CSE 532 – Theory of Database Systems

Lecture 22 (Chapter 14)
Object Databases

Ilchul Yoon
Assistant Professor
State University of New York, Korea

What’s in This Module?

- Motivation
- Conceptual model
- SQL:1999/2003 object extensions
- ODMG
  - ODL – data definition language
  - OQL – query language
- CORBA
Problems with Flat Relations

Consider a relation

Person(SSN, Name, PhoneN, Child)

with:

- FD: SSN → Name
- Any person (identified by SSN) can have several phone numbers and children
- Children and phones of a person are not related to each other except through that person

An Instance of Person

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>PhoneN</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-123-4567</td>
<td>222-33-4444</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-345-6789</td>
<td>222-33-4444</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-123-4567</td>
<td>333-44-5555</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
<td>516-345-6789</td>
<td>333-44-5555</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-6543</td>
<td>444-55-6666</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-1111</td>
<td>555-66-7777</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-6543</td>
<td>555-66-7777</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
<td>212-987-1111</td>
<td>444-55-6666</td>
</tr>
</tbody>
</table>
Dependencies in Person

Join dependency (JD):
   Person = (SSN,Name,PhoneN) \rightarrow (SSN,Name,Child)

Functional dependency (FD):
   SSN \rightarrow Name

Redundancies in Person

- Due to the JD:
  Every PhoneN is listed with every Child SSN
  Hence Joe Public is twice associated with 222-33-4444
  and with 516-123-4567
  Similarly, for Bob Public and other phones/children

- Due to the FD:
  Joe Public is associated with the SSN 111-22-3333 four times (for each of Joe’s child and phone)!
  Similarly for Bob Public
Dealing with Redundancies

- What to do?
- **Normalize!**
  - Split Person according to the JD
  - Then each resulting relation using the FD
  - Obtain four relations (two are identical)

Normalization removes redundancy:

<table>
<thead>
<tr>
<th>Person1</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>Name</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>Joe Public</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>Bob Public</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>PhoneN</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-22-3333</td>
<td>516-345-6789</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>516-123-4567</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>212-987-6543</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>212-135-7924</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-22-3333</td>
<td>222-33-4444</td>
</tr>
<tr>
<td>111-22-3333</td>
<td>333-44-5555</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>444-55-6666</td>
</tr>
<tr>
<td>222-33-4444</td>
<td>555-66-7777</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>ChildOf</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-22-3333</td>
<td></td>
</tr>
<tr>
<td>111-22-3333</td>
<td></td>
</tr>
<tr>
<td>222-33-4444</td>
<td></td>
</tr>
<tr>
<td>222-33-4444</td>
<td></td>
</tr>
</tbody>
</table>
But querying is still cumbersome:

- Get the phone numbers of Joe’s grandchildren.

Against the original relation: three cumbersome joins

```
SELECT G.PhoneN
FROM Person P, Person C, Person G
WHERE P.Name = 'Joe Public' AND
      P.Child = C.SSN AND C.Child = G.SSN
```

Against the decomposed relations is even worse: four joins

```
SELECT N.PhoneN
FROM Person1 P, ChildOf C, ChildOf G, Phone N
WHERE P.Name = 'Joe Public' AND P.SSN = C.SSN AND
      C.Child = G.SSN AND G.Child = N.SSN
```

Objects Allow Simpler Design

Schema:
```
Person(SSN: String,
       Name: String,
       PhoneN: {String},
       Child: {SSN})
```

No need to decompose in order to eliminate redundancy: the set data type takes care of this.

Object 1:
```
( 111-22-3333,
   "Joe Public",
   {516-345-6789, 516-123-4567},
   {222-33-4444, 333-44-5555}
 )
```

Object 2:
```
( 222-33-4444,
   "Bob Public",
   {212-987-6543, 212-135-7924},
   {444-55-6666, 555-66-7777}
 )
```
Objects Allow Simpler Queries

Schema (slightly changed):
Person (SSN: String,
    Name: String,
    PhoneN: {String},
    Child: {Person})

- Because the type of Child is the set of Person-objects, it makes sense to continue querying the object attributes in a path expression.

Object-based query:
SELECT P.Child.Child.PhoneN
FROM Person P
WHERE P.Name = 'Joe Public'

- Much more natural!

ISA (or Class) Hierarchy

Person(SSN, Name)
Student(SSN,Major)

Query: Get the names of all computer science majors

Relational formulation:
SELECT P.Name
FROM Person P, Student S
WHERE P.SSN = S.SSN and S.Major = 'CS'

Object-based formulation:
SELECT S.Name
FROM Student S
WHERE S.Major = 'CS'

Student-objects are also Person-objects, so they inherit the attribute Name.
Object Methods in Queries

- Objects can have associated operations (methods), which can be used in queries. For instance, the method `frameRange(from, to)` might be a method in class `Movie`. Then the following query makes sense:

```sql
SELECT M.frameRange(20000, 50000)
FROM Movie M
WHERE M.Name = 'The Simpsons'
```

The “Impedance” Mismatch

- One cannot write a complete application in SQL, so SQL statements are embedded in a host language, like C or Java.
- **SQL**: Set-oriented, works with relations, uses high-level operations over them.
- **Host language**: Record-oriented, does not understand relations and high-level operations on them.
- **SQL**: Declarative.
- **Host language**: Procedural.
- *Embedding SQL in a host language involves ugly adaptors (cursors/iterators)* – a direct consequence of the above mismatch of properties between SQL and the host languages. It was dubbed “impedance” mismatch.
Can the Impedance Mismatch be Bridged?

- This was the original idea behind object databases:
  - Use an object-oriented language as a data manipulation language. Since data is stored in objects and the language manipulates objects, there will be no mismatch!

- Problems:
  - Object-oriented languages are procedural – the advantages of a high-level query language, such as SQL, are lost.
  - C++, Java, Smalltalk, etc., all have significantly different object modeling capabilities. Which ones should the database use? Can a Java application access data objects created by a C++ application?
  - Instead of one query language we end up with a bunch! (one for C++, one for Java, etc.)

Is Impedance Mismatch Really a Problem?

- Two main approaches/standards:
  - ODMG (Object Database Management Group):
    Impedance mismatch is worse than the ozone hole!
  - SQL:1999/2003:
    Couldn’t care less – SQL rules!

- We will discuss both approaches.
Object Databases vs. Relational Databases

- **Relational**: set of relations; relation = set of tuples
- **Object**: set of classes; class = set of objects

- **Relational**: tuple components are primitive (int, string)
- **Object**: object components can be complex types (sets, tuples, other objects)

- **Unique features of object databases**:
  - Inheritance hierarchy
  - Object methods
  - In some systems (ODMG), the host language and the data manipulation language are the same

The Conceptual Object Data Model (CODM)

- Plays the same role as the relational data model
- Provides a common view of the different approaches (ODMG, SQL:1999/2003)
- Close to the ODMG model, but is not burdened with confusing low-level details
Object Id (Oid)

- Every object has a unique Id: different objects have different Ids
- **Immutable**: does not change as the object changes

- Different from primary key!
  - Like a key, identifies an object uniquely
  - But key values can change – oids cannot

Objects and Values

- An object is a pair: (oid, value)
- Example: A Joe Public’s object

  (#32, [ SSN: 111-22-3333,
  Name: “Joe Public”,
  PhoneN: {“516-123-4567”, “516-345-6789”},
  Child: {#445, #73} ] )
Complex Values

- A value can be of one of the following forms:
  - **Primitive** value: an integer (e.g., 7), a string ("John"), a float (e.g., 23.45), a Boolean (e.g., false)
  - **Reference** value: An oid of an object, e.g., #445
  - **Tuple** value: \([A_1: v_1, ..., A_n: v_n]\)
    - \(A_1, ..., A_n\) = distinct attribute names
    - \(v_1, ..., v_n\) = values
  - **Set** value: \([v_1, ..., v_n]\)
    - \(v_1, ..., v_n\) = values

- Example: previous slide

Classes

- **Class**: set of semantically similar objects (e.g., people, students, cars, motorcycles)

- A class has:
  - **Type**: describes common structure of all objects in the class (semantically similar objects are also structurally similar)
  - **Method signatures**: declarations of the operations that can be applied to all objects in the class.
  - **Extent**: the set of all objects in the class

- Classes are organized in a class hierarchy
  - The extent of a class contains the extent of any of its subclasses
Complex Types: Intuition

• Data (relational or object) must be properly structured

• Complex data (objects) – complex types

Object: {#32, [ SSN: 111-22-3333,
  Name: “Joe Public”,
  PhoneN: {“516-123-4567”, “516-345-6789”},
  Child: {#445, #73} ] }

Its type: [ SSN: String,
  Name: String,
  PhoneN: {String},
  Child: {Person} ]

Complex Types: Definition

• A type is one of the following:
  • Basic types: String, Float, Integer, etc.
  • Reference types: user defined class names, e.g., Person, Automobile
  • Tuple types: [A₁: T₁, ..., Aₙ: Tₙ]
    • A₁, ..., Aₙ – distinct attribute names
    • T₁, ..., Tₙ – types
    • e.g., [SSN: String, Child: {Person}]
  • Set types: {T}, where T is a type
    • e.g., {String}, {Person}
Subtypes: Intuition

- A subtype has “more structure” than its supertype.

- Example: Student is a subtype of Person
  
  Person: [SSN: String, Name: String,  
            Address: [StNum: Integer, StName: String]]
  
  Student: [SSN: String, Name: String,  
            Address: [StNum: Integer, StName: String, Rm: Integer],  
            Majors: {String},  
            Enrolled: {Course}]

Subtypes: Definition

- $T$ is a subtype of supertype $T'$ if and only if $T \neq T'$ and one of the following conditions holds:
  
  - **Reference types:**
    - $T$ and $T'$ are reference types and $T$ is a subclass of $T'$
  
  - **Tuple types:**
    - $T = [A_1: T_1, ..., A_n: T_n, A_{n+1}: T_{n+1}, ..., A_m: T_m]$,  
      $T' = [A_1: T'_1, ..., A_n: T'_n]$  
    - are tuple types and for each $i = 1, ..., n$, either $T_i = T'_i$ or $T_i$ is a subtype of $T'_i$
  
  - **Set types:**
    - $T = \{T_0\}$ and $T' = \{T'_0\}$ are set types and $T_0$ is a subtype of $T'_0$
Domain of a Type

- \textit{domain}(T) is the set of all objects that conform to type \( T \). Namely:
  - \textit{domain}(\text{Integer}) = \text{set of all integers},
  - \textit{domain}(\text{String}) = \text{set of all strings}, etc.

- \textit{domain}(T), where \( T \) is reference type and is the extent of \( T \), i.e., oids of all objects in class \( T \)

- \textit{domain}([A_1: \ T_1, \ ..., \ A_n: \ T_n]) is the set of all tuple values of the form \([A_1: v_1, \ ..., \ A_n: v_n]\), where each \( v_i \in \text{domain}(T_i) \)

- \textit{domain}(\{\ T\}) is the set of all finite sets of the form \( \{w_1, \ ..., \ w_m\} \), where each \( w_i \in \text{domain}(T) \)

Database Schema

- For each class includes:
  - Type
  - Method signatures. E.g., the following signature could be in class Course:
    \[
    \text{Boolean enroll(Student)}
    \]
- The subclass relationship
- The integrity constraints (keys, foreign keys, etc.)
Database Instance

- *Set of extents* for each class in the schema
- Each *object in the extent of a class must have the type of that class*, i.e., it must belong to the domain of the type
- Each object in the database must have *unique oid*
- The extents *must satisfy the constraints* of the database schema

Object-Relational Data Model

- A straightforward subset of CODM: *only tuple types at the top level*
- More precisely:
  - Set of classes, where each class has a tuple type (the types of the tuple component can be anything)
  - *Each tuple is an object* of the form (oid, tuple-value)
- Pure relational data model:
  - Each class (relation) has a tuple type, **but**
  - The types of tuple components must be primitive
  - Oids are not explicitly part of the model – tuples are pure values
Objects in SQL:1999/2003

- Object-relational extension of SQL-92
  - Backward compatibility
- Includes the legacy relational model
- SQL:1999/2003 database = a finite set of relations
- relation =
  - a set of tuples (extends legacy relations) or
  - a set of objects (completely new)
- object = (oid, tuple-value)
- tuple = tuple-value in the form of [Attr₁: v₁, ..., Attrₙ: vₙ]

SQL:1999/2003 Tuple Values

- Tuple value: [Attr₁: v₁, ..., Attrₙ: vₙ]
- Attrᵢ are all distinct attribute names
- Each vᵢ is one of these:
  - Primitive value: a constant of type CHAR(...), INTEGER, FLOAT, etc.
  - Reference value: an object Id
  - Another tuple value (i.e., nested tuples)
  - A multiset value = {v₁, ..., vₙ} – can create with MULTISET introduced in SQL:2003.
    - ARRAY in SQL1999 – a fixed size array
    - e.g., WEEKDAYS VARCHAR(10) ARRAY[7]
Row Types

- The same as the original (legacy) relational tuple type. However:
  - Row types can now be the types of the individual attributes in a tuple
  - In the legacy relational model, tuples could occur only as top-level types

```sql
CREATE TABLE PERSON (
    Name CHAR(20),
    Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5))
)
```

Row Types (Contd.)

- Use path expressions to refer to the components of row types:
  ```sql
  SELECT P.Name
  FROM PERSON P
  WHERE P.Address.ZIP = '11794'
  ```

- Update operations:
  ```sql
  INSERT INTO PERSON(Name, Address)
  VALUES ('John Doe', ROW(666, 'Hollow Rd.', '66666'))

  UPDATE PERSON
  SET Address.ZIP = '66666'
  WHERE Address.ZIP = '55555'

  UPDATE PERSON
  SET Address = ROW(21, 'Main St', '12345')
  WHERE Address = ROW(123, 'Maple Dr.', '54321') AND Name = 'J. Public'
  ```
User Defined Types (UDT)

- UDTs allow specification of complex objects/tuples, methods, and their implementation
- Like ROW types, UDTs can be types of individual attributes in tuples
- UDTs can be much more complex than ROW types (even disregarding the methods): the components of UDTs do not need to be elementary types

A UDT Example

```
CREATE TYPE PersonType AS (
    Name CHAR(20),
    Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5))
);

CREATE TYPE StudentType UNDER PersonType AS (
    Id INTEGER,
    Status CHAR(2)
)
METHOD award_degree() RETURNS BOOLEAN;

CREATE METHOD award_degree() FOR StudentType LANGUAGE C EXTERNAL NAME 'file:/home/admin/award_degree';
```
Using UDTs in CREATE TABLE

- **As an attribute type:**

  ```sql
  CREATE TABLE TRANSCRIPT (  
    Student StudentType,  
    CrsCode CHAR(6),  
    Semester CHAR(6),  
    Grade CHAR(1)  
  )  
  
  - some attributes (Student) have “complex” type

- **As a table type:**

  ```sql
  CREATE TABLE STUDENT OF StudentType;  
  
  - Such a table is called *typed table.*  
  - each row is considered to be an object.

Objects

- **Only typed tables contain objects (i.e., tuples with oids)**

- **Compare:**

  ```sql
  CREATE TABLE STUDENT OF StudentType;  

  and

  CREATE TABLE STUDENT1 (  
    Name CHAR(20),  
    Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5)),  
    Id INTEGER,  
    Status CHAR(2)  
  )  
  
  - Both contain tuples of exactly the same structure  
  - Only the tuples in STUDENT – not STUDENT1 – have oids  
  - Will see later how to reference objects, create them, etc.
Querying UDTs

- Nothing special – just use path expressions

```sql
SELECT T.Student.Name, T.Grade
FROM TRANSCRIPT T
WHERE T.Student.Address.Street = 'Main St.'
```

Note: T.Student has the type StudentType. The attribute Name is not declared explicitly in StudentType, but is inherited from PersonType.

Updating User-Defined Types

- Inserting a record into TRANSCRIPT:

```sql
INSERT INTO TRANSCRIPT(Student,Course,Semester,Grade)
VALUES (????, 'CS308', '2000', 'A')
```

- The type of the Student attribute is StudentType. How does one insert a value of this type (in place of ????)?
- Further complication: the UDT StudentType is encapsulated, i.e., it is accessible only through public methods, which we did not define
  - Do it through the observer and mutator methods provided by the DBMS automatically
Observer Methods

- For each attribute $A$ of type $T$ in a UDT, an SQL:1999 DBMS is supposed to supply an observer method, $A: () \rightarrow T$, which returns the value of $A$ (the notation "()" means that the method takes no arguments).

- Observer methods for StudentType:
  - $Id: () \rightarrow INTEGER$
  - $Name: () \rightarrow CHAR(20)$
  - $Status: () \rightarrow CHAR(2)$
  - $Address: () \rightarrow ROW(INTEGER, CHAR(20), CHAR(5))$

- For example, in
  ```sql
  SELECT T.Student.Name, T.Grade
  FROM TRANSCRIPT T
  WHERE T.Student.Address.Street = 'Main St.'
  ```

Name and Address are observer methods, since T.Student is of type StudentType.

*Note:* Grade is not an observer, because TRANSCRIPT is not part of a UDT, but this is a conceptual distinction – syntactically there is no difference.

Mutator Methods

- An SQL DBMS is supposed to supply, for each attribute $A$ of type $T$ in a UDT $U$, a mutator method $A: T \rightarrow U$

- For any object $o$ of type $U$, it takes a value $t$ of type $T$ and replaces the old value of $o.A$ with $t$; it returns the new value of the object. Thus, $o.A(t)$ is an object of type $U$.

- Mutators for StudentType:
  - $Id: INTEGER \rightarrow StudentType$
  - $Name: CHAR(20) \rightarrow StudentType$
  - $Address: ROW(INTEGER, CHAR(20), CHAR(5)) \rightarrow StudentType$
Example: Inserting a UDT Value

```
INSERT INTO TRANSCRIPT(Student,Course,Semester,Grade)
VALUES (NEW StudentType(),
    .Id(111111111)
    .Status('G5')
    .Name('Joe Public')
    .Address(ROW(123,'Main St.','54321'))
    )
```

Create a blank `StudentType` object

Example: Changing a UDT Value

```
UPDATE TRANSCRIPT
SET    Student = Student
    .Address(ROW(21,'Maple St.','12345'))
    .Name('John Smith'),
    Grade = 'B'
WHERE   Student.Id = 111111111 AND CrsCode = 'CS532' AND Semester = 'S2002'
```

- Mutators are used to change the values of the attributes `Address` and `Name`
**Example Use of Method for Insertion**

```sql
ALTER TABLE StudentType
ADD METHOD StudentConstr(name CHAR(20), id INTEGER,
                          streetNumber INTEGER, ...)
RETURNS StudentType;

CREATE METHOD StudentConstr(name CHAR(20), id INTEGER, ...)
FOR StudentType
RETURNS StudentType
LANGUAGE SQL
BEGIN
    RETURN NEW StudentType()
    .Name(name)
    .Id(id)
    ..... 
END;
```

**Example Use of Method for Insertion (cont’d)**

```sql
INSERT INTO Transcript(Student, Course, Semester, Grade)
VALUES (StudentConstr('Joe Public', 1234, ...),
        'CSE532',
        'F2013',
        'A');
```
Referencing Objects

- Consider again
  ```sql
  CREATE TABLE TRANSCRIPT (
    Student StudentType,
    CrsCode CHAR(6),
    Semester CHAR(6),
    Grade CHAR(1)
  )
  ```

- **Problem:** TRANSCRIPT records for the same student refer to distinct values of type StudentType (even though the contents of these values may be the same) – a maintenance/consistency problem

- **Solution:** use self-referencing column (next slide)
  - Bad design, which distinguishes objects from their references
  - Not truly object-oriented

Self-Referencing Column

- **Every typed table has a self-referencing column**
  - Normally invisible
  - Contains explicit object Id for each tuple in the table
  - Can be given an explicit name – the only way to enable referencing of objects

```sql
CREATE TABLE STUDENT2 OF StudentType
REF IS stud_oid;
```

- Self-referencing columns can be used in queries just like regular columns. Their values cannot be changed, however
Reference Types and Self-Referencing Columns

- To reference objects, use:
  - self-referencing columns + reference types: $\text{REF(some-UDT)}$
    
    ```sql
    CREATE TABLE TRANSCRIPT1 (
        Student REF(StudentType) SCOPE STUDENT2,
        CrsCode CHAR(6),
        Semester CHAR(6),
        Grade CHAR(1)
    )
    ```

- Two issues:
  - How does one query the attributes of a reference type
  - How does one provide values for the attributes of type $\text{REF(...)}$
    - Remember: you can’t manufacture these values out of thin air – they are oids!

Querying Reference Types

- Recall: Student $\text{REF(StudentType)}$ $\text{SCOPE STUDENT2}$ in $\text{TRANSCRIPT1}$. How does one access, for example, student names?
  - SQL:1999 has the same misfeature as C/C++ has (and which Java and OQL do not have): it distinguishes between objects and references to objects. To pass through a boundary of $\text{REF(...)}$ use “$\Rightarrow$” instead of “.”

```sql
SELECT T.Student $\Rightarrow$ Name, T.Grade
FROM TRANSCRIPT1 T
WHERE T.Student $\Rightarrow$ Address.Street = "Main St."
```
Inserting REF Values

- How does one give values to REF attributes, like Student in TRANSCRIPT1?
  - Use explicit self-referencing columns, like stud_oid in STUDENT2

```
CREATE TABLE TRANSCRIPT1 {
  Student REF(StudentType) SCOPE STUDENT2,
  CrsCode CHAR(6),
  Semester CHAR(6),
  Grade CHAR(1)
}
```

- Example:
  - Creating a TRANSCRIPT1 record whose Student attribute has an object reference to an object in STUDENT2:

```
INSERT INTO TRANSCRIPT1(Student, Course, Semester, Grade)
SELECT S.stud_oid, 'HIS666', 'F1462', 'D'
FROM STUDENT2 S
WHERE S.Id = '111111111'
```

Explicit self-referential column of STUDENT2

Inheritance

- Assume Student be a table of type StudentType, and Person be a table of type PersonType.

```
CREATE TABLE STUDENT OF StudentType
```

- What should be done?
  - Will ‘A’ be automatically added to the relation Person?
    - No.
  - One can define several tables using the same UDT PersonType. SQL designers disagreed to automatic addition of tuple in Person
  - User can make Person as the supertable of Student by:

```
CREATE TABLE Student OF StudentType UNDER Person
```
Collection Data Types

- Set (multiset) data type was added in SQL:2003.

```
CREATE TYPE StudentType UNDER PersonType AS (
    Id INTEGER,
    Status CHAR(2),
    Enrolled REF(CourseType) MULTISET
)
```

A bunch of references to objects of type CourseType

Querying Collection Types

- For each student, list the Id, address, and the courses in which the student is enrolled (assume STUDENT is a table of type StudentType):

```
SELECT S.Id, S.Address, C.Name
FROM   STUDENT S, COURSE C
WHERE  C.CrsCode IN
    ( SELECT E.CrsCode
      FROM    UNNEST(S.Enrolled) E
    )
```

- Note: E is bound to tuples in a 1-column table (created by UNNEST) of object references
  - the reverse conversion (i.e., from 1-column table to MULTISET) is also possible by: MULTISET (SELECT colA FROM...)

Convert multiset to table
Querying Collection Types (cont’d)

CREATE TABLE Transcript1 (  
Student REF(StudentType),  
Course REF(CourseType),  
Semester CHAR(6),  
Grade CHAR(1));

INSERT INTO STUDENT (Id, Status , Enrolled)  
VALUES ( 123, ‘G4’,  
MULTISET (SELECT T.Course  
FROM Transcript1 T  
WHERE T.Student->Id = 123)  
);