c)  CREATE TYPE GRADE_TYPE AS (
         COURSENO        CHAR (8),
         SEMESTER        VARCHAR (8),
         YEAR            CHAR (4),
         GRADE           CHAR
     );
     CREATE TYPE STUDENT_TYPE UNDER PERSON_TYPE AS (
         MAJOR_CODE      CHAR (4),
         STUDENT_ID      CHAR (12),
         DEGREE          VARCHAR (5),
         TRANSCRIPT      GRADE_TYPE ARRAY [100]
     INSTANTIABLE
     NOT FINAL
     INSTANCE METHOD GPA( ) RETURNS FLOAT;
     CREATE INSTANCE METHOD GPA( ) RETURNS FLOAT
         FOR STUDENT_TYPE
         BEGIN
             RETURN /* CODE TO CALCULATE A STUDENT'S GPA FROM
                       SELF.TRANSCRIPT */
         END;
     );
     CREATE TYPE EMPLOYEE_TYPE UNDER PERSON_TYPE AS (
         JOB_CODE        CHAR (4),
         SALARY          FLOAT,
         SSN             CHAR (11)
     INSTANTIABLE
     NOT FINAL
     );
     CREATE TYPE MANAGER_TYPE UNDER EMPLOYEE_TYPE AS (
          DEPT_MANAGED CHAR (20)
     INSTANTIABLE
     );
```

Figure 12.4c Illustrating some of the object features of SQL. Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two subtypes of PERSON_TYPE.

Creating Tables based on UDT

- UDT must be INSTANTIABLE
- One or more tables can be created
- Table inheritance:
  - UNDER keyword can also be used to specify supertable/subtable inheritance
  - Objects in subtable must be a **subset of** the objects in the supertable

- Here, a new record that is inserted into a subtable, say the MANAGER table, is also inserted into its supertables EMPLOYEE and PERSON.
- Notice that when a record is inserted in MANAGER, we must provide values for all its inherited attributes. INSERT, DELETE, and UPDATE operations are appropriately propagated.
- The rule is that a tuple in a sub-table must also exist in its super-table to enforce the set/subset constraint on the objects.

```
(d) CREATE TABLE PERSON OF PERSON_TYPE
        REF IS PERSON_ID SYSTEM GENERATED;
    CREATE TABLE EMPLOYEE OF EMPLOYEE_TYPE
        UNDER PERSON;
    CREATE TABLE MANAGER OF MANAGER_TYPE
        UNDER EMPLOYEE;
    CREATE TABLE STUDENT OF STUDENT_TYPE
        UNDER PERSON;
```

Figure 12.4d Illustrating some of the object features of SQL. Creating tables based on some of the UDTs, and illustrating table inheritance.

A component attribute of one tuple may be a reference (specified using the keyword
REF) to a tuple of another (or possibly the same) table.

The keyword SCOPE specifies the name of the table whose tuples can be referenced by the reference attribute.

Notice that this is similar to a foreign key, except that the system-generated OID value is used rather than the primary key value.

```
(e) CREATE TYPE COMPANY_TYPE AS (
        COMP_NAME     VARCHAR (20),
        LOCATION      VARCHAR (20));
    CREATE TYPE EMPLOYMENT_TYPE AS (
        Employee REF (EMPLOYEE_TYPE) SCOPE (EMPLOYEE),
        Company REF (COMPANY_TYPE) SCOPE (COMPANY) );
    CREATE TABLE COMPANY OF COMPANY_TYPE (
        REF IS COMP_ID SYSTEM GENERATED,
        PRIMARY KEY (COMP_NAME) );
    CREATE TABLE EMPLOYMENT OF EMPLOYMENT_TYPE;
```

Figure 12.4(e) Specifying relationships using REF and SCOPE.

Summary of SQL Object Extensions

- UDT to specify complex types
  - INSTANTIABLE specifies if UDT can be used to create tables; NOT FINAL specifies if UDT can be inherited by a subtype
- REF for specifying **object identity** and inter-object references
- Encapsulation of operations in UDT
- Keyword `UNDER` to specify type inheritance and table inheritance