Chapter 12
Object and Object Relational Databases

- Relational Data Model
- Object data model (OODBs)
- Object-relational data models

Traditional data models
- network
- hierarchical
- relational

They lack complex applications, engg design, (CAD/CAM), biological, geographical information systems, multi-media, complex stored structures

ODBs can do the above modern database applications
- OO programming languages helped to spur ODBs
- ODBs seamlessly integrate with applications
- Relational databases also incorporated object concept (Won Kim) resulting in object-relational DBs

(Orion, OpenOODB, Ode, Encore, Ontos, Objectivity, Versant, ObjectStore, etc..)

- Standards were developed ODMG 3.0 (consortium of vendors)

Object-oriented DBMs adopted OO features of programming languages:
- object identity
- object structure
- type constructor
- encapsulation of operations
- Class hierarchy
- Inheritance

These features are incorporated into latest SQL standards leading to ORDBMSs

SQL-1999
SQL-2008

1. Object Database Concepts

Originated from OO programming languages, OO concepts can be applied to many areas: hardware, software and systems.

OOPL roots SIMULA in 1960s

Smalltalk in 1970s

C++/Java/C# and more

Object has state and behavior

Objects exist during execution (instance), called transient objects

OO database extends this concept to make it persistent objects, needs indexing, object sharing, concurrency and recovery

- Internal structure of an object includes instance variables, internal state of the object, not visible from outside. They can be complex data types
- Operations may be required to be predefined in objects
- Operations can be involved by passing message to an object, which includes operation name and parameters
- OO systems provide class hierarchies, inheritance, can help do incremental development and reuse of existing objects
- Initial OODBs used relationships between objects, later used methods to represent relationships
- Operator overloading applies to different types of objects, also called operator polymorphism; ex: area of an object may differ in its method implementation, depending on triangle, circle, and rectangle. This may require, late binding at run time for choosing appropriate method

**Object Identity:** objects can be easily identified and operated on by using a unique identity (OID). Not visible from outside. OID is immutable, i.e. should not change (real world representation).
Generate and preserve OIDs in ODB. Also, OID must be used once, even if it is deleted from DB. It is inappropriate to use a physical address for OID, however some DBs use it. Use an indirect address to change it dynamically. Simple value should be represented as an object (int, string, etc..); this is not practical as it creates too many objects; they allow objects and literals (or values)

**Complex Data Structures:**
Complex types may be constructed from other types by nesting of type constructors. (1) atom (2) struct (tuple) (3) collection
Atom: a basic data type of a programming language, single valued or atomic
Struct (tuple): record type, compound or composite type
Name <FirstName: String, MiddleInitial: char, LastName: string>
CollegeDegree <Major:string, Year:date, Degree:string>
Collection: (multi-valued)
Set(T), list(T), bag(T), array(T), dictionary(K, T); also called type generators, many different types can be generated
Examples: set(string) all must be string values
  set(int)
  set(employee); all the elements in a particular collection value must be of the same type

-set constructor create an objet or literals that are a set of distinct elements (i1, i2, …, in); all of the same type

- bag constructor (called a multiset), the elements need to be distinct
- List: elements are ordered, refer to them as first, second, , jth,.. element
- Array: Single dimensional array of same type
- Dictionary: creates a collection of key-value pairs (k, v), value of a key k can be used to retrieve

Object Definition Language

Fig. 12-1

The attributes that refer to other objects, are basically OIDs that serve as reference to other objects to represent relationships among the objects. A binary relationship can be represented in one direction or it can have inverse reference.

Employee ➔ works for Department

Department -> has employees (is a set of OIDs)
Figure 12.1  Specifying the object types EMPLOYEE, DATE, and DEPARTMENT using type constructors.

```plaintext
define type EMPLOYEE
tuple (  Fname:  string;
            Minit :  char;
            Lname:  string;
            Ssn:  string;
            Birth_date:  DATE;
            Address:  string;
            Sex:  char;
            Salary:  float;
            Supervisor:  EMPLOYEE;
            Dept:  DEPARTMENT;
)
define type DATE
tuple (  Year:  integer;
            Month:  integer;
            Day:  integer; )
define type DEPARTMENT
tuple (  Dname:  string;
            Dnumber:  integer;
            Mgr:  tuple (  Manager:  EMPLOYEE;
                           Start_date:  DATE; )
            Locations:  set(string);
            Employees:  set(EMPLOYEE);
            Projects:  set(PROJECT); )
```
**Encapsulation**
Encapsulation, information hiding, abstract data types (OOP)

Traditional DBMS – not applicable, users should see it

Selecting, inserting, deleting, and modifying tuples are generic operation

Implementation is hidden from users, external users made aware of the interfaces

Interface is called signature

Implementation is called by the method

Instead of completely hiding, we have visible and hidden attributes (instance variables)

Visible attributes can be seen by and are directly accessible to DB users and programmers

Hidden attributes are encapsulated and accessed through pre-defined operations.

A class is referred to a type definition.

Fig. 12.2

Referring to attributes:

d.no_of_emps

d.destroy_dept
Figure 12.2  Adding operations to the definitions of EMPLOYEE and DEPARTMENT.

define class EMPLOYEE
  type tuple ( Fname: string;
               Minit: char;
               Lname: string;
               Ssn: string;
               Birth_date: DATE;
               Address: string;
               Sex: char;
               Salary: float;
               Supervisor: EMPLOYEE;
               Dept: DEPARTMENT; );
  operations age: integer;
               create_emp: EMPLOYEE;
               destroy_emp: boolean;
end EMPLOYEE;

define class DEPARTMENT
  type tuple ( Dname: string;
               Dnumber: integer;
               Mgr: tuple ( Manager: EMPLOYEE;
                             Start_date: DATE; );
               Locations: set (string);
               Employees: set (EMPLOYEE);
               Projects set(PROJECT); );
  operations no_of_emps: integer;
               create_dept: DEPARTMENT;
               destroy_dept: boolean;
               assign_emp(e: EMPLOYEE): boolean;
               (* adds an employee to the department *)
               remove_emp(e: EMPLOYEE): boolean;
               (* removes an employee from the department *)
end DEPARTMENT;
**Object Persistence**

ODBS is closely coupled with an OOPL. OOPL is used to code database program as well as application. All objects in the program are not persistent in the DB. There are transient and persistent objects. Mechanism to make them persistent is naming and reachability (Java, C++ languages support object persistence).

Fig. 12-3

Define class DEPARTMENT_SET

Persistent name ALL_DEPARTMENTS:DEPARTMENT_SET;

It is possible to define several persistent collections for the same class definition.
Figure 12.3  Creating persistent objects by naming and reachability.

```plaintext
define class DEPARTMENT_SET
    type set (DEPARTMENT);
    operations add_dept(d: DEPARTMENT): boolean;
        (* adds a department to the DEPARTMENT_SET object *)
    remove_dept(d: DEPARTMENT): boolean;
        (* removes a department from the DEPARTMENT_SET object *)
    create_dept_set: DEPARTMENT_SET;
    destroy_dept_set: boolean;
end Department_Set;
...
persistent name ALL_DEPARTMENTS: DEPARTMENT_SET;
    (* ALL_DEPARTMENTS is a persistent named object of type DEPARTMENT_SET *)
...
d:= create_dept;
    (* create a new DEPARTMENT object in the variable d *)
...
b:= ALL_DEPARTMENTS.add_dept(d);
    (* make d persistent by adding it to the persistent set ALL_DEPARTMENTS *)
```
**Type Hierarchies & Inheritance**

ODBs allow type hierarchies and inheritance. Attributes and operations are treated uniformly.

A type is defined by assigning it to a type name and then defining a number of attributes and operations; we call both of them functions.

A type has a type name and a list of visible functions.

TYPE_NAME:function, function, .......

Example:

PERSON: Name, Address, Birth_date, Age, SSN

(Age is a function)

A subtype inherits all the functions of a predefined type.

```
PERSON
  Name, Address,
  Birth_date, Age,
  SSN

EMPLOYEE
  Salary, Hire_date,
  Seniority

STUDENT
  Major, GPA
```
EMPLOYEE subtype_of PERSON: Salary, Hire_date, Seniority
STUDENT subtype_of PERSON: Major, GPA

GEOMETRY_OBJECT: Shape, Area, Reference_Point (Shape is an attribute with values: triangle, rectangle, circle, ....)
RECTANGLE subtype_of GEOMETRY_OBJECT: width, height
TRIANGLE subtype_of GEOMETRY_OBJECT: Side1, Side2, Angle
CIRCLE subtype_of GEOMETRY_OBJECT: radius

Area operation may be implemented by a different method for each subtype
Reference point may have different meaning for each subtype.

**Constraints on Extents**

Extent is defined to store the collection of persistent for each type or subtype
ROOT class or the OBJECT class, whose extent contains all the objects in the system
Every object in an extent that corresponds to a subtype, must also be a member of the extent that corresponds to its supertype (constraint)
An extent is a named persistent object, whose value is a persistent collection that holds a collection of objects of the same type that are stored permanently in the database. It is also possible to create transient collection (results of query)
**Other OO Concepts**

Polymorphism of operations (operator overloading)

Allows same operator or name to be bound to two or more implementations of the operator, depending on the type of objects.

Example from C++ notes

Example: add two integers, two floats, two sets, etc… + operator

GEOMETRY object: area function; write a general area function, and overload with specific area functions

The ODBS must select appropriate method for the area function based on the type of object.

If the object types are known at the compile time, then it is static binding.

In weak types, it is not known until run time (Smalltalk, Lisp, PHP)

Need late or dynamic binding.

**SKIP Multiple Inheritance**
2. Object Database Extensions to SQL

SQL Chamberlin and Boyce 1974

SQL Standards 1989 and 1992

SQL3 (Object database)

SQL/Object

The relational model with object extensions known as relational object model. Additional extensions of SQL to include XML known as SQL/Foundation in SQL 2008.

Object Features in SQL:

- Type constructors
- Object identity
- Encapsulation of operations
- Inheritance
  (1) User Defined Types (UDT) using CREATE TYPE and Complex Objects

**CREATE TYPE type_name AS (<component declarations>)**;

A UDT can be used as a type of an attribute or as a type of a table.

Complex types can be created by nested type structures similar to struct in C.
Figure 12.4a  Illustrating some of the object features of SQL. Using UDTs as types for attributes such as Address and Phone.

(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 12.4b  Illustrating some of the object features of SQL. Specifying UDT for PERSON_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 12.4c  Illustrating some of the object features of SQL. Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two subtypes of 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]``);`
Figure 12.4c (continued) Illustrating some of the object features of SQL. Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two subtypes of PERSON_TYPE.

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

(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 12.4e  Illustrating some of the object features of SQL. Specifying relationships using REF and SCOPE.

(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;
Fig. 12.4 (a) UDTs for using address and phone
Fig. 12.4 (b) UDT for PERSON_TYPE
Fig. 12.4 (c) STUDENT_TYPE and EMPLOYEE_TYPE
Fig. 12.4 (d) Illustrating Inheritance
Fig. 12.4 (e) Relationships using REF and SCOPE

If a UDT does not have any operations such as 12.4(a), then it is possible to use the concept of ROW_TYPE by directly create a structured attribute by using a keyword ROW.

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

Array types can be referenced PHONES[1] for first location.  
A built-in function CARDINALITY can return the current number of elements in an array or any other collection type  
Ex: PHONES[CARDINALITY(PHONES) refer to the last element  
ADDR.CITY refers to CITY component of ADDR attribute (12.4(b))
OIDs can be generated by the system, or use the traditional keys of the relational data model

REF IS <OID_ATTRIBUE><VALUE_GENERATION_METHOD>;
REF IS SYSTEM GENERATED;
REF IS <PERSON_ID> SYSTE GENERATED;

Creating Tables Based on the UDTs:
Fig. 12-2(d): one or more tables may be created if they are INSTANTIABLE

Encapsulation of Operations:
UDTs can specify attributes and methods
CREATE TYPE <TYPE_NAME> ( 
   <LIST OF COMPONENT ATTRIBUTES AND THEIR TYPES>
   <DECLARATION OF FUNCTION METHODS>
);

Age() in 12.b. calculates the age of the individual object of the type PERSON_TYPE

In general, a UDT can have a number of user-defined functions associated with it.
INSTANCE METHOD
<NAME>(<ARGUMENT_LIST>)RETURNS<RETURN_TYPE>

Internal and external functions (external functions in host language)
PUBLIC (visible at the UDT interface)
PRIVATE (not visible at the UDT interface)
PROTECTED (visible only at the subtypes)

**Inheritance and overloaded functions:**
-all attributes are inherited

- an instance of subtype can be used in every context in which a supertype instance is used

- the order of super types in the UNDER clause determines the inheritance hierarchy

- a subtype can redefine any function that is defined in its super type with the restriction that the signature is same

Fig. 12.4(c )

NOT FINAL must be created in a UDT if subtypes are allowed to be created under the UDT

**Specifying relationships via reference:**
A component attribute of one tuple can refer to another tuple attribute using REF; similar to FK, system generated OID is used

SELECT E.Employee->NAME
FROM EMPLOYEE AS E
WHERE E.Company->COMP_NAME=’ABCXYZ’;
3. The ODMG Object Model

The Object Definition Language ODL (similar to DDL)
The Object Query Language OQL (similar to DML)
Object: OID and State
Literal: value, no OID, constants; atomic or structural literals
The object has:
- Identifier (unique system wide)
- Name (entity points to DB, unique names)
- Lifetime (persistent or transient)
- Structure (atomic or not containing other objects)
- Creation (new)

(Read the rest)
Figure 12.6  Inheritance hierarchy for the built-in interfaces of the object model.
Figure 12.7  The attributes, relationships, and operations in a class definition.

class EMPLOYEE
(
    extent ALL_EMPLOYEES
    key Ssn
)
{
    attribute string Name;
    attribute string Ssn;
    attribute date Birth_date;
    attribute enum Gender{M, F} Sex;
    attribute short Age;
    relationship DEPARTMENT Works_for
        inverse DEPARTMENT::Has_emps;
    void reassign_emp(in string New_dname)
        raises(dname_not_valid);
};
class DEPARTMENT
(
    extent ALL_DEPARTMENTS
    key Dname, Dnumber
)
{
    attribute string Dname;
    attribute short Dnumber;
    attribute struct Dept_mgr {EMPLOYEE Manager, date Start_date}
        Mgr;
    attribute set<string> Locations;
    attribute struct Projs {string Proj_name, time Weekly_hours}
        Projs;
    relationship set<EMPLOYEE> Has_emps inverse EMPLOYEE::Works_for;
    void add_emp(in string New_ename) raises(euname_not_valid);
    void change_manager(in string New_mgr_name; in date Start_date);
};
Only binary relationships are represented. By specifying inverse relationships, the database system can maintain the referential integrity constraint (the related attribute exists or it is null).

The extent is a given a name and it will contain all persistent objects.

The object definition language (ODL)

**Figure 12.9a** An example of a database schema. Graphical notation for representing ODL schemas.
Figure 12.9b  An example of a database schema. A graphical object database schema for part of the UNIVERSITY database (GRADE and DEGREE classes are not shown).
Figure 12.10  Possible ODL schema for the UNIVERSITY database in Figure 12.8(b).

```
class PERSON
{
    extent PERSONS
    key Ssn
    { attribute struct PName { string FName, string Mname, string Lname } Name;
        attribute string Ssn;
        attribute date Birth_date;
        attribute enum Gender{M, F} Sex;
        attribute struct Address { short No,
            string Street,
            short Apt_no,
            string City,
            string State,
            short Zip } Address;
        short Age();
    }
    class FACULTY extends PERSON
    { extent FACULTY
        { attribute string Rank;
            attribute float Salary;
            attribute string Office;
            attribute string Phone;
            relationship DEPARTMENT Works_in inverse DEPARTMENT::Has_faculty;
            relationship set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
            relationship set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
            void give_raise(in float raise);
            void promote(in string new rank);
        }
    }
    class GRADE
    { extent GRADES
        { attribute enum GradeValues{A,B,C,D,F,I} Grade;
            relationship SECTION Section inverse SECTION::Students;
            relationship STUDENT Student inverse STUDENT::Completed_sections;
        }
    }
    class STUDENT extends PERSON
    { extent STUDENTS
        { attribute string Class;
            attribute Department Minors_in;
            relationship Department Majors_in inverse DEPARTMENT::Has_majors;
            relationship set<GRADE> Completed_sections inverse GRADE::Student;
            relationship set<CURR_SECTION> Registered_in INVERSE CURR_SECTION::Registered_students;
            void change_major(in string new major) raises(name_not_valid);
            float gpa();
            void register(in short secno) raises(section_not_valid);
            void assign_grade(in short secno; IN GradeValue grade)
                raises(section_not_valid, grade_not_valid);
        }
    }
```
class DEGREE
{
    attribute string College;
    attribute string Degree;
    attribute string Year;
};

class GRAD_STUDENT extends STUDENT
(
    extent GRAD_STUDENTS
{
    attribute set<Degree> Degrees;
    relationship set<FACULTY> Committee inverse FACULTY::On_committee_of;
    void assign_advisor(in string Lname; in string Fname)
        raises(faculty_not_valid);
    void assign_committee_member(in string Lname; in string Fname)
        raises(faculty_not_valid);
};

class DEPARTMENT
(
    extent DEPARTMENTS
    key Dname
{
    attribute string Dname;
    attribute string Dphone;
    attribute string Doffice;
    attribute string College;
    attribute FACULTY Chair;
    relationship set<FACULTY> Has_faculty inverse FACULTY::Works_in;
    relationship set<STUDENT> Has_majors inverse STUDENT::Majors_in;
    relationship set<COURSE> Offers inverse COURSE::Offered_by;
};

class COURSE
(
    extent COURSES
    key Cno
{
    attribute string Cname;
    attribute string Cno;
    attribute string Description;
    relationship set<SECTION> Has_sections inverse SECTION::Of_course;
    relationship <DEPARTMENT> Offered_by inverse DEPARTMENT::Offers;
};

class SECTION
(
    extent SECTIONS
{
    attribute short Sec_no;
    attribute string Year;
    attribute enum Quarter{Fall, Winter, Spring, Summer} Qtr;
    relationship set<Grade> Students inverse Grade::Section;
    relationship COURSE Of_course inverse COURSE::Has_sections;
};

class CURR_SECTION extends SECTION
(
    extent CURRENT_SECTIONS
{
    relationship set<STUDENT> Registered_students
        inverse STUDENT::Registered_in
    void register_student(in string Ssn)
        raises(student_not_valid, section_full);
};
Figure 12.11a  An illustration of interface inheritance via ".". Graphical schema representation.
4. Object Database Conceptual Design

**Differences between conceptual design of ODB and RDB:**
Relationships are handled differently, OID references are used. It could be single or both directions. ODB equivalent of referential integrity constraints can be enforced.

In RDB the references are single valued, in ODB they can be structures or multi-values

Inheritance handling in RDB and ODB is different as there is no built in constructs in RDB

The operations are specified at design phase in ODB, and in RDB they implement at the application level

The behavioral specification is also different (persistent vs transient)

**Mapping an EER schema to ODB schema:**

No categories and n-ary relationships.

(1) Create ODL class for each entity type or subclass. ODL class should include all attributes of the entity. Multivalued attributes use set, bag, or list. Composite attributes are mapped into a tuple

(2) Add binary relationship attributes for participating relations.
Single valued for 1:1, N:1 and collections for 1:N and M:N.

(3) Include appropriate operations for each class (not in RDB)
(4) Subclass inherits all attributes of superclass
(5) Weak entities can be mapped as same as regular entities
(6) N-ary relationships mapped to separate entities
The Object Query Language OQL

OQL is the query language proposed by ODMG

It is designed to work with programming languages, where ODMG binding is defined

OQL is embedded into programming languages and they return objects that match a type system of a given language

Examples:

**Q0.**

SELECT D.Dname
From DEPARTMENTS D (extent)
Where D.College = ‘Engineering’;

Entry point to the database is a named persistent object, whose type is a collection. DEPARTMENTS is of type set<DEPARTMENT>

Query results can be any type of object types.

**Q1.**

DEPARTMENTS;

Returns a reference to the collection of all persistent DEPARTMENT objects, where type is set<DEPARTMENT>

**Q1A.**

CS_DEPARTMENT;

Returns a reference to that individual object

(SKIP)