

Databases Illuminated

Chapter 7

The Enhanced Entity-Relationship Model and the Object-Relational Model

Why Extend the E-R Model?

- E-R suitable for traditional business applications
- E-R not semantically rich enough for advanced applications
- Applications where E-R is inadequate
 - Geographical information systems
 - Search engines
 - Data mining
 - Multimedia
 - CAD/CAM
 - Software development
 - Engineering design...and others

Specialization Abstraction

- Specialization-needed when an entity set has subsets that have special attributes or that participate in special relationships
- Process of breaking up a class into subclasses

Ex: Faculty contains AdjunctFac and FullTimeFac

- All Faculty have attributes facid, lastName, firstName, rank.
- AdjunctFac also have coursePayRate
- FullTimeFac have annualSalary

Representing Specialization

- E-ER diagram – See **Figure 7.1(a)** - shows specialization circle (*isa* relationship), and inheritance symbol (subset symbol)
- Specialization can also involve just one subclass – no need for circle, but show inheritance symbol –see **Figure 7.1(b)**

Ex. Class has LabClass specialization

- Subclasses can participate in their own relationships – See **Figure 7.1(c)**

Ex. FullTimeFac *subscribes to* Pension

Generalization Abstraction

- Inverse of specialization
- Recognizing that classes have common properties and identifying a superclass for them

Ex. Student and Faculty are both people

- Bottom-up process, as opposed to top-down process of specialization
- EER diagram is the same as for specialization – See **Figure 7.1(d)**

Generalization/Specialization Constraints

- **Subclasses can be overlapping or disjoint**
- Place **o** or **d** in specialization circle to indicate constraint
- Specialization can be **total** (every member of superclass must be in some subclass) or **partial**
 - Total -double line connecting superclass to specialization circle
 - Partial-single line
- Specialization definition can be
 - **predicate-defined** - each subclass has a defining predicate
 - **Attribute defined** – the value of the **same** attribute is used in defining predicate for all subclasses
 - **User-defined** – user responsible for identifying proper subclass

Multiple Specializations

- Can have different specializations for the same class

Ex. **Figure 7.3(a)** shows undergraduates specialized by year and by residence. These are independent of each other

- Can have shared subclasses – have multiple inheritance from two or more superclasses

Ex. **Figure 7.3(b)** shows Teaching Assistant as subclass of both Faculty and Student

Union or Category

- Subclass related to a collection of superclasses
- Each instance of subclass belongs to one, not all, of the superclasses
- Superclasses form a **union** or **category**

Ex. A Sponsor may be a team, a department, or a club – **Figure 7.4(a)**, top portion

- Each Sponsor entity instance is a member of **one** of these superclasses, so Sponsor is a subclass of the union of Team, Dept, Club
- EER diagram - connect each superclass to union circle, connect circle to subclass, with subset symbol on line bet circle and subclass

Total and Partial Unions

- **Total category** – every member of the sets that make up the union must participate
 - Shown on E-ER by double line from union circle to subset
 - In Figure 7.4(b) every Concert or Fair must be a Campus-Wide Event
- **Partial category** – not every member of the sets must participate
 - Shown by single line
 - In Figure 7.4(b) not every Club, Team, or Dept must be a Sponsor

Total Union vs. Specialization

- Total union can often be replaced by hierarchy
- Choose hierarchy representation if superclasses have many common attributes
- See **Figure 7.5**

(min..max) Notation for Relationships

- Shows both cardinality and participation constraints
- Can be used for both E-R and E-ER diagrams
- Use pair of integers (min..max) on line connecting entity to relationship diamond
 - **min** is the least number of relationship instances an entity must participate in
 - **max** is the greatest number it can participate in (can write M or N for many); some authors use * for many
 - See **Figure 7.6** and **Figure 7.7**

E-ER to Relational Model -1

- For entity sets that are not part of generalization or union, do the mapping as usual
 - Map strong entity sets to tables, with column for each attribute, but
 - For composite attributes, create column for each component, or single column for composite
 - For multi-valued attribute, create separate table with primary key of entity, plus multi-valued attribute as composite key
 - Weak entity sets – include primary key of owner
 - Binary Relationships
 - 1-M: use key of “one” side as foreign key in “many” side
 - 1-1: use either key as foreign key in the other’s table
 - M-M: create relationship table with both primary keys
 - Higher-Order Relationships-create relationship table

E-ER to Relational Model-2

- **Mapping Class Hierarchies to Tables**

- **Method 1:** Create a table for superclass and one for each of the subclasses, placing primary key of superclass in each subclass

Ex : Faculty(facId, lastName, firstName, rank)
 AdjunctFac(facId, coursePayRate)
 FullTimeFac(facId, annualSalary)

- **Method 2:** Create table for each subclass, including all attributes of superclass in each, with no table for superclass

Ex: AdjunctFac(facId, lastName, firstName, rank, coursePayRate)
 FullTimeFac(facId, lastName, firstName, rank, annualSalary)

- **Method 3:** Create a single table with all attributes of superclass and of all subclasses

Ex: AllFac(facId, lastName, firstName, rank, annualSalary, coursePayRate)

- Variation of method 3 is to add a “type field” to each record, indicating subclass it belongs to

E-ER to Relational Model-3

- **Mapping Unions**

- create table for the union itself, and individual tables for each of the superclasses, using foreign keys to connect them. Include a type code field in union table

Ex:

CampusWideEvent(eventName, date, time, **eventType**)

Concert(eventName, performer)

Fair(eventName, theme, charity)

- If superclasses have different primary keys, create a surrogate key which will be the primary key of the union

Ex:

Sponsor(**sponsorId**, sponsorType)

Club(clubName, president, numberOfMembers, **sponsorId**)

Team(teamName, coach, sport, **sponsorId**)

Department(deptName, deptCode, office, **sponsorId**)

SQL:1999 Object-Relational Features

- Richer fundamental datatypes, including types for multimedia – text, images, video, audio, etc.
- Collection types that can hold multiple attributes
- User-defined datatypes, including user-written operations to manipulate them
- Representation of class hierarchies, with inheritance of both data structures and methods
- Reference or pointer types for objects
 - enable us to refer to large objects such as multimedia files that are stored elsewhere
 - Can also store relationships, replacing foreign keys
- **NOTE: This discussion is about standard SQL:1999. Oracle and other DBMSs have different syntax**

New Datatypes

- SQL:1999 added two new fundamental datatypes
 - **BOOLEAN**
Ex. To add a matriculated field to the Student table
matriculated BOOLEAN DEFAULT false
 - **LARGE OBJECT (LOB)**
 - Store text, audio, video, multimedia objects, signatures, etc.
 - Have very restricted functions, such as substring operations
 - Comparisons only for equality or inequality,
 - Cannot be used for ordering or in GROUP BY or ORDER BY clauses
 - large files, normally simply stored and rarely retrieved again
 - Manipulated using a LOB locator, a system-generated binary surrogate for the actual value.
 - Ex.: If we stored students' signatures, add in CREATE TABLE command
signature CLOB REF IS sigid SYSTEM GENERATED,
 - sigid is a pointer to the file containing an image of the signature
 - Variants: **BLOB** (BINARY LARGE OBJECT) and **CLOB** (CHARACTER LARGE OBJECT)
- new type constructors, **ARRAY** and **ROW**

Array Type

- **ARRAY[n]**

- Ordered collection of values of the same base datatype stored as a single attribute

- must specify the base type

- *attribute-name basetype* ARRAY [n]

- Ex: If we allow students to have double majors, specify in CREATE TABLE

- majors **VARCHAR(10) ARRAY[2]**, //Oracle does not use this syntax

- Use constructor to create array of values

- INSERT INTO Stu VALUES('S555', 'Quirk', 'Sean', true,
ARRAY['French', 'Psychology'], 30);

- Use standard square bracket notation to refer to individual elements

- SELECT Stu.majors[1]

- FROM Stu

- WHERE stuld = 'S999';

Declaring Row Type

- declaration of **ROW type**
 - sequence of pairs of field names and datatypes
 - can use the ROW type declaration to create a datatype and then use the type to create a table
 - Ex:

```
CREATE ROW TYPE team_row_type(  
    teamName      CHAR(20),  
    coach         CHAR(30),  
    sport         CHAR(15));
```

```
CREATE TABLE Team OF TYPE team_row_type;
```

- can refer to an individual field using the dot notation, as in

```
SELECT Team.coach  
FROM Team  
WHERE Team.sport= 'soccer';
```

Nested Tables Using Row Type

- An attribute can have a row-type, allowing nested tables.
- Ex: An attribute of a table can represent one team the student belongs to:

```
CREATE TABLE NewStu (  
    stuld CHAR(6),  
    ...  
    team team_row_type,  
    ...);
```

Arrays of Row Types

- The ROW constructor can use any datatype for its fields, including ROW and ARRAY.
- Ex: student's record may have an array of teams:

```
CREATE TABLE NewStu2 (  
  stuld          CHAR(6),  
  lastName      CHAR(20) NOT NULL,  
  firstName     CHAR(20) NOT NULL,  
  matriculated  BOOLEAN DEFAULT false;  
  teams         team_row_type ARRAY[3],  
  majors        CHAR(10) ARRAY[2],  
  credits       SMALLINT DEFAULT 0,  
  CONSTRAINT...);
```

Using Array and Row Constructors

- Insert NewStu2 records using ARRAY and ROW constructors

```
INSERT INTO NewStu2 VALUES('S999', 'Smith', 'Michael',  
true, ARRAY[ROW('Angels','Jones','soccer'),  
ROW('Devils','Chin','swimming'),ROW('Tigers','Walters','  
basketball')], ARRAY['Math', 'Biology'], 60);
```

Accessing Attributes within Arrays and Rows

- Can refer to the individual attributes using nested dot notation

Ex: To retrieve all coaches for teams that a student is on

```
SELECT NewStu2.teams.coach  
FROM NewStu2  
WHERE stuId = "S999";
```

To get only the coach of the first team, use
SELECT NewStu2.teams[1].coach FROM...

User-defined Distinct Types

- Can construct user-defined **DISTINCT datatype**, from a base type

- Ex

```
CREATE DISTINCT TYPE studentAgeType AS INTEGER;  
CREATE DISTINCT TYPE numberOfCreditsType AS  
    INTEGER;
```

- cannot compare a studentAgeType attribute with a numberOfCreditsType attribute.

User-defined Structured Types

- **User-defined structured datatypes (UDTs)** can have several attributes
- Attributes can be any SQL type, including built-in types, LOB types, ARRAY or ROW types, or other structured types
- Can nest structured types
- a structured UDT can be either the type used for an attribute, or the type used for a table
- Attributes can be stored items or they can be virtual
- Type can be **instantiateable**, which allows instances of the type to be created, or **not instantiateable**
- SQL provides a default **constructor function** for instantiateable types
 - Has the same name and datatype as the UDT
 - Invoke it by using the name of the type
 - Takes no parameters; assigns default values to the instance attributes
- Users can also create their own constructors

Operations on UDTs

- UDTs have only a few basic operations predefined on them
 - Default constructor
 - Automatic **observer** methods that return the values of attributes
 - Automatic **mutator** methods that allow values to be assigned to attributes
 - User can override these methods by redefining them, but they cannot be overloaded
- Built-in functions (SUM, COUNT, MAX, MIN,AVG) are **not** defined for these types
- Users define operations on their UDTs by building on the basic operations defined for the source types
- The definition of a new type includes a list of its **attributes** and its **methods**
- Methods are defined only for a single user-defined type, and do not apply to other types the user creates
- Structured types can be manipulated using operations defined by the user as methods, functions, and procedures

Example of UDT

- We define a new StudentType as follows:

```
CREATE TYPE StudentType AS
(studId      VARCHAR(6),
lastName    VARCHAR(15),
firstName    VARCHAR(12),
advisorId    VARCHAR2(6),
credits      SMALLINT,
dateOfBirth DATE)
METHOD      addCredits(smallint);
INSTANTIABLE
NOT FINAL
```

Type Definition Components

- Type definition contains
 - The name of the type
 - List of attributes with their datatypes within parentheses
 - Attributes can be previously-defined UDTs
 - List of methods (if any) for the type
 - Indicate the argument types and return type (if any) for each method
 - Note that we do not indicate a primary key for the type, since we are defining a type, not a table

Writing and Using Methods

- The code for the methods is written separately
- Within the method, the implicit parameter is referred to as *self*
- Ex: The code for the addCredits method might be

```
CREATE METHOD addCredits (numberOfCredits smallint) FOR StudentType
BEGIN
    SET self.credits = self.credits + numberOfCredits;
END;
```

- When invoked, a method takes an instance of the type as an implicit parameter.
- Methods are invoked using dot notation.

Ex: if firstStudent is an instance of the StudentType, write

```
firstStudent.addCredits(3);
```

Constructing Object Tables

- Ex: Using this new StudentType, can create a new version of the Student table:

```
CREATE TABLE Student of StudentType  
(CONSTRAINT Student_stuld_pk PRIMARY KEY(stuld),  
CONSTRAINT Student_advisorId_fk FOREIGN KEY  
  (advisorId) REFERENCES Faculty (facId));
```

- These are called **object** tables
- They are tables that consist of a single UDT

Inserting Records into an Object Table

- Use the constructor for the UDT
- Ex: Use the StudentType type constructor, as in

```
INSERT INTO Student
```

```
VALUES(StudentType('S999', 'Fernandes',  
    'Luis', 'F101', 0, '25-Jan-1985'));
```

- Can also insert records in the usual way, without invoking constructor

Using Methods for UDT

- Invoke user-written methods using dot notation:
Ex. StudentType has the addCredits() method that we wrote earlier. Invoke by
`myStudent.addCredits(3);`
- Invoking built-in accessor method:
attribute-name() returns the value of the named attribute of a tuple
Ex: to get the first Name of firstStudent:
`firstStudent.firstName;()`
- Invoking built-in mutator method:
attribute-name(value) assigns the value to the named attribute
Ex: to assign value of 30 to credits of secondStudent:
`secondStudent.credits(30);`

Type Hierarchies

- Structured types can participate in type hierarchies
- Subtypes inherit attributes and operations, may have additional attributes and operations of their own
- if UDT is FINAL, no subtypes can be defined for it; if NOT FINAL, subtypes allowed
- If class is NOT INSTANTIABLE, cannot create instances of that type, but if it has subtypes, instances of the subtypes can be created, if subtypes are instantiable

EX: Using StudentType, which was NOT FINAL, can define a subtype, UndergraduateType, by writing

```
CREATE TYPE UndergraduateType UNDER StudentType AS (  
  major varchar(10) ARRAY[2])  
INSTANTIABLE,  
NOT FINAL;
```

- Can create a sub-table of Undergraduate type

```
CREATE TABLE Undergraduate OF UndergraduateType UNDER Student;
```

UDT Functions, Methods, Procedures

- UDTs and subtypes can have their own methods, functions, and procedures
- Methods have implicit *self* parameter
- Functions do not have implicit parameter, and must have a return type
- Procedures have no return type
 - can have input parameters, identified by the keyword **IN**
 - output parameters, identified by **OUT**
 - two-way parameters, identified by **IN/OUT**
 - Specify before the name of the formal parameter in the procedure heading

UDT Functions

- Ex: can define a new function, hasDoubleMajor, for the Undergraduate subtype:
CREATE FUNCTION hasDoubleMajor (u UndergraduateType)
RETURNS BOOLEAN
BEGIN
 IF (u.major[2] IS NOT NULL)
 THEN
 RETURN TRUE;
 ELSE
 RETURN FALSE;
 END IF;
END;
• To use a function in a program, we must provide an actual parameter of UndergraduateType –as in IF (hasDoubleMajor(thirdStudent)) THEN...

UDT Procedures

- Ex:

```
CREATE PROCEDURE countMajors(IN u UndergraduateStudent, OUT  
    numberOfMajors SMALLINT)
```

```
BEGIN
```

```
    numberOfMajors =0;
```

```
    IF (u.majors[1] IS NOT NULL) THEN numberOfMajors =  
        numberOfMajors +1;
```

```
    IF (u.majors[2] IS NOT NULL) THEN numberOfMajors =  
        numberOfMajors +1;
```

```
END;
```

- The procedure would be invoked in statements such as
countMajors(fourthStudent, count);
- The output parameter can be used as in
IF (count = 2) THEN...

Subtypes of Subtypes

- Can create a type called FreshmanType under UndergraduateType

```
CREATE TYPE FreshmanType UNDER  
UndergraduateType AS (  
peerMentor varchar(25))
```

```
INSTANTIABLE,
```

```
FINAL;
```

- CREATE TABLE Freshmen OF FreshmanType UNDER Undergraduate;

Creating References

- Foreign keys are used to establish relationships between strictly relational tables
- In object-relational tables can have attributes that are actually **references** or pointers to another type, in place of foreign keys
- Ex: Suppose we had already defined FacultyType and a Faculty table of that type. Now we can define StudentType with a reference to the faculty advisor, called ald:

```
CREATE OR REPLACE TYPE StudentType AS (  
  (stuld          VARCHAR(6),  
  lastName       VARCHAR(15),  
  firstName      VARCHAR(12),  
  ald            REF(FacultyType) SCOPE Faculty,  
  credits        SMALLINT,  
  dateOfBirth   DATE)  
  METHOD         addCredits(smallint);
```

Then to create the Student table, we write
CREATE TABLE Student OF StudentType;

Using Reference Types

- System generates values for references when tuples of the referenced types are inserted
- User must explicitly find and set the reference value when a referencing tuple is inserted

- Ex. To insert Student tuple, first insert with null reference for advisor's aid
`INSERT INTO Student VALUES('S555', 'Hughes', 'John', null, 0, '04-Jan-1988');`

- Then update the tuple using a sub-query to find reference to the advisor

```
UPDATE Student
SET aid = (SELECT f.aid
          FROM Faculty AS f
          WHERE facId = 'F101')
```

```
WHERE studId = 'S555';
```

- In queries, can use Deref operator to find the entire referenced tuple.

```
SELECT s.lastName, s.firstName, Deref(s.aid), s.dateOfBirth
FROM Student s
WHERE s.lastName = 'Smith';
```

- Can use the `->` operator to retrieve the value of an attribute of the tuple referenced
`aid-> lastName`

Retrieves the last name of the Faculty advisor

// Note: Oracle uses different method and syntax for references

Converting an E-ER Diagram to an Object-Relational Database

- Entity sets and relationships that are not part of generalizations or unions
 - Proceed as in the relational model
 - Entities map to tables, either using SQL2 syntax or creating a type and using that type for the table
 - For composite attributes, can use ROW type, or define a structured type for the composite, with the components as attributes of the type.
 - Multi-valued attributes use ARRAY
 - For relationships, reference types can be used in place of the foreign key
- For specialization hierarchies, types and subtypes can be defined, and tables and sub-tables UNDER them to correspond to the types
- **No subtype can have multiple inheritance**
 - choose among potential supertypes for a shared subtype
 - subtype's relationship to other supertype(s) expressed using references or foreign keys
- **No direct representation of unions (categories)**
 - can use the same method as for relational model
 - create a table for the union itself, and individual tables for each of the superclasses, using foreign keys or references to connect them.