CS 235: Introduction to Databases
Svetlozar Nestorov
Lecture Notes #16

Outline
• Features of object-relational model.
• Features of ODL.
• Object Query Language (OQL).

Methods
• Real reason object-relational isn't just nested structures in relations.
• We'll follow Oracle syntax.
• Declared in a CREATE TYPE statement, defined in a CREATE TYPE BODY statement.

Example
• Let's add a method priceInLevs to the MenuType and thus to the Sells relation.

```
CREATE TYPE MenuType AS OBJECT (
  bar REF BarType,
  beer REF BeerType,
  price FLOAT,
  MEMBER FUNCTION priceInLevs(
    rate IN FLOAT) RETURN FLOAT,
  PRAGMA RESTRICT_REFERENCES(priceInLevs, WINDS)
);
```

```sql
CREATE TYPE BODY MenuType AS
MEMBER FUNCTION priceInLevs(rate FLOAT)
  RETURN FLOAT IS
  BEGIN
    RETURN rate * SELF.price;
  END;
END;
```

CREATE TABLE Sells OF MenuType;

Example (contd.)
CREATE TYPE BODY MenuType AS
MEMBER FUNCTION priceInLevs(rate FLOAT)
  RETURN FLOAT IS
  BEGIN
    RETURN rate * SELF.price;
  END;
END;
```

CREATE TABLE Sells OF MenuType;
Some Points to Remember

- The pragma is needed to allow priceInLevs to be used in queries.
- WNDS = "write no database state."
- In the declaration, function/procedure arguments need a mode, IN, OUT, or IN OUT, just like PL/SQL procedures.
- But the mode does not appear in the definition.
- Many methods will take no arguments (relying on the built-in "self").
- In that case, do not use parentheses after the function name.
- The body can have any number of function declarations, separated by semicolons.

Example of Method Use

- Follow a designator for the object to which you want to apply the method by a dot, the name of the method, and arguments.
  
  ```sql
  SELECT ss.beer.name,
  ss.priceInLevs(2.1)
  FROM Sells ss
  WHERE ss.bar.name = 'Spoon';
  ```

Built-In Comparison Functions (SQL3)

- We can define for each ADT two functions EQUAL and LESSTHAN.
- Allow values of this ADT to participate in WHERE clauses involving =, <=, etc. and in ORDER-BY sorting.

Order Methods in Oracle 8

- We can declare one method for a type to be an ORDER method.
- Definition of this method must return <0, 0, >0, if "self" is less than, equal to, or greater than the argument object.
- Also used in comparisons for WHERE and ORDER BY.

Example

- Order BarType objects by name.
  ```sql
  CREATE TYPE BarType AS OBJECT
  (  
    name CHAR(20) UNIQUE,
    addr CHAR(20),
    ORDER MEMBER FUNCTION before(
    bar2 IN BarType) RETURN INT,
    PRAGMA RESTRICT_REFERENCES(before,
    WNDS,RNDS,WNPS,RNPS)
  );
  /
  ```

Example (contd.)

```sql
CREATE TYPE BODY BarType AS
ORDER MEMBER FUNCTION before(bar2 BarType)
RETURN INT IS
BEGIN
  IF SELF.name < bar2.name
  THEN RETURN -1;
  ELSIF SELF.name = bar2.name
  THEN RETURN 0;
  ELSE RETURN 1;
  END IF;
END;
END;
```
Pragmas

- The extra codes in the pragma guarantee no reading or writing of the database state or the “package state.”
- Options:
  - W N D S
  - R N D S
  - W N P S
  - R N P S

ODL Subclasses

- Follow name of subclass by colon and its superclass.
- Example: Ales are Beers with a Color
  ```java
  interface Ales:Beers {
    attribute string color;
  }
  ```
  - Objects of the Ales class acquire all the attributes and relationships of the Beers class.
- While E/R entities can have manifestations in a class and subclass, in ODL we assume each object is a member of exactly one class.

Keys in ODL

- Indicate with key(s) following the class name, and a list of attributes forming the key.
- Several lists may be used to indicate several alternative keys.
- Parentheses group members of a key, and also group key to the declared keys.
- Thus, \((\text{key}(a_1, a_2, \ldots, a_n))\) = “one key consisting of all \(n\) attributes.” \((\text{key} a_1, a_2, \ldots, a_n)\) = “each \(a_i\) is a key by itself.”

Example

```java
interface Beers (key name) {
  \{attribute string name \ldots\}
}
interface Courses
  (key (dept, number), (room, hours))
  \{...\}
```
- Keys are optional in ODL. The object ID suffices to distinguish objects that have the same values in their elements.

Translating ODL to Relations

1. Classes without relationships: like entity set, but several new problems arise.
2. Classes with relationships:
   a) Treat the relationship separately, as in E/R.
   b) Attach a many-one relationship to the relation for the many.

ODL Class Without Relationships

- Problem: ODL allows attribute types built from structures and collection types.
- Structure: Make one attribute for each field.
- Set: make one tuple for each member of the set.
  - More than one set attribute? Make tuples for all combinations.
- Problem: ODL class may have no key, but we should have one in the relation to represent “OID.”
Example: ODL

interface Drinkers (key name) {
  attribute string name;
  attribute Struct Addr
    {string street, string city, int zip} address;
  attribute Set<string> phone;
}

Example: Relations

<table>
<thead>
<tr>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>n1</td>
<td>s1</td>
<td>c1</td>
<td>z1</td>
<td>p1</td>
</tr>
<tr>
<td>n1</td>
<td>s1</td>
<td>c1</td>
<td>z1</td>
<td>p2</td>
</tr>
</tbody>
</table>

• Surprise: the key for the class (name) is not the key for the relation (name, phone).
• name in the class determines a unique object, including a set of phones.
• name in the relation does not determine a unique tuple.
• Since tuples are not identical to objects, there is no inconsistency!
• BCNF violation: separate out name-phone.

ODL Relationships

• If the relationship is many-one from A to B, put key of B attributes in the relation for class A.
• If relationship is many-many, we'll have to duplicate A-tuples as in ODL with set-valued attributes.
  • Wouldn't you really rather create a separate relation for a many-many-relationship?
  • You'll wind up separating it anyway, during BCNF decomposition.

Example: ODL

interface Drinkers (key name) {
  attribute string name;
  attribute string addr;
  relationship Set<Beers> likes
    inverse Beers::fans;
  relationship Beers favorite
    inverse Beers::realFans;
  relationship Drinkers husband
    inverse wife;
  relationship Drinkers wife
    inverse husband;
  relationship Set<Drinkers> buddies
    inverse buddies;
}

Example: Relations

Drinkers (name, addr, beerName, favBeer, wife, buddy)

• Not in BCNF; decompose to:
  Drinkers(name, addr, favBeer, wife)
  DrBeer(name, beer)
  DrBuddy(name, buddy)

Motivation for OQL

• Relational languages suffer from impedance mismatch when we try to connect them to conventional languages like C or C++.
  • The data models of C and SQL are radically different, e.g., C does not have relations, sets, or bags as primitive types; C is tuple-at-a-time, SQL is relation-at-a-time.
Object Query Language (OQL)

- OQL is an attempt by the OO community to extend languages like C++ with SQL-like, relation-at-a-time dictions.

OQL Types

- Basic types: strings, ints, reals, etc., plus class names.
- Type constructors:
  - Struct for structures.
  - Collection types: set, bag, list, array.
- Like ODL, but no limit on the number of times we can apply a type constructor.
- Set(Struct()) and Bag(Struct()) play special roles akin to relations.

OQL and ODL

- ODL is the Schema-Definition language for OQL.
- For every class we can declare an extent = name for the current set of objects of the class.
- Remember to refer to the extent, not the class name, in queries.

Example

```java
interface Bar (extent Bars)
{ attribute string name;
  attribute string addr;
  relationship Set<Sell> beersSold
  inverse Sell::bar;}

interface Beer (extent Beers)
{ attribute string name;
  attribute string manf;
  relationship Set<Sell> soldBy
  inverse Sell::beer;}
```

Example (contd.)

```java
interface Sell (extent Sells)
{ attribute float price;
  relationship Bar bar
  inverse Bar::beersSold;
  relationship Beer beer
  inverse Beer::soldBy;
}
```

Path Expressions

- Let \( x \) be an object of class \( C \).
- If \( a \) is an attribute of \( C \), then \( x.a \) = the value of \( a \) in the \( x \) object.
- If \( r \) is a relationship of \( C \), then \( x.r \) = the value to which \( x \) is connected by \( r \).
  - Could be an object or a collection of objects, depending on the type of \( r \).
- If \( m \) is a method of \( C \), then \( x.m(\cdot) \) is the result of applying \( m \) to \( x \).
Examples

- Let \( s \) be a variable whose type is \texttt{Sells}.
  - \( s\.\text{price} \) = the price in the object \( s \).
  - \( s\.\text{bar.addr} \) = the address of the bar mentioned in \( s \).
  - Note: cascade of dots OK because \( s\.\text{bar} \) is an \textit{object}, not a collection.
- Illegal use of dot:
  - \( b\.\text{beersSold}.\text{price} \), where \( b \) is a Bar object.
  - Illegal because \( b\.\text{beersSold} \) is a \textit{set} of objects, not a single object.

OQL Select-From-Where

\[
\begin{align*}
\text{SELECT} & \text{<list of values> } \\
\text{FROM} & \text{<list of collections and typical members>} \\
\text{WHERE} & \text{<condition>}
\end{align*}
\]

- Collections in \texttt{FROM} can be:
  1. Extents.
  2. Expressions that evaluate to a collection.
- Following a collection is a name for a typical member, optionally preceded by \texttt{AS}.

Example

- Get the menu at Spoon.

\[
\begin{align*}
\text{SELECT} & \text{ s\.beer.name, s.price } \\
\text{FROM} & \text{ Sells s } \\
\text{WHERE} & \text{ s\.bar.name = "Spoon"}
\end{align*}
\]
- Notice double-quoted strings in OQL.

Example

- Another way to get Spoon's menu, this time focusing on the Bar objects.

\[
\begin{align*}
\text{SELECT} & \text{ s\.beer.name, s.price } \\
\text{FROM} & \text{ Bars b, b\.beersSold s } \\
\text{WHERE} & \text{ b.name = "Spoon"}
\end{align*}
\]
- Notice that the typical object \( b \) in the first collection of \texttt{FROM} is used to help define the second collection.

Typical Usage

- If \( x \) is an object, you can extend the path expression, like \( s \) or \( s\.\text{beer} \) in \( s\.\text{beer.name} \).
- If \( x \) is a collection, you use it in the \texttt{FROM} list, like \( b\.\text{beersSold} \) above, if you want to access attributes of \( x \).

Tailoring the Type of the Result

- Default: bag of structs, field names taken from the ends of path names in \texttt{SELECT} clause.
- Example

\[
\begin{align*}
\text{SELECT} & \text{ s\.beer.name, s.price } \\
\text{FROM} & \text{ Bars b, b\.beersSold } \\
\text{WHERE} & \text{ b.name = "Spoon"}
\end{align*}
\]
has result type:
\[
\text{Bag}((\text{Struct(}
\text{name: string,}
\text{price: real)})
\text{)}
\]
### Rename Fields
- Prefix the path with the desired name and a colon.
  ```sql
  SELECT beer: s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Spoon"
  ```
  Has type:
  ```
  Bag[Struct(
    beer: string,
    price: real
  )]
  ```

### Change the Collection Type
- Use `SELECT DISTINCT` to get a set of structs.
  ```sql
  SELECT DISTINCT s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Spoon"
  ```
- Use `ORDER BY` clause to get a list of structs.

### Example
```sql
ejoeMenu =
  SELECT s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Spoon"
  ORDER BY s.price ASC;
```
- `ASC = ascending (default); DESC = descending.`
- We can extract from a list as if it were an array, e.g.,
  ```
  cheapest = joeMenu[1].name;
  ```