INF212 – Database Theory

Object-Oriented Query Languages: Object Query Language (OQL)

Edited by N. Akkøk spring 2003 and spring 2005
from Pål Halvorsen’s slides (spring 2002)

Also contains slides made by
Arthur M. Keller, Vera Goebel, Ragnar Normann
Overview

- OQL
  - Queries/sub-queries
  - Return types
  - Quantifiers
  - Object creation
  - Aggregation
  - Using host languages
  - Operators on set or bag objects
  - Grouping with properties
Object Query Language (OQL)

♣ Motivation:

♣ 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

♣ OQL is the query language in the ODMG standard

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

♣ Like SQL, OQL is a declarative (not procedural) language
OQL uses ODL

OQL is designed to operate on data described in ODL:

♣ 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.
OQL: Object- and Value-Equality

◆ Two objects of the same type (instances of the same class) cannot be equal, but they may have the same values.

◆ Example: Object $O_1$ and $O_2$ are instances of the same class.
  - The OQL expression $O_1 = O_2$ will always be FALSE.
  - The OQL expression $*O_1 = *O_2$ can be TRUE if the two objects have the same state, i.e., same value of all attributes.
OQL: Computations

♦ Mutable objects are manipulated by executing defined methods for this class

♦ Select in OQL may have side effects, i.e., it can change the state in the database (OQL does not have an own update function in contrast to SQL)

♦ Methods are called by navigating along paths; there is no difference for addressing of attributes, relationships, or methods.
OQL: Types

- **Basic types:** string, integer, float, boolean, character, enumerations, etc.

- **Type constructors:**
  - **Struct** for structures.
  - **Collection types:** set, bag, list, array.
    (NOTE: dictionary is not supported)
  - **Set(Struct()) and Bag(Struct())** play special roles akin to relations.
OQL: Path Expressions

🔹 We access components using dot-notations

🔹 Let \( x \) be an object of class \( C \):
  
 🔹 If \( a \) is an attribute of \( C \), then \( x.a \) is the value of \( a \) in the \( x \) object.
  
 🔹 If \( r \) is a relationship of \( C \), then \( x.r \) is the value to which \( x \) is connected by \( r \), i.e., could be an object or a collection of objects, depending on the type of \( r \)
  
 🔹 If \( m \) is a method of \( C \), then \( x.m(\cdots) \) is the result of applying \( m \) to \( x \).

🔹 We can form expressions with several dots (only last element may be a collection)

🔹 OQL allows arrows as a synonym for the dot, i.e., \( x\rightarrow a \) is equal to \( x.a \), opposed to for example in C
class Bar (extent Bars)
{  attribute string name;
    attribute string addr;
    relationship Set<Sell> beersSold inverse Sell::bar;
}

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

class Sell (extent Sells)
{  attribute float price;
    relationship Bar bar inverse Bar::beersSold;
    relationship Beer beer inverse Beer::soldBy;
    void raise_price(float price);
}
OQL:
Path Expressions for BBS Example

♦ Let $s$ be a variable whose type is $Sell$
  ♦ $s.price$ is the price in the object $s$ (the beer sold in this bar)
  ♦ $s.raise_price(x)$ raises the price of $s.beer$ in $s.bar$ with $x$
  ♦ $s.bar$ is a pointer to the bar mentioned in $s$
  ♦ $s.bar.addr$ is the address of the bar mentioned in $s$
    Note: cascade of dots OK because $s.bar$ is an object, not a collection

♦ Let $b$ be a variable whose type is $Bar$
  ♦ $b.name$ is the name of the bar
  ♦ $b.beersSold$ is a set of beers that this bar sells (set of pointers to $Sell$)
  ♦ *Illegal* use of path expressions: $b.beersSold.price$
    Note: illegal because $b.beersSold$ is a *set* of objects, not a single object

♦ Typical Usage:
  ♦ If $x$ is an object, you can extend the path expression, like $s$ is extended with $s.beer$ and $s.beer.name$ above
  ♦ If $x$ is a collection, like $b.beersSold$ above, it can be used anywhere a collection is appropriate (e.g., FROM), if you want to access attributes of $x$. 
OQL: Select-From-Where

♣ Similar to SQL syntax:
   SELECT <list of values>
   FROM <list of collections and typical members>
   WHERE <condition>

♣ Collections in 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 the keyword AS

♣ Note: there may be several different queries giving the same answer
OQL BBS Example: Select-From-Where

・ Get menu at “Joe’s” focusing on Sells objects:
  SELECT s.beer.name, s.price
  FROM Sells s
  WHERE s.bar.name = "Joe's"

・ Notice double-quoted strings in OQL (SQL has single-quoted)

・ Get “Joe’s” menu, this time focusing on the Bar objects:
  SELECT s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Joe's"

・ Notice that the typical object b in the first collection of FROM
  is used to help define the second collection.
OQL: Comparison Operators

Values can generally be compared using operators:

- `=` : equality
- `!=` : different form
- `<` : less than
- `>` : greater than
- `<=` : less or equal
- `>=` : greater or equal

Additional text comparison operators

- `IN` checks if a character is in a text string: `<c> IN <text>`
- `LIKE` checks if two texts are equal: `<text_1> LIKE <text_2>`
  
  `<text_2>` may contain special characters:
  
  - `_` or `?` : one arbitrary character
  - `*` or `%` : any arbitrary text string
OQL,BBS Example: Comparison Operators

Example: find name and price of all beers at “Joe’s” starting with “B” and consisting of the text string “ud”

```
SELECT s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = "Joe's" AND
      s.beer.name LIKE "B*" AND
      s.beer.name LIKE "*ud*"
```

NOTE 1: The name of the bar is equal to “Joe’s”

NOTE 2: The beer name starts with “B” followed by arbitrary characters

NOTE 3: The beer name contains “ud” starting with and followed by arbitrary characters
OQL: Quantifiers

♦ We can test whether all members, at least one member, some members, etc. satisfy some condition

♦ Boolean-valued expressions for use in WHERE-clauses.

  All:     FOR ALL x IN <collection> : <condition>
  At least one: EXISTS x IN <collection> : <condition>
  Only one:  UNIQUE x
  Some/any:  <collection>  <comparison> SOME/ANY <condition>
  where <comparison> = <, >, <=, >=, or =

♦ The expression has value TRUE if the condition is true

♦ NOT reverses the boolean value
OQL BBS Example: Quantifiers - I

♣ Example:
Find all bars that sell some beer for more than $5

SELECT b.name
FROM Bars b
WHERE EXISTS s IN b.beersSold : s.price > 5.00

♣ Example:
How would you find the bars that only sold beers for more than $5?

SELECT b.name
FROM Bars b
WHERE FOR ALL s IN b.beersSold : s.price > 5.00
Example:
Find the 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
    ( SELECT s.beer
        FROM b.beersSold s
        WHERE s.price > 5.00 ) :
    be.manf = "Pete's"
```

**NOTE 1:**
find all beers in a bar where the price is more than $5

**NOTE 2:**
all these “expensive” beers must be manufactured by “Pete’s”
OQL: Type of the Result

♣ Default: *bag* of structs, field names taken from the ends of path names in `SELECT` clause.

♣ Example: menu at “Joe’s”:

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

has result type:

```
Bag(Struct(name: string, price: real))
```
OQL: Rename Fields

♣ The result type
   \[
   \text{Bag(Struct(name: string, price: real))}
   \]
   may not have appropriate names for the results’ attributes

♣ Rename by prefixing the path with the desired name and a colon

♣ Example: rename attributes of the menu at “Joe’s”:
   \[
   \text{SELECT beername: s.beer.name, s.price} \\
   \text{FROM Bars b, b.beersSold s} \\
   \text{WHERE b.name = "Joe's"}
   \]
   has type:
   \[
   \text{Bag(Struct(beername: string, price: real))}
   \]
A *bag* of structs (default) returned by the SFW-statement is not always appropriate.

* Use `SELECT DISTINCT` to get a *set* of structs.

Example:

```
SELECT DISTINCT s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = "Joe's"
```
OQL: Change the Collection Type - II

♦ Use `ORDER BY` clause to get a *list* of structs

♦ Example:

```
joeMenu = SELECT s.beer.name, s.price
    FROM Bars b, b.beersSold s
    WHERE b.name = "Joe's"
    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_beer = joeMenu[0].name;
```
OQL: Subqueries

- Used where the result can be a collection type is appropriate, i.e., mainly
  - in FROM clauses and
  - with quantifiers like EXISTS, FOR ALL, etc.

- Example: subquery in FROM:
  Find the manufacturers of the beers served at "Joe's"
  
  ```sql
  SELECT DISTINCT b.manf
  FROM ( SELECT s.beer
          FROM Sells s
          WHERE s.bar.name = "Joe's"
          ) b
  ```
**OQL:**

Assigning Values to Host–Language Variables

- Unlike SQL, which needs to move data between tuples and variables, OQL fits naturally into a host language.
- Select-From-Where produces collections of objects.
- It is possible to assign any variable of proper type a value that is a result from OQL expressions.

**Example (C++ like):**

```
Name of bars that *only* sold beers for more than $5

Set<string> expensive_bars;
expensive_bars = SELECT DISTINCT b.name
              FROM Bars b
              WHERE FOR ALL s IN b.beersSold :
                      s.price > 5.00
```
OQL: Extraction of Collection Elements – I

♣ A collection with a single member:
Extract the member with ELEMENT.

♣ Example:
Find the price “Joe’s” charges for “Bud” and put the result in a variable \( p \):

\[
p = \text{ELEMENT}( \text{SELECT s.price} \\
\quad \text{FROM Sells s} \\
\quad \text{WHERE s.bar.name = "Joe's" AND} \\
\quad \text{s.beer.name = "Bud"})
\]
OQL: Extraction of Collection Elements – II

extracting all elements of a collection, one at a time:
1. Turn the collection into a list.
2. Extract elements of a list with <list_name>[i]

Example (C-like):
Print Joe's menu, in order of price, with beers of the same price listed alphabetically

L = SELECT s.beer.name, s.price
   FROM Sells s
   WHERE s.bar.name = "Joe's"
   ORDER BY s.price, s.beer.name;

printf("Beer\tPrice\n\n");

for( i=0; i<=COUNT(L); i++)
   printf("%s\t%f\n", L[i].name, L[i].price);

NOTE 1: make a list

NOTE 2: The ith element in L is obtained from L[i-1]. The index i starts at 0
OQL: Creating New Objects

- A Select-From-Where statement allows us to create new objects whose type is defined in by the types returned in the SELECT statement.

- Example:
  ```sql
  SELECT beername: s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Joe's Bar"
  ```

- **NOTE:** Defines a new object: `Bag<Struct( beername: string, price: integer)>

- *Constructor functions*: create new instances of a class or other defined type (details depend on host language).

- Example: insert a new beer
  ```python
  newBeer = Beer(name: "XXX",
                 manufacturer: "YYY")
  ```

  Effects:
  - Create a new `Beer` object, which becomes part of the extent `Beers`
  - The value of the host language variable `newBeer` is this object
OQL: Aggregation

♣ The five operators $\text{avg}$, $\text{sum}$, $\text{min}$, $\text{max}$, and $\text{count}$ apply to any collection, as long as the operators make sense for the element type.

♣ Example:
Find the average price of beer at Joe’s.

$$x = \text{AVG}(\text{SELECT s.price}$$
$$\text{FROM Sells s}$$
$$\text{WHERE s.bar.name = "Joe's"});$$

♣ Note: result of SELECT is technically a bag of 1-field structs, which is identified with the bag of the values of that field.
OQL: Union, Intersection, and Difference

- We may apply **union**, **intersection**, and **difference** operators on any objects of **Set** or **Bag** type.
- Use keywords **UNION**, **INTERSECT**, and **EXCEPT**, respectively.
- Result type is a **Bag** if one object is of type **Bag**; **Set** otherwise.
- Example:
  Find the name of all beers served at “Joe’s” that are not served at “Steve’s”

```sql
( SELECT s.beer.name  
  FROM Sells s  
  WHERE s.bar.name = "Joe's" )  
EXCEPT  
( SELECT s.beer.name  
  FROM Sells s  
  WHERE s.bar.name = "Steve's" )
```

**NOTE 1:** find all beers served at “Joe’s”

**NOTE 2:** find all beers served at “Steve’s”

**NOTE 3:** remove beers served at “Steve’s” for beers served at “Joe’s”
OQL supports grouping similar to SQL - some differences

Example in SQL: find average price of beers in all bars

```
SELECT bar.name, AVG(price)
FROM Sells
GROUP BY bar;
```

Is the `bar` value the “name” of the group, or the common value for the `bar` component of all tuples in the group?

In SQL it doesn't matter, but in OQL, you can create groups from the values of any function(s), not just attributes.

Thus, groups are identified by common values, not “name.”

Example: group by first letter of bar names (method needed).
OQL: Grouping – II

♦ General form:

\[
\text{GROUP BY } f_1: e_1, f_2: e_2, \ldots, f_n: e_n
\]

♦ Thus, made by the OQL clause:

♦ Keywords \text{GROUP BY}

♦ Comma separated list of partition attributes:

♦ name
♦ colon, and
♦ expression

♦ Example:

\[
\text{SELECT } \ldots
\text{FROM } \ldots
\text{GROUP BY } \text{barName: s.bar.name}
\]
OQL: Grouping Outline

- **Initial collection**: defined by FROM, WHERE

- **Intermediate collection**: with function values and partition

- **Output collection**

**NOTE 1:**
the selected objects (WHERE) from the collection of objects in FROM, but technically it is a Bag of structs

**NOTE 2:**
actual values returned from initial collection when applying GROUP BY expressions:
Struct(f₁:v₁, ..., partition:P).
First fields indicate the group, P is a bag of values belonging to this group

**NOTE 3:**
The SELECT clause may select from intermediate collection, i.e., f₁, f₂, ..., fₙ and partition – values may only be referred through aggregate functions on the members of bag P.
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
```
OQL BBS Example: Grouping – II

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

1. Initial collection: Sells

   ♦ But technically, it is a bag of structs of the form
     Struct(s: s1) where s1 is a Sell object.

   ♣ Note, the lone field is named s. In general, there are fields
     for all of the “typical objects” in the FROM clause.
OQL BBS Example: Grouping – III

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

2. Intermediate collection
   ♦ One function: s.bar.name maps Sell objects s to the value of the name of the bar referred to by s

   ♦ Collection is a set of structs of type:
     Struct{barName:string, partition:Set<Struct{s:Sell}>}

   ♦ For example:
     Struct{barName = "Joe's", partition = {s₁,...,sₙ}}
     where s₁,...,sₙ are all the structs with one field, named s, whose value is one of the Sell objects that represent Joe's Bar selling some beer.
OQL BBS Example: Grouping – IV

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

3. Output collection: consists of beer-average price pairs, one for each struct in the intermediate collection

♣ Type of structures in the output:
  Struct{barName: string, avgPrice: real}

♣ Note that the subquery in the SELECT clause – variables in the partition is referred through the AVG aggregate function

♣ We let p range over all structs in partition. Each of these structs contains a single field named s and has a Sell object as its value. Thus, p.s.price extracts the price from one of the Sell objects belonging to this particular bar.

♣ Typical output struct - example:
  Struct(barName = "Joe's", avgPrice = 2.83)
Another OQL BBS Example: Grouping – I

♣ Example:
Find, for each beer, the number of bars that charge a “low” price (\(\leq 2.00\)) and a “high” price (\(\geq 4.00\)) for that beer

♣ Strategy: group by three things:
The beer name,
a boolean function that is true if the price is low,
and a boolean function that is true if the price is high.

```
SELECT    beerName, low, high, count: COUNT(partition)
FROM       Beers b, b.soldBy s
GROUP BY   beerName: b.name,
            low: s.price <= 2.00,
            high: s.price >= 4.00
```
1. **Initial collection**: Pairs \((b, s)\), where \(b\) is a \texttt{Beer} object, and \(s\) is a \texttt{Sell} \(b.soldBy\) object representing the sale of that beer at some bar

   ♠ **Type of collection members:**
   \[
   \text{Struct}\{b: \texttt{Beer}, s: \texttt{Sell}\}
   \]
SELECT bName, low, high, count: COUNT(partition)
FROM Beers b, b.soldBy s
GROUP BY bName: b.name,
        low: s.price <= 2.00,
        high: s.price >= 4.00

2. Intermediate collection:
   Quadruples consisting of a beer name, booleans telling whether this
group is for high prices, low prices, and the partition for that group

- The partition is a set of structs of the type:
  Struct{b: Beer, s: Sell}

- A typical partition value:
  Struct (b:"Bud" object, s:a Sell object involving Bud)
Another OQL BBS Example: Grouping – IV

2. Intermediate collection (continued):

- Type of quadruples in the intermediate collection:

  ```
  Struct{   bName: string,  
            low: boolean,  
            high: boolean,  
            partition: Set<Struct{b: Beer, s:Sell}>}
  ```

- Typical structs in intermediate collection:

<table>
<thead>
<tr>
<th>bName</th>
<th>low</th>
<th>high</th>
<th>partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bud</td>
<td>TRUE</td>
<td>FALSE</td>
<td>$S_{low}$</td>
</tr>
<tr>
<td>Bud</td>
<td>FALSE</td>
<td>TRUE</td>
<td>$S_{high}$</td>
</tr>
<tr>
<td>Bud</td>
<td>FALSE</td>
<td>FALSE</td>
<td>$S_{mid}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

  **NOTE 1:**
  $S_X$ are the sets of beer-sells pairs $(b, s)$

  **NOTE 2:**
  $S_{low}$: price is low ($\leq 2$)

  **NOTE 3:**
  $S_{high}$: price is high ($\geq 4$)

  **NOTE 4:**
  $S_{mid}$: medium price (between 2 and 4)

  **NOTE 5:**
  the partition with $low = high = TRUE$ must be empty and will not appear
Another OQL BBS Example: Grouping – V

SELECT    bName, low, high, count: COUNT(partition)
FROM       Beers b, b.soldBy s
GROUP BY   bName: b.name,
           low: s.price <= 2.00,
           high: s.price >= 4.00

3. Output collection:

• The first three components of each group's struct are copied to the output

• The last (partition) is counted

• An example of the result:

<table>
<thead>
<tr>
<th>bName</th>
<th>low</th>
<th>high</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bud</td>
<td>TRUE</td>
<td>FALSE</td>
<td>27</td>
</tr>
<tr>
<td>Bud</td>
<td>FALSE</td>
<td>TRUE</td>
<td>14</td>
</tr>
<tr>
<td>Bud</td>
<td>FALSE</td>
<td>FALSE</td>
<td>36</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
OQL: Having

- GROUP BY may be followed by HAVING to eliminate some of the groups created by GROUP BY.

- The condition applies to the partition field in each structure in the intermediate collection.

- If condition in HAVING clause is FALSE, the group does not contribute to the output collection.
OQL BBS Example: Having

Example:
Find the average price of beers at each bar, but only in those bars where the most expensive beer cost more than 10$

```
SELECT barName, avgPrice: AVG(SELECT p.s.price
FROM partition p)
FROM Sells s
GROUP BY barName: s.bar.name
HAVING MAX(SELECT p.s.price
FROM partition p) > 10
```

**NOTE 1:**
Same as above, finds average price of beers in a bar

**NOTE 2:**
Select only those groups where the maximum price is larger than 10
Summary

・ OQL
  ・ Queries/subqueries – Select-From-Where
  ・ Return types – bags, sets, or lists
  ・ Quantifiers – for all, exists, etc.
  ・ Object creation –
    both new elements and returned form queries
  ・ Aggregation – count, max, min, avg, sum
  ・ Using host languages – OQL fits naturally
  ・ Operators on set or bag objects –
    union, intersect, except
  ・ Grouping with properties – group by with having