CHAPTER LOG

Synopsis
Chapter 3 presented several advanced features of the SQL standard. This chapter continues that coverage with a presentation of the object-relational features of the standard. After presenting the use of constructed types such as row types and arrays, the chapter focuses on the use of User-Defined Types (UDTs). UDTs provide extensibility to the SQL predefined types, where the behavior of the type is defined through the use of methods. UDTs also provide the basis for the creation of typed tables. Typed tables are the relational equivalent of classes in the object-oriented data model, where typed tables can be formed into hierarchies and instances of typed tables have object identifiers. References to objects can then be used to create relationships between typed tables. This chapter elaborates on the use of these object-relational features, also providing guidelines for mapping EER and UML conceptual designs to object-relational designs via the SQL standard.

Assumed Knowledge
- Enhanced Entity Relationship (EER) Diagrams (Chapter 1)
- Unified Modeling Language (UML) Conceptual Class Diagrams (Chapter 2)
- The SQL Standard: Advanced Relational Features (Chapter 3)
- Mapping Object-Oriented Conceptual Models to the Relational Data Model (Chapter 4)
- Object-Oriented Databases and the ODMG Standard (Chapter 7)

Case Study
- Object-Relational Design of the School Database Enterprise (Chapter 9)
OUTLINE

- Introduction to Object-Relational Technology
- Constructed Types
  - Row Types
  - Arrays
- User-Defined Data Types (UDTs)
  - Distinct Types
  - Structured Types
  - Type Hierarchies
- Typed Tables and Table Hierarchies
- Reference Types
- Mapping to the SQL Object-Relational Features
OBJECT-RELATIONAL TECHNOLOGY

Motivation

- The simplicity of the relational model has contributed to the popularity of relational database systems in industry.
  - Data stored in table form.
  - Set-orientation supports powerful query operations.
- The changing nature of information technology, however, has challenged the application of traditional relational concepts.
  - Object-oriented programming
  - Object-oriented database systems
    - objects, attributes, methods, encapsulation, polymorphism, inheritance
  - Non-traditional data
    - complex relationships, new types (multimedia, spatial, etc.), user-defined types.
EVOLUTIONARY VS. REVOLUTIONARY APPROACH

- Object-oriented database systems are viewed as a revolutionary approach to changing information processing needs, defining a totally new type of database system.

- Object-relational systems offer an evolutionary approach to changing information processing needs.
  - Traditional relational database technology is a multi-billion dollar business.
  - Decades of research on relational database technology (transaction processing, concurrency control, query processing).
  - Relational techniques can be extended with object-oriented concepts to build on the existing technology and market base.
ORIGIN OF OBJECT-RELATIONAL CONCEPTS

• Michael Stonebraker pioneered the development of object-relational concepts.
  o 1970’s/1980’s – Developed Ingres as a relational database research project.
  o 1990’s – Developed Postgres as an extension to Ingres to demonstrate object-oriented extensions to relational database technology.
• Important papers influencing the development of object-relational technology:
• Today, most major relational database vendors support universal data managers (i.e., relational technology extended with object-oriented concepts.)
THE SCHOOL DATABASE EXAMPLE

From Succeeding with Object Databases, by Akmal B. Chaudri and Roberto Zicari. Copyright © 2000 by Akmal B. Chaudri and Roberto Zicari. All rights reserved. Reproduced here by permission of Wiley Publishing, Inc.
CONSTRUCTED TYPES

- SQL has traditionally supported atomic types for the definition of columns, variables, and parameters.
- SQL provides *constructed types*, which are data types that are capable of holding multiple values.
- **Built-in Constructed Types:**
  - Row Types
  - Arrays
- **User-Defined Constructed Types:**
  - Distinct Types
  - Structured Types
- **Reference Types**
ROW TYPES

- *Row types* allow rows to be stored as attribute values inside of other rows.
- Using row types, tables can be created with columns that have non-atomic values.
- The elements of a row are called *fields*.
- In the School Database Enterprise, the location of a campus club can be represented as a row type.

```sql
create table campusClub
( cId varchar(10),
  name varchar(50) not null,
  location row (street varchar(30), bldg varchar(5), room varchar(5)),
  advisor varchar(11) references faculty(pId),
  primary key (cId));
```
CONSTRUCTING ROW VALUES

- The row constructor is used to assign values to the fields of a row.
- The values in the row constructor can either be a list of values or the result of a query.

```sql
insert into campusClub values
    ('CC123',
    'Campus Computer Club',
    row('Mill Avenue, 'Brickyard Building', 'Rm 222'),
    'FA123');
```
ACCESSING ROW VALUES

The fields of a row type are accessed using dot notation.

```sql
select c.location.street, c.location.bldg, c.location.room
from campusClub c
where c.name = 'Campus Computer Club';
```
ARRAYS AS COLLECTIONS

- Collections are represented in SQL through the use of the array constructed type.
- An array specification indicates the number of elements in the array and the data type of the array contents.
- In the School Database, an array definition can be used to store member information.

```sql
create table campusClub
( cld varchar(10),
  name varchar(50) not null,
  location row (street varchar(30), bldg varchar(5), room varchar(5)),
  advisor varchar(11) references faculty(pld),
  members varchar(11) array[50] references student(pld),
  primary key (cld));
```
CONSTRUCTING ARRAYS

- The array constructor is used to assign values to an array.
- `array[ ]` constructs an empty array.
- `array['value1', 'value2', 'value3']` constructs an array with three values.

```sql
insert into campusClub values
    ('CC123',
     'Campus Computer Club ',
     row('Mill Avenue', 'Brickyard Building', 'Rm 222'),
     'FA123',
     array[ ]); 
```

```sql
update campusClub
    set members = array['ST111', 'ST222', 'ST333']
where name = 'Campus Computer Club';
```
ACCESSING ARRAYS

- An index can be used to access a value in a specific position of an array.
- The query below returns the identifier of the second element of the members array.

```
select members[2]
from campusClub
where name = 'Campus Computer Club';
```
USER-DEFINED TYPES

- User-defined data types (UDT’s) allow users to define their own types and operations on those types.
  - Distinct Types
  - Structured Types
- UDT’s can be used as types of columns within tables.
- Structured types can also be used as the basis for creating typed tables.
- Instances of typed tables are similar to creating objects as instances of classes in object-oriented database systems.
DISTINCT TYPES

- A *distinct type* is a limited version of a UDT that is based on a single atomic data type.
- A distinct type allows atomic types to be used in a way that is meaningful to the application.

```sql
create type age as integer final;
create type weight as integer final;

create table person
  (  personId varchar(3),
      personAge age,
      personWeight weight,
      primary key (personId));
```
USING DISTINCT TYPES

- The use of a distinct type cannot be mixed with the use of the data type on which it is based without the specific use of the `cast` statement.
- Allowed:
  ```sql
  select p1.personId
  from person p1, person p2
  where p2.personId = '123' and p1.personAge < p2.personAge;
  ```
- Not allowed:
  ```sql
  select personId
  from person
  where (personAge * 2) < personWeight;
  ```
- Allowed:
  ```sql
  select personId
  from person
  where cast (personAge as integer) * 2 < cast (personWeight as integer);
  ```
STRUCTURED TYPES

- A **structured type** is a UDT that is composed of several internal components, called *attributes*.
- A structured type may have *method* definitions associated with the type.
- A method is a function (not a procedure!) that is tightly bound to the type definition.
- A structured type is the SQL approach to supporting *encapsulation*, where the type is manipulated through the methods that are part of the type definition.
- When used as a column type, variable type, or parameter type, an instance of a structured type is a *value* and *not* an object.
- An instance of a structured type becomes an *object* with object identity when the type is used in conjunction with typed tables.
DEFINING LOCATION AS A STRUCTURED TYPE

create type locationUdt as (  
  street varchar(30),  
  bldg varchar(5),  
  room varchar(5))  
not final; /* This is a mandatory finality clause indicating  
that the subtype may have proper subtypes. */

create table campusClub (  
  cId varchar(10),  
  name varchar(50) not null,  
  location locationUdt, -- used as a column type  
  advisor varchar(11) references faculty(pld),  
  members varchar(11) array[50] references student(pld),  
  primary key (cId));
BUILT-IN METHODS FOR STRUCTURED TYPES

- A *constructor function* for creating instances of the type.
  - Same name as the type.
  - Invoked using the `new` expression.
- *Observer functions* for retrieving the attribute values of a structured type.
  - Same name as the attribute.
  - Invoked using dot notation.
- *Mutator functions* for modifying the attribute values of a structured type.
  - Same name as the attribute.
  - Invoked using dot notation.
USING CONSTRUCTOR AND MUTATOR METHODS

begin

declare loc locationUdt;
set loc = new locationUdt(); /* invoking the constructor function */
set loc.street = 'Mill Avenue'; /* invoking the mutator functions */
set loc.bldg = 'Brickyard Building';
set loc.room = 'RM 222';
insert into campusClub values(
    'CC123',
    'Campus Computer Club',
    loc, /* initializing location */
    'FA123',
    array[ ]);
USING OBSERVER FUNCTIONS

Structured types and observer functions in queries can only be accessed through the use of alias names.

Allowed:
```
select name, c.location.street, c.location.bldg, c.location.room
from campusClub c
where name = 'Campus Computer Club';
```

Not Allowed (because of potential naming ambiguity in SQL):
```
select name, location.street, location.bldg, location.room
from campusClub
where name = 'Campus Computer Club';
```
ADDING A METHOD SIGNATURE TO A TYPE DEFINITION

- The method on the type below will return the sum of the attributes values of the structured type.
- The implicit `self` parameter is used to refer to the type instance.

```sql
create type threeNumbers as(
    one integer,
    two integer,
    three integer)
not final
method sum( ) returns integer;

create method sum( ) returns integer for threeNumbers
begin
    return self.one + self.two + self.three;
end
```
OVERRIDING THE CONSTRUCTOR FUNCTION

create type locationUdt as (  
    street varchar(30),  
    bldg varchar(5),  
    room varchar(5))  
not final  
overriding constructor method locationUdt /* a new constructor with parameters */  
(street varchar(30), bldg varchar(5), room varchar(5))  
returns locationUdt;

create method locationUdt(st varchar(30), bl varchar(5), rm varchar(5))  
returns locationUdt for locationUdt  
begin  
    set self.street = st; set self.bldg = bl; set self.room = rm;  
    return self;  
end;
TYPED TABLES

- A typed table is a table that is associated with a specific structured type.
- A typed table has a column for each of the attributes of the structured type on which it is based.
- In addition, a typed table has a self-referencing column that contains a unique identifier for each row in the table.
- When a structured type is used to define a typed table, an instance of the type is viewed as an object, with the self-referencing column providing the object identifier.
- The object identifier is unique only within a specific typed table (i.e., two typed tables may have rows with identical self-referencing values).
RULES FOR CREATING TYPED TABLES

- Other than the self-referencing column and the attributes of the structured type, additional columns cannot be added to the table definition.
- Table and referential integrity constraints can be defined on the columns of the typed table (i.e., the attributes of the structured type).
- The means for generating the value for the object reference must be specified in the type and in the typed table. Object references can be specified as either:
  - *system generated* (guaranteed unique by the database)
  - *user defined* (the user must provide the unique value)
  - *derived* (the user must specify attributes that serve as an object reference)
CREATING A TYPED TABLE FROM A STRUCTURED TYPE

create type departmentUdt as (  
    code     varchar(3),  
    name     varchar(40))
  instantiable
  not final
  ref is system generated;

create table department of department_udt (  
    primary key (code),  
    ref is departmentID system generated);
CREATING INSTANCES OF A TYPED TABLE

insert into department values ('cse', 'Computer Science and Engineering');
Insert into department values ('ece', 'Electrical and Computer Engineering');
insert into department values ('mae', 'Mechanical and Aerospace Engineering');

<table>
<thead>
<tr>
<th>(self-referencing column)</th>
<th>code</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>10287534556</td>
<td>cse</td>
<td>Computer Science and Engineering</td>
</tr>
<tr>
<td>27259489035</td>
<td>ece</td>
<td>Electrical and Computer Engineering</td>
</tr>
<tr>
<td>90324854948</td>
<td>mae</td>
<td>Mechanical and Aerospace Engineering</td>
</tr>
</tbody>
</table>
TYPE AND TABLE HIERARCHIES

- Structured types and typed tables can be used to create the equivalent of superclass/subclass hierarchies in object-oriented databases.
- Structured types are first formed into a hierarchy using the under keyword, supporting inheritance of attributes and methods.
- Typed tables can be created to correspond to the structured type hierarchy.
FORMING OBJECT HIERARCHIES

Structured type hierarchy

create type personUdt as(
  pId varchar(11),
  firstName varchar(20),
  lastName varchar(20),
  dob date)

instantiable
not final
ref is system generated;

create type facultyUdt under personUdt as(
  rank varchar(10))

instantiable
not final;

create type studentUdt under personUdt as(
  status varchar(10))

instantiable
not final;

Corresponding table hierarchy

create table person of personUdt (  
  primary key (pId),  
  ref is personID system generated);

create table faculty of facultyUdt under person;
create table student of studentUdt under person;
INSTANTIABLE VS. NOT INSTANTIABLE TYPES

- SQL supports the object-oriented concept of abstract supertypes through the use of the instantiable clause in structured type definitions.
- If a type is specified as instantiable, then the type has a constructor and instances of the type can be directly created (the default option).
- If a type is specified as non instantiable, then the type has no constructor method defined and instances of the type cannot be created.
- A non instantiable type is only useful if it has instantiable subtypes.
- Typed tables can only be used with instantiable types.
RULES FOR TYPE AND TABLE HIERARCHIES

- Only single inheritance is supported. Every subtype/subtable must have only one maximal supertype/supertable (the root of the hierarchy).
- The concept of an interface for inheritance of behavior only is not supported.
- Every instance/row has a most specific type/table, which is the type/table where the instance was directly created.
- An instance/row cannot migrate to other types/tables in the hierarchy.
ADDITIONAL RULES FOR TYPE/TABLE HIERARCHIES

- A primary key can only be defined for a maximal supertable. Subtables, however, can specify attributes as unique.
- A self-referencing column is only defined for a maximal supertable. The self-referencing column is inherited by all subtables.
- Constraints in a typed table definition can only be defined on *originally-defined attributes* (i.e., the new attributes introduced in the structured type) and not on inherited attributes.
INSERTING INTO TYPED TABLES

- Insert statements at the subclass levels should include values for inherited attributes.
- Examples (ignoring the mandatory specialization constraint for illustration purposes):

```sql
insert into person values('PP111', 'Joe', 'Smith', '2/18/82');
insert into person values('PP222', 'Alice', 'Black', '2/15/80');
insert into student values('ST333', 'Sue', 'Jones', '8/23/87', 'freshman');
insert into student values('ST444', 'Joe', 'White', '5/16/86', 'sophomore');
insert into faculty values('FA555', 'Alice', 'Cooper', '9/2/51', 'professor');
```
QUERYING TABLES HIERARCHIES

- Querying the person supertable returns Smith, Black, Jones, White, and Cooper.
  
  ```sql
  select firstName, lastName
  from person;
  ```

- Query the faculty subtable returns only Cooper (similar query for the student subtable returns only Jones and White).
  
  ```sql
  select firstName, lastName
  from faculty;
  ```

- Use the `only` keyword to retrieve only the direct instances of the person supertable, Smith and Black (and not the instances of the subtables).
  
  ```sql
  select firstName, lastName
  from only (person);
  ```
DELETING FROM TABLE HIERARCHIES

- The following statement will delete anyone with the first name of ‘Alice’ from the person table hierarchy (Black and Cooper), even if the most specific type of Alice is a student or a faculty.

  delete from person where firstName = 'Alice';

- Deleting a tuple from a subtable also deletes the tuple from its supertables. The deletion of Sue Jones from the student table will also result in Sue Jones no longer being visible from the person table.

  delete from student where firstName = 'Sue' and lastName = 'Jones';

- The following statement will delete rows from person with a first name of ‘Joe’ only if the person (Smith) is not also a student or a faculty.

  delete from only (person) where firstName = 'Joe';
UPDATING TABLE HIERARCHIES

- The following statement will change the first name of rows (Jones) in the person table and in any of its subtables.

  ```sql
  update person
  set firstName = 'Suzy'
  where firstName = 'Sue';
  ```

- The following statement will change the first name of rows (Smith) in the person table, but not in its subtables.

  ```sql
  update only (person)
  set firstName = 'Joey'
  where firstName = 'Joe';
  ```
REFERENCE TYPES (REF)

- A reference type (ref) can be used to define the type of a column in a table, an attribute in a structured type, a variable, or a parameter.
- A reference type stores the self-referencing value of a row in a typed table.
- Ref types can be used to model object associations that are based on object identity rather than on foreign keys.
USING REFS IN CAMPUS CLUB

create type campusClubUdt as(
    cId varchar(10),
    name varchar(50) not null,
    location locationUdt,
    phone varchar(12),
    advisor ref(facultyUdt),  /* Stores a reference to a faculty object */
    members ref(studentUdt) array[50])  /* Stores an array of references to student objects */

create table campusClub of campusClubUdt(
    primary key (cId),
    ref is campusClubID system generated);

©2005 by Dietrich and Urban
THE SCOPE CLAUSE

- The scope clause can be used to specify the typed table from which reference type values can be used.
- The scope clause should also specify whether references are checked.
- If references are not checked, then the system does not validate the reference value that is stored.
- If references are checked, then the system does validate that the ref value is from the specified table.

advisor ref(facultyUdt) scope faculty

references are checked on delete set null
SETTING REFERENCE VALUES

The name of the self-referencing column is used to assign a ref value to a column defined as a ref type.

```
update campusClub
set advisor = (select personID
              from person
              where fName = 'Alice' and lName = 'Cooper')
where name = 'Campus Computer Club';
```
REFERENCE VALUES IN QUERIES

- The dereference operator (→) is used to traverse through reference values (an implicit join operator).
- The following query will display the name of the advisor of the Campus Computer Club.

```sql
select advisor → fName
from campusClubs
where name = 'Campus Computer Club';
```
RETRIEVING STRUCTURED TYPES IN QUERIES

- The `deref()` function can be used to return the entire structured type that is referenced by a ref type.
- The following query will return all instances of the `facultyUdt` values for advisors of campus clubs that are located in the Brickyard Building (a table with one column of type `facultyUdt`).

```sql
select deref (advisor)
from campusClub c
where c.location.bldg = 'Brickyard Building';
```
MAPPING TO
OBJECT-RELATIONAL FEATURES

- Mappings are addressed for:
  - Classes
  - Associations
  - Class Hierarchies
  - Shared Subclasses
  - Categories

- The mapping procedure emphasizes the use of:
  - Structured types with typed tables for classes
  - Reference types and arrays of reference types for modeling bi-directional associations. The user must write procedures or triggers to maintain inverse relationships.
  - Reference types and/or arrays of reference types together with methods to model uni-directional associations
ER MODEL
Abstract Enterprise

[Diagram of an ER model with entities and relationships labeled including attrA1, attrA2, attrA3, compositeOfA, attrOfAB, keyOfA, attrOfA, derivedAttr, dependsOn, attrOfWk, partialKey, attrOfC, multiValuedAttr, keyOfC, bc, parent, child, keyOfB, attrOfB, ab, N, M, ba, 1, 1, N, 1]
UML Abstract Enterprise

Model as a record with attrA1, attrA2, and attrA3

```
<persistent>
A
  keyOfA
  attrOfA
  compositeOfA/derivedAttr
  partialKey
    dependsOn
      0..1
  1

<persistent>
Weak
  attrOfWk
```

```
<persistent>
AB
  attrOfAB
```

```
<persistent>
B
  keyOfB
  attrOfB
  parent
    0..1 bb
  0..*
    child
```

```
<persistent>
BA
  attrOfBA
```

```
<persistent>
C
  keyOfC
  attrOfC
  multiValuedAttr[*]
  bc
    1..1
```

Ch 8. The SQL Standard: Object-Relational Features 46 ©2005 by Dietrich and Urban
CLASSES
Simple and Composite Attributes

Create a structured type and a typed table for each class.
Composite attributes (compositeOfA) can be represented using a row type or a structured type.
Derived attributes (derivedAttr) can be defined as a method in the type definition.
SIMPLE AND COMPOSITE ATTRIBUTE EXAMPLE

create type aUdt as
(keyOfA varchar(3),
attrOfA varchar(3),
compositeOfA row(attrA1 varchar(3), attrA2 varchar(3), attrA3 varchar(3)))

create table a of aUdt
(primary key(keyOfA),
ref is aID system generated);
/* declare primary keys and any other constraints on columns */

method derivedAttr( ) returns varchar(3);

ref is system generated

Multi-valued attributes can be directly represented using arrays.

create type cUdt as
(keyOfC varchar(3),
attrOfC varchar(3),
multiValuedAttr varchar(3) array[10])
instantiable not final ref is system generated;

create table c of cUdt ...
ASSOCIATIONS WITHOUT ATTRIBUTES

Use reference types between the classes involved in the association.
ASSOCIATIONS WITHOUT ATTRIBUTES

1:1 Example

create type bUdt as
(....
bTOc ref(cUdt) scope c references are checked on delete no action, ...
...)
instantiable not final ref is system generated;

create type cUdt as
(....
cTOb ref(bUdt) scope b references are checked on delete set null, ...
...)
instantiable not final ref is system generated;

create table b of bUdt ... bTOc with options not null ...

create table c of cUdt ...
ASSOCIATIONS WITHOUT ATTRIBUTES
Fictitious M:N Example

cREATE TYPE bUdt AS
(....
bTOMANYc ref(cUdt) scope c array[10] references are checked on delete cascade,
....)
instantiable not final ref is system generated;

cREATE TYPE cUdt AS
(...
cTOMANYb ref(bUdt) scope b array[10] references are checked on delete set null,
....)
instantiable not final ref is system generated;

cREATE TABLE b OF bUdt ...

cREATE TABLE c OF cUdt ...
ASSOCIATIONS WITHOUT ATTRIBUTES
Fictitious 1:N Example

create type bUdt as
(....
bTOMANYc ref(cUdt) scope c array[10] references are checked on delete cascade,
...) instantiable not final ref is system generated;

create type cUdt as
(...
cTOMANYb ref(bUdt) scope b references are checked in delete set null,
...) instantiable not final ref is system generated);

create table b of bUdt ...

create table c of cUdt ...
ASSOCIATIONS WITHOUT ATTRIBUTES
Structural Constraints

- **Cardinality Ratio/Multiplicity Constraints**
  Inherent in the specification of the object-relational schema by defining the reference type of the relationship attribute and its inverse attribute.

- **Participation Constraints**
  Participation constraints are inherent in the specification of the object-relational schema through the use of the not null constraint. References can also be automatically checked and dangling references can be prevented through the constraint specifications of table definitions.
ASSOCIATIONS WITH ATTRIBUTES

Create a typed table to represent the association, known as the association table, with appropriate attributes in the association table and bi-directional inverse relationships with the structured types that participate in the association.
REIFIED ASSOCIATION
ASSOCIATIONS WITH ATTRIBUTES

Example

create type aUdt as
(…
aTOab ref(abUdt) scope ab array[10] references are checked on delete set null,
….) instantiable not final ref is system generated;
create type bUdt as
(…. bTOab ref(abUdt) scope ab array[10] references are checked on delete set null,
….) instantiable not final ref is system generated;
create type abUdt as
(…
attrOfAB varchar(3),
abTOa ref(aUdt) scope a references are checked on delete cascade,
abTOb ref(bUdt) scope b references are checked on delete cascade
…. ) instantiable not final ref is system generated;

create table a of aUdt …
create table b of bUdt …
create table ab of abUdt …
RECURSIVE ASSOCIATIONS

Use attributes with recursive reference types to the structured type of the typed table.
create type bUdt as
( ...
  parent ref(bUdt) scope b references are checked on delete set null,
  child ref(bUdt) scope b array[10] references are checked on delete set null,
  ...) instantiable not final ref is system generated;

create table b of bUdt ...
N-ARY ASSOCIATIONS

EER

- Person
  - pId
  - pName
  - inventoryId
  - bankId
  - make
  - model
  - loanAmount

- Car
  - inventoryId
  - make
  - model

- Bank
  - bankId

- finance
  - m
  - n

UML

- Person
  - pId
  - pName

- Car
  - inventoryId
  - make
  - model

- Bank
  - bankId

- Finance
  - loanAmount
N-ARY ASSOCIATIONS

Example

create type financeUdt ( …
    financedBank    ref(bankUdt) scope bank references are checked on delete cascade,
    financedCar     ref(carUdt) scope car references are checked on delete cascade,
    financedPerson  ref(personUdt) scope person references are checked on delete cascade …);
create type bankUdt ( …
    carsFinanced ref(financeUdt) scope finance array[10] references are checked on delete set null …) instantiable not final ref is system generated;
create type carUdt ( …
    financedBy ref(financeUdt) scope finance references are checked on delete set null …) instantiable not final ref is system generated;
create type personUdt ( …
    carsFinanced ref(financeUdt) scope finance array[10] references are checked on delete set null …) instantiable not final ref is system generated;
create table finance of financeUdt …
create table bank of bankUdt …
create table car of carUdt …
create table person of personUdt …
NAVIGATION
Unidirectional Associations

Use a ref or an array of refs on one side of the relationship with a method to derive the inverse on the other side of the relationship.

create type bUdt ( ...
    bTOc ref(cUdt) scope c
    references are checked
    on delete no action,    ...
    instantiable not final
    ref is system generated;
create type cUdt ( ...) instantiable not final
    ref is system generated
    method cTOc( ) return bUdt ...;
create table b of bUdt .. bTOc
    with options not null ...;
create table c of cUdt ...

©2005 by Dietrich and Urban
WEAK OR QUALIFIED ASSOCIATION

- Create a typed table for the weak class with a composite key that includes the key of the owner class.
- The weak class can also include a relationship to the object of the owner class.
WEAK OR QUALIFIED ASSOCIATION

Example

create type aUdt as ( ...
    keyOfA varchar(3),
    linkToWeak ref(weakUdt) scope weak array[10] references are checked
        on delete set null, … ) instantiable not final ref is system generated;
create type weakUdt as ( ...
    partialKey varchar(3),
    keyOfA varchar(3),
    linkToOwner ref(aUdt) scope a references are checked
        on delete cascade … ) instantiable not final ref is system generated;
create table a of aUdt ( 
    primary key (keyOfA) …);
create table weak of weakUdt ( 
    linkToOwner with options not null, 
    foreign key(partialKey) references a(keyOfA) on delete cascade, 
    primary key(partialKey, keyOfA) …);
UNIDIRECTIONAL
ABSTRACT ENTERPRISE IN UML

Model as a record with, attrA1, attrA2, and attrA3

«persistent»
A
keyOfA
attrOfA
compositeOfA / derivedAttr
partialKey
dependsOn
0..1
1

«persistent»
BA
attrOfBA

0..1
1..1

«persistent»
B
keyOfB
attrOfB

0..1
bc

1..1

«persistent»
C
keyOfC
attrOfC
multiValuedAttr[*]
OBJECT-RELATIONAL SCHEMA
Abstract Enterprise – Table a

create type aUdt as
(keyOfA varchar(3),
attrOfA varchar(3),
compositeOfA row(attrA1 varchar(3), attrA2 varchar(3), attrA3 varchar(3)),
aTOab ref(abUdt) scope ab array[10] references are checked on delete set null,
aTOb ref(bUdt) scope ab references are checked on delete cascade,
attrOfBA varchar(3),
linkToWeak ref(weakUdt) scope weak array[10] references are checked on delete set null)
instantiable not final ref is system generated
method derivedAttr( ) returns varchar(3);

create table a of aUdt
(aTOb with options not null,
aTOab with options not null,
primary key(keyOfA),
ref is aID system generated);
OBJECT-RELATIONAL SCHEMA
Abstract Enterprise – Weak Entity

create type weakUdt as
(partialKey varchar(3),
keyOfA varchar(3),
attrOfWk varchar(3),
linkToOwner ref(aUdt) scope a references are checked on delete cascade)

instantiable not final ref is system generated;

create table weak of weakUdt
(linkToOwner with options not null,
foreign key(partialKey) references a(keyOfA) on delete cascade,
primary key(partialKey, keyOfA),
ref is wID system generated);
create type bUdt as
(keyOfB   varchar(3),
attrOfB   varchar(3),
parent    ref(bUdt) scope b references are checked on delete set null,
bTOab     ref(abUdt) scope ab array[10] references are checked on delete set null,
bTOc      ref(cUdt) scope c references are checked on delete no action)
instantiable  not final  ref is system generated
method bTOa( ) returns aUdt,
method child( ) returns bUdt array[10];

create table b of bUdt
(bTOc with options not null,
primary key(keyOfB),
ref is bID system generated);
create type abUdt as
(attrOfAB varchar(3),
abTOa ref(aUdt) scope a references are checked on delete cascade,
abTOb ref(bUdt) scope b references are checked on delete cascade)
imstantiable not final ref is system generated;

create table ab of abUdt
(abTOa with options not null,
abTOb with options not null,
ref is abID system generated);
create type cUdt as
(keyOfC varchar(3),
attrOfC varchar(3),
multiValuedAttr varchar(3) array[10])
instantiable not final ref is system generated
method cTOb( ) returns bUdt;
create table C of cUdt
(primary key(keyOfC),
ref is cID system generated);
CLASS HIERARCHIES

The under clause in type and table definition supports the inheritance of state and behavior.

```sql
create type personUdt as (
pId varchar(11),
firstName varchar(20),
lastName varchar(20),
dob date)
instantiable
not final
ref is system generated;
create type facultyUdt under personUdt as (
rank varchar(20))
instantiable
not final;
create type studentUdt under personUdt as (
status varchar(2))
instantiable
not final;
create table person of personUdt (primary key (pId),
ref is personID system generated);
create table faculty of facultyUdt under person;
create table student of studentUdt under person;
```
CLASS HIERARCHIES
Specialization Constraints

- **Total vs. Partial Specialization**
  - The specialization of a supertable does not require total participation of the supertable in its specialization associations.
  - Implementation: Develop the application to prevent the creation of an object directly in the supertable.

- **Disjoint vs. Overlapping Specialization**
  - The `under` clause (as in most object-oriented implementations) inherently supports a *disjoint* specialization of a supertable into its subtables.
  - Implementation: Be creative.
SHARED SUBCLASSES

Multiple inheritance of state and behavior is not supported in SQL.
CATEGORIES

- Create a typed table for the category class.
- Create uni-directional attributes that point to the typed tables that define the category class.
- For each instance of the category typed table, only one of the uni-directional attributes will have a value (to indicate the type of the category object). The other uni-directional attribute(s) will always be null.
CATEGORIES
Example: EER

Entity: Person
- pId
- name
- phone
- gender
- address

Entity: Company
- cId
- cName

Association: Sponsor
- 1:
  - sponsoredBy

Entity: ModelingProject
- description
In UML, the category mapping also applies to the use of XOR Constraints (similar to partial or total EER category).
create type personUdt as ( … ) instantiable not final ref is system generated;

create type companyUdt as ( … ) instantiable not final ref is system generated;

create type sponsorUdt as ( …
  personSponsor ref(personUdt),  //Can also be implemented as bi-directional
  companySponsor ref(companyUdt)  //relationships for total categories.
  … ) instantiable not final ref is system generated;

create table person of personUdt …
create table company of companyUdt …
create table sponsor of sponsorUdt …
OBJECT-RELATIONAL SCHEMA
School Database

create type personUdt as
  (  pId varchar(11),
      firstName varchar(20),
      lastName varchar(20),
      dob date)
  instantiable not final ref is system generated;
create type facultyUdt under personUdt as
  (  rank varchar(10),
      advisorOf ref(campusClubUdt) scope campusClub array[5] references are checked on delete set null,
      worksIn ref(departmentUdt) scope department references are checked on delete no action,
      chairOf ref(departmentUdt) scope department references are checked on delete set null)
  instantiable not final;
create type studentUdt under personUdt as
  (  status varchar(10),
      memberOf ref(campusClubUdt) scope campusClub array[5] references are checked on delete set null,
      major ref(departmentUdt) scope department references are checked on delete no action)
  instantiable not final;
create table person of personUdt(primary key (pId), ref is personID system generated);
create table faculty of facultyUdt under person;
create table student of studentUdt under person;
create type departmentUdt as
( code varchar(3),
  name varchar(40),
  deptChair ref(facultyUdt) scope faculty referenced are checked on delete no action)
instantiable not final ref is system generated
method getStudents() returns studentUdt array[1000],
method getFaculty() returns facultyUdt array[50];
create table department of department_udt
( deptChair with options not null, primary key (code), ref is departmentID system generated);
create type locationUdt as
( street varchar(30),
  bldg varchar(5),
  room varchar(5)) not final;
create type campusClubUdt as
( cId number,
  name varchar(50),
  location locationUdt,
  phone varchar(12),
  advisor ref(facultyUdt) scope faculty references are checked on delete cascade,
  members ref(studentUdt) scope student array[50] references are checked on delete set null)
instantiable not final ref is system generated;
create table campusClub of campusClubUdt( primary key (cId), ref is campusClubID system generated);