Object-Oriented Database Languages

Object Description Language

Object Query Language

Object-Oriented DBMS's

- Standards group: ODMG = Object Data Management Group.
- ODL = Object Description Language, like CREATE TABLE part of SQL.
- OQL = Object Query Language, tries to imitate SQL in an OO framework.

Framework --- 1

- ODMG imagines OO-DBMS vendors implementing an OO language like C++ with extensions (OQL) that allow the programmer to transfer data between the database and “host language” seamlessly.

Framework --- 2

- ODL is used to define *persistent* classes, those whose objects may be stored permanently in the database.
  - ODL classes look like Entity sets with binary relationships, plus methods.
  - ODL class definitions are part of the extended, OO host language.

ODL Overview

- A class declaration includes:
  1. A name for the class.
  2. Optional key declaration(s).
  3. *Extent* declaration = name for the set of currently existing objects of the class.
  4. Element declarations. An *element* is either an attribute, a relationship, or a method.

Class Definitions

```
class <name> {
  <list of element declarations, separated
     by semicolons>
}
```

Attribute and Relationship Declarations

- Attributes are (usually) elements with a type that does not involve classes.

  attribute <type> <name>;

- Relationships connect an object to one or more other objects of one class.

  relationship <type> <name>

  inverse <relationship>;

Inverse Relationships

- Suppose class *C* has a relationship *R* to class *D*.
- Then class *D* must have some relationship *S* to class *C*.
- *R* and *S* must be true inverses.
  - If object *d* is related to object *c* by *R*, then *c* must be related to *d* by *S*.

Example: Attributes and Relationships

```
class Bar {
   attribute string name;
   attribute string addr;
   relationship [illegible] serves inverse Beer::servedAt;
}
class Beer {
   attribute string name;
   attribute string manf;
   relationship Set<Bar> servedAt inverse Bar::serves;
}
```

The type of relationship serves is a set of Beer objects.

The :: operator connects a name on the right to the context containing that name, on the left.

Types of Relationships

- The type of a relationship is either
  1. A class, like Bar. If so, an object with this relationship can be connected to only one Bar object.
  2. Set<Bar>: the object is connected to a set of Bar objects.
  3. Bag<Bar>, List<Bar>, Array<Bar>: the object is connected to a bag, list, or array of Bar objects.

Multiplicity of Relationships

- All ODL relationships are binary.
- Many-many relationships have Set<…> for the type of the relationship and its inverse.
- Many-one relationships have Set<…> in the relationship of the "one" and just the class for the relationship of the "many."
- One-one relationships have classes as the type in both directions.

Example: Multiplicity

class Drinker { ...

relationship likes inverse Beer::fans;

relationship favBeer inverse Beer::superfans;

}

class Beer { ...

relationship fans inverse Drinker::likes;

relationship superfans inverse Drinker::favBeer;

}

Many-many uses Set<…> in both directions.

Many-one uses Set<…> only with the "one."

Another Multiplicity Example

husband and wife are one-one and inverses of each other.

```
class Drinker {
   attribute ... ;
   relationship Drinker
   relationship Drinker
   relationship Set<Drinker> buddies
      ;
}
```

buddies is many-many and its own inverse. Note no :: needed if the inverse is in the same class.

Coping With Multiway Relationships

- ODL does not support 3-way or higher relationships.
- We may simulate multiway relationships by a “connecting” class, whose objects represent tuples of objects we would like to connect by the multiway relationship.

Connecting Classes

- Suppose we want to connect classes X, Y, and Z by a relationship R.
- Devise a class C, whose objects represent a triple of objects (x, y, z) from classes X, Y, and Z, respectively.
- We need three many-one relationships from (x, y, z) to each of x, y, and z.

Example: Connecting Class

- Suppose we have Bar and Beer classes, and we want to represent the price at which each Bar sells each beer.
  - A many-many relationship between Bar and Beer cannot have a price attribute as it did in the E/R model.
- One solution: create class Price and a connecting class BBP to represent a related bar, beer, and price.

Example, Continued

- Since Price objects are just numbers, a better solution is to:
  1. Give BBP objects an attribute price.
  2. Use two many-one relationships between a BBP object and the Bar and Beer objects it represents.

Example, Concluded

- Here is the definition of BBP:

```
class BBP {
    attribute price:real;
    relationship Bar theBar inverse Bar::toBBP;
    relationship Beer theBeer inverse Beer::toBBP;
}
```

- Bar and Beer must be modified to include relationships, both called toBBP, and both of type Set<BBP>.

Structs and Enums

- Attributes can have a structure (as in C) or be an enumeration.
- Declare with

attribute [Struct or Enum] <name of struct or enum> { <details> } <name of attribute>;

- Details are field names and types for a Struct, a list of constants for an Enum.

Example: Struct and Enum

```
class Bar {
   attribute string name;
   attribute Struct
        {string street, string city, int zip}            ;
   attribute Enum
        { FULL, BEER, NONE }            ;
   relationship ...
}
```

Names for the structure and enumeration

names of the attributes

Reuse of Structs and Enums

- We can refer to the name of a Struct or Enum in another class definition.
  - Use the :: operator to indicate source class.
- Example:

```
class Drinker {
  attribute string name;
  attribute Struct            address; ...
```

Use the same street-city-zip structure here.

Method Declarations

- A class definition may include declarations of methods for the class.
- Information consists of:
  1. Return type, if any.
  2. Method name.
  3. Argument modes and types (no names).
     - Modes are in, out, and inout.
  4. Any exceptions the method may raise.

Example: Methods

```
real gpa(in string)raises(noGrades);
```

1. The method gpa returns a real number (presumably a student's GPA).
2. gpa takes one argument, a string (presumably the name of the student) and does not modify its argument.
3. gpa may raise the exception noGrades.

The ODL Type System

- Basic types: int, real/float, string, enumerated types, and classes.
- Type constructors:
  - Struct for structures.
  - *Collection types* : Set, Bag, List, Array, and Dictionary (= mapping from a domain type to a range type).
- Relationship types can only be a class or a single collection type applied to a class.

ODL Subclasses

- Usual object-oriented subclasses.
- Indicate superclass with a colon and its name.
- Subclass lists only the properties unique to it.
  - Also inherits its superclass' properties.

Example: Subclasses

- Ales are a subclass of beers:

```
class Ale:Beer {
  attribute string color;
}
```

ODL Keys

- You can declare any number of keys for a class.
- After the class name, add:

  (key <list of keys>)
- A key consisting of more than one attribute needs additional parentheses around those attributes.

Example: Keys

```
class Beer (key name) { …
```

- name is the key for beers.

```
class Course (key
  (dept,number),(room, hours)){
```

- dept and number form one key; so do room and hours.

Extents

- For each class there is an *extent*, the set of existing objects of that class.
  - Think of the extent as the one relation with that class as its schema.
- Indicate the extent after the class name, along with keys, as:

  (extent <extent name> ...)

Example: Extents

```
class Beer
  (extent Beers key name) { …
}
```

- Conventionally, we'll use singular for class names, plural for the corresponding extent.

OQL

- OQL is the object-oriented query standard.
- It uses ODL as its schema definition language.
- Types in OQL are like ODL's.
- Set(Struct) and Bag(Struct) play the role of relations.

Path Expressions

- Let x be an object of class C.

1. If a is an attribute of C, then $x.a$ is the value of that attribute.
2. If r is a relationship of C, then $x.r$ is the value to which x is connected by r.
    - Could be an object or a set of objects, depending on the type of r.
3. If m is a method of C, then $x.m$ (...) is the result of applying m to x.

Running Example

```
class Sell (extent Sells) {
   attribute real price;
   relationship Bar bar inverse Bar::beersSold;
   relationship Beer beer inverse Beers::soldBy;
}
class Bar (extent Bars) {
   attribute string name;
   attribute string addr;
   relationship Set<Sell> beersSold inverse Sell::bar;
}
```

Running Example, Concluded

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

Example: Path Expressions

- Let *s* be a variable of type Sell, i.e., a bar-beer-price object.
  1. s.price = the *price* in object *s*.
  2. s.bar.addr = the address of the bar we reach by following the *bar* relationship in *s*.
     - Note the cascade of dots is OK here, because s.bar is an object, not a collection of objects.

Example: Illegal Use of Dot

- We cannot apply the dot with a collection on the left --- only with a single object.
- Example (illegal), with *b* a Bar object:

.price

This expression is a set of Sell objects.
It does not have a price.

OQL Select-From-Where

- We may compute relation-like collections by an OQL statement:

SELECT <list of values>

FROM <list of collections and names for typical members>

WHERE <condition>

FROM clauses

- Each term of the FROM clause is:

<collection> <member name>

- A collection can be:
  1. The extent of some class.
  2. An expression that evaluates to a collection, e.g., certain path expressions like b.beersSold .

Example

- Get the menu at Joe's Bar.

```
SELECT s.beer.name, s.price
FROM
WHERE s.bar.name = "Joe's Bar"
```

Sells is the extent representing all Sell objects; s represents each Sell object, in turn.

Legal expressions. s.beer is a beer object and s.bar is a Bar object.

Notice OQL uses double-quotes.

Another Example

- This query also gets Joe's menu:

  SELECT s.beer.name, s.price

  FROM Bars b,

  WHERE b.name = "Joe's Bar"

b.beersSold is a set of Sell objects,
and s is now a typical sell object
that involves Joe's Bar.

Trick For Using Path Expressions

- If a path expression denotes an object, you can extend it with another dot and a property of that object.
  - Example: s, s.bar, s.bar.name .
- If a path expression denotes a collection of objects, you cannot extend it, but you can use it in the FROM clause.
  - Example: b.beersSold .

The Result Type

- As a default, the type of the result of select-from-where is a Bag of Structs.
  - Struct has one field for each term in the SELECT clause. Its name and type are taken from the last name in the path expression.
- If SELECT has only one term, technically the result is a one-field struct.
  - But a one-field struct is identified with the element itself.

Example: Result Type

```
SELECT s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = “Joe’s Bar”
```

- Has type:

Bag(Struct(name: string, price: real))

Renaming Fields

- To change a field name, precede that term by the name and a colon.
- Example:

```
SELECT beer: s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = “Joe’s Bar”
```

- Result type is

  Bag(Struct(beer: string, price: real)).

Producing a Set of Structs

- Add DISTINCT after SELECT to make the result type a set, and eliminate duplicates.
- Example:

```
SELECT DISTINCT s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = “Joe’s Bar”
```

- Result type is

  Set(Struct(name: string, price: string))

Producing a List of Structs

- Use an ORDER BY clause, as in SQL to make the result a list of structs, ordered by whichever fields are listed in the ORDER BY clause.
  - Ascending (ASC) is the default; descending (DESC) is an option.
- Access list elements by index [1], [2],...
- Gives capability similar to SQL cursors.

Example: Lists

- Let joeMenu be a host-language variable of type

  List(Struct(name:string, price:real))

```
joeMenu =
  SELECT s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = “Joe’s Bar”
  ORDER BY s.price;
```

Example, Continued

- Now, joeMenu has a value that is a list of structs, with name and price pairs for all the beers Joe sells.
- We can find the first (lowest price) element on the list by joeMenu[1], the next by joeMenu[2], and so on.
- Example: the name of Joe's cheapest beer:

```
cheapest = joeMenu[1].name;
```

Example, Concluded

- After evaluating joeMenu, we can print Joe’s menu by code like:

```
cout << “Beer\tPrice\n\n”;
```

```
for (i=1; i<=                ); i++)
  cout << joeMenu[i].name << “\t”
  << joeMenu[i].price << “\n”;
```

COUNT gives the
number of members
in any collection.

Subqueries

- A select-from-where expression can be surrounded by parentheses and used as a subquery in several ways, such as:
  1. In a FROM clause, as a collection.
  2. In EXISTS and FOR ALL expressions.

Example: Subquery in FROM

- Find the manufacturers of beers sold at Joe's:

SELECT DISTINCT b.manf

FROM (

)

Bag of Beer objects for the beers sold by Joe

Technically a one-field struct containing a Beer object, but identified with that object itself.

Quantifiers

- Two boolean-valued expressions for use in WHERE clauses:

FOR ALL *x* IN <collection> : <condition>

EXISTS *x* IN <collection> : <condition>

- True if and only if all members (resp. at least one member) of the collection satisfy the condition.

Example: EXISTS

- Find all names of bars that sell at least one beer for more than $5.

  SELECT b.name FROM Bars b

  WHERE

At least one Sell object for bar b has a price above $5.

Another Quantifier Example

- Find the names of all bars such that the only beers they sell for more than $5 are manufactured by Pete's.

SELECT b.name FROM Bars b

WHERE FOR ALL be IN (

) :

Bag of Beer objects (inside structs) for all beers sold by bar b for more than $5.

One-field structs are unwrapped automatically, so *be* may be thought of as a Beer object.

Simple Coercions

- As we saw, a one-field struct is automatically converted to the value of the one field.
  - Struct(f : x) coerces to x.
- A collection of one element can be coerced to that element, but we need the operator ELEMENT.
  - E.g., ELEMENT(Bag(x)) = x.

Example: ELEMENT

- Assign to variable *p* of type real, the price Joe charges for Bud:

p = ELEMENT(

);

Bag with one element, a Struct with field price and value = the price Joe charges for Bud.

Aggregations

- AVG, SUM, MIN, MAX, and COUNT apply to any collection where they make sense.
- Example: Find and assign to *x* the average price of beer at Joe's:

x = AVG(

);

Bag of structs with the prices for the beers Joe sells.

Grouping

- Recall SQL grouping:
  1. Groups of tuples based on the values of certain (grouping) attributes.
  2. SELECT clause can extract from a group only items that make sense:
     - Aggregations within a group.
     - Grouping attributes, whose value is a constant within the group.

OQL Grouping

- OQL extends the grouping idea in several ways:
  1. Any collection may be partitioned into groups.
  2. Groups may be based on any function(s) of the objects in the initial collection.
  3. Result of the query can be any function of the groups.

Outline of OQL GROUP BY

Initial collection defined by FROM, WHERE

↓ Group by values of function(s)

Intermediate collection, with function values and partition

↓ Terms from SELECT clause

Output collection

Example: GROUP BY

- We’ll work through these concepts using an example: “Find the average price of beer at each bar.”

```
SELECT barName, avgPrice: AVG(
  SELECT p.s.price FROM partition p)
FROM Sells s
GROUP BY barName: s.bar.name
```

Initial Collection

- Based on FROM and WHERE (which is missing): FROM Sells s
- The initial collection is a Bag of structs with one field for each "typical element" in the FROM clause.
- Here, a bag of structs of the form Struct(s: *obj*), where *obj* is a Sell object.

Intermediate Collection

- In general, bag of structs with one component for each function in the GROUP BY clause, plus one component always called *partition*.
- The partition value is the set of all objects in the initial collection that belong to the group represented by this struct.

Example: Intermediate Collection

SELECT barName, avgPrice: AVG(

SELECT p.s.price FROM partition p)

FROM Sells s

GROUP BY

One grouping function. Name is barName, type is string. Intermediate collection is a set of structs with fields barName: string and partition: Set<Struct{s: Sell}>

Example: Typical Member

- A typical member of the intermediate collection in our example is:

Struct(barName = “Joe’s Bar”,

partition = $\{s_1, s_2, \ldots, s_n\}$)

- Each member of partition is a Sell object s_i, for which s_i.bar.name is “Joe’s Bar”.

The Output Collection

- The output collection is computed by the SELECT clause, as usual.
- Without a GROUP BY clause, the SELECT clause gets the initial collection from which to produce its output.
- With GROUP BY, the SELECT clause is computed from the intermediate collection.

Example: Output Collection

SELECT [illegible], avgPrice: AVG(
SELECT p.s.price FROM partition p)

Extract the barName field from a group's struct.

From each member *p* of the group's partition, get the field *s* (the Sell object), and from that object extract the price.

Average these prices to create the value of field avgPrice in the structs of the output collection.

Typical output struct:
Struct(barName = "Joe's Bar",
AvgPrice = 2.83)

A Less Typical Example

- Find for each beer, the number of bars that charge a “low” price (< $2) and a “high” price (> $4) for that beer.
- Strategy --- group by three values:
  1. The beer name.
  2. A boolean function that is TRUE if and only if the price is low.
  3. A boolean function that is TRUE if and only if the price is high.

The Query

SELECT beerName, low, high,

  count: COUNT(partition)

FROM

GROUP BY beerName: b.name,

  low: s.price < 2.00, high: s.price > 4.00

Initial collection: structs of the form
Struct(b: Beer object, s: Sell object),
where s.beer = b.

The Intermediate Collection

- A set of structs with four fields:
  1. beerName: string
  2. low: boolean
  3. high: boolean
  4. partition: Set<Struct{b: Beer, s: Sell}>

Typical Structs in the Intermediate Collection

beerName	low	high	partition
Bud	TRUE	FALSE	S_{low}
Bud	FALSE	TRUE	S_{high}
Bud	FALSE	FALSE	S_{mid}

- S_{low} , etc., are sets of Beer-Sell pairs.
- Note low and high cannot both be true; their groups are always empty.

The Output Collection

```
SELECT beerName, low, high,
  count: COUNT(partition)
```

- Copy the first three components of each intermediate struct, and count the number of pairs in its partition, e.g.:

beerName	low	high	count
Bud	TRUE	FALSE	27