THE ODMG OBJECT QUERY LANGUAGE: OQL

I) COMPARING RELATIONAL AND OBJECT QUERY LANGUAGES
Objects have an Identity

- Objects have an Identifier + a Value
- **Pb**: Value based equality is not sufficient
- **Solution**: Extension of equality predicates
  - **Identity**: two objects are identical if they have the same oid
  - **Shallow equality**: two objects are equal if they have the same value
  - **Deep equality**: the object values may be different but if the oids’ are replaced by the corresponding values we can find two objects with the same values

Attributes can be multi-valued

- Attributes may have collection types (set, list, bag)
- **Pb**: The predicates for the comparison of atomic values are no longer sufficient
- **Solution**: Introduction of predicates for sets
  - **Membership predicates**: $y \in x.a$ is true if $y$ is a member of the value of the object $x$ attribute $a$
  - **Inclusion predicates**: $y.a \subseteq x.b$ is true if the value of the attribute $a$ of object $y$ is a subset of the value of attribute $b$ of object $x$
Objects/Values can be Complex (part_of)

- Attributes may also have complex types (including references)
- Pf: Navigating in the structure of complex objects/values without data reconstruction
- Solution: Introduction of path expressions to traverse the composition graph
- NOTE: As selectors in a path expression can be also use list indices and methods

Classes may have Subclasses (is_a)

- The objects of a subclass can be used whenever an expression is defined on its superclass (i.e., object references)
- Pf: The evaluation of predicates on the attributes that are defined or renamed at some level in the inheritance hierarchy
- Solutions:
  - Restriction of queries only on attributes defined in the most specific common superclass
  - Introduction of explicit selectors or usage of the casting mechanism for the objects of the queried classes
Object References may be Cyclic

- **Pb**: Cycle treatment in schema and instances in a declarative way (i.e., not using the methods)
- **Solution**: The cycles in the instances may be treated by joins based on object identity

Objects can also have a Behavior

- Object operations (methods) encapsulate data
- **Pb**:
  - a) The search of methods code at query compilation in the case of methods overloading
  - b) Treatment of side effects and method termination
- **Solution**:
  - a) Resolution at maximum of method names at query compile time and utilization of dynamic linking with low level optimization techniques (i.e C++)
  - b) Restriction of the used methods within a query only to the methods of the search (without side effects)
II) OQL SYNTAX AND SEMANTICS

Objectives of OQL

- Easy access in an OODB
  - Two needs for usage => two modes of usage (interactive or embedded in the PL)
- Support of the ODMG model
  - No impedance mismatch between the QL and the PL
- SQL-like declarative language
  - Optimization of queries
Characteristics of OQL

- **Functional Language** (closure)
  - Composition of queries

- **Orthogonal Language** (orthogonality)
  - Non privileged input data: the atoms, the n-arity relations and the set values are all treated uniformly
  - Non privileged output data: a query may return an object or a value of any type

- **Generic Language** (adequacy)
  - Usage of all the primitives of the ODMG model (encapsulation, object identity, inheritance etc.)

**An Example Schema**

```
UNIVERSITY
  name (string)
  address
  president

ADDRESS
  street (string)
  num (string)
  city (string)
  country (string)

PERSON
  name (string)
  birth_date (string)
  address

STUDENT
  enr_year (string)
  status (string)

EMPLOYEE
  salary (real)
  cars [ ]

VEHICLE
  model (string)
  color (string)
  manufacturer

COMPANY
  name (string)
  address
  owner
  employees { }
  president
```

---

Characteristics of OQL

- **Functional Language** (closure)
  - Composition of queries

- **Orthogonal Language** (orthogonality)
  - Non privileged input data: the atoms, the n-arity relations and the set values are all treated uniformly
  - Non privileged output data: a query may return an object or a value of any type

- **Generic Language** (adequacy)
  - Usage of all the primitives of the ODMG model (encapsulation, object identity, inheritance etc.)

**An Example Schema**

```
UNIVERSITY
  name (string)
  address
  president

ADDRESS
  street (string)
  num (string)
  city (string)
  country (string)

PERSON
  name (string)
  birth_date (string)
  address

STUDENT
  enr_year (string)
  status (string)

EMPLOYEE
  salary (real)
  cars [ ]

VEHICLE
  model (string)
  color (string)
  manufacturer

COMPANY
  name (string)
  address
  owner
  employees { }
  president
```
OQL in Interactive mode

- Atomic values
  - *How much is 2 + 2?*
    - \( 2 + 2 \)
  - *How much did the boss gained in February 1998?*
    - \(_\text{Boss} \rightarrow \text{salary}(5,2000)\)

- N-ary relations
  - *Give the boss’s address*
    - \(_\text{Boss} \rightarrow \text{address}\)
    - \(_\text{Boss}.\text{address}\)

- Sets
  - *Find the red vehicles manufactured by Renault*
    - `select v
      from VEHICLES v
      where v->color = “red” and
        v->manufacturer->name = “Renault”`
  - *Find the colors of the vehicles manufactured by Citroën*
    - `select v->color
      from VEHICLES v
      where v->manufacturer->name = “Citroën”`
  - *Find the address of the president of Peugeot*
    - `element(select c->president->address
      from COMPANIES c
      where c->name = “Peugeot”)`
Accessing Complex Structures

- **Path Expressions**
  - **Find the names of the employees of Renault that drive a car manufactured by Peugeot**
    
    ```
    select e->name
    from COMPANIES c, c->employees e, e->cars v
    where c->name = "Renault" and
      v->manufacturer->name = "Peugeot"
    ```
  
  - **Find the companies for which there exists (all) an employee having a car manufactured by Citroën**
    
    ```
    select c
    from COMPANIES c
    where exists e in c->employees:
      exists v in e->cars:
        v->manufacturer->name = "Citroën"
    ```
Constructing New Structures

- N-arity relations
  - Find the companies and the universities in Paris that are directed by the same president
    ```sql
    select struct(comp: c, univ: u)
    from COMPANIES c, UNIVERSITIES u
    where c->president = u->president and u->address->city = "Paris"
    ```
  - Find the names (in pairs) of companies having the same president, after elimination of duplicates
    ```sql
    select distinct struct(name1:c1->name, name2:c2->name)
    from COMPANIES c1, COMPANIES c2
    where c1->president = c2->president and c1 != c2
    ```

- Complex Values
  - Find the companies and the cars of their presidents if they have at least one car manufactured by their company
    ```sql
    select struct(comp: c, cars: v)
    from COMPANIES c, c->president->cars v
    where v->manufacturer = c
    ```
  - Find for each city with a department of the company Renault, the corresponding universities
    ```sql
    select struct(city: c->address->city, univs:(select u
    from UNIVERSITIES u
    where u->address->city = c->address->city))
    from COMPANIES c
    where c->name = "Renault"
    ```
Objects

- Create a new employee

  EMPLOYEE (name: "Vassilis", birth_date: "26/06/64", salary: 4300, cars:list())

- Create new objects of the class UNIVR for each company whose president lives in the city where a university is located

  select UNIVR(company:c, univs:
      select u from UNIVERSITIES u
      where u->president->address->city = c->address->city)
     from COMPANIES c

Manipulation of Collections

Set Operations

- Find the names of the presidents of Parisian universities and those of Parisian car companies

  define Paris_Univers_presidents as
      select u->president->name
      from UNIVERSITIES u
      where u->address->city = "Paris"

  define Paris_Company_president as
      select c->president->name
      from COMPANIES c
      where c->address->city = "Paris"

  Paris_Univers_presidents union Paris_Company_president
Querying a Hierarchy of Classes

- Find the names of the persons more than 20 years old
  
  ```sql
  select p->name
  from PERSONS p
  where p->age > 20
  ```

- Find the salary of the employees
  
  ```sql
  select ((EMPLOYEE) p)->salary
  from COMPANIES c, PERSONS p
  where p in c->employees
  ```

Cycle Treatment

- At instance level
  - Find the blue vehicles driven by the president of the manufacturing company
    
    ```sql
    select v
    from VEHICLES v
    where v->color = "blue" and
    v in v->manufacturer->president->cars
    ```

- At schema level
  - Find all companies that belong to the Renault Group
    
    ```sql
    select c->companies_of(c)
    from COMPANIES c
    where c->name = "Renault"
    ```
Ordering and Regrouping Of Objects

- Find and order the universities according to their name and the name of their city
  
  ```
  select u
  from UNIVERSITIES u
  order by u->address->city, u->name
  ```

- Group the universities by their city name
  
  ```
  select *
  from UNIVERSITIES u
  group by city: u->address->city
  ```

- Group the universities by their city name and keep only their names
  
  ```
  select city, univrs: select p.u->name from p in partition
  from UNIVERSITIES u
  group by city: u->address->city
  ```

OQL and SQL Incompatibilities: Input & Output

- **OQL**
  
  - Nested Queries: Select, From, Where
  - Type of Input: Any kind of Collection
  - Type of Output: Any Type of Data

- **SQL**
  
  - Where
  - Bags of Tuples
  - Bags of Tuples
OQL and SQL Incompatibilities: Ordered Tuples

- **Company** | **Date**
  - Renault  | 1899
  - Peugeot  | 1889

- **Firm** | **Creation**
  - Ford  | 1903
  - Mercedes  | 1900

- YES in the Relational World
- NO in the Object World

OQL and SQL Incompatibilities: Aggregates

**select** ...  
**from** UNIVERSITIES  
**group by** city

<table>
<thead>
<tr>
<th>City</th>
<th>Partition</th>
<th>OQL Interpretation</th>
<th>SQL Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Diego</td>
<td>UCSD</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>SDSU</td>
<td></td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>

F(name)  
F(partition)
OQL in Programming mode: An O2C Example

```java
run body{
    o2 set (VEHICLE) cars;
    o2 VEHICLE v;
    o2 set (EMPLOYEES) employees;
    oql(cars,
        "select v \
        from $1 e, e->cars v \
        where v->color = "red" and \
        v->manufacturer->name = "Renault", \
        employees);
    for (v in cars) v->display;
}
```

Defining Indices

- Definition of indices in the collections populated by n-arity relations
- The collections may be indexed on:
  - atomic attributes
  - attributes referencing other objects
  - attributes of type collection
  - a linear path expression for complex values
- The indices are used for the optimization of OQL queries
- Examples:
  - `create index COMPANIES on name;`
  - `create index COMPANIES on president;`
  - `create index COMPANIES on employees;`
Interpreting OQL queries

- Syntactical analysis of the query and rewriting in a conjunctive normal form:

  ```sql
  select v->color
  from VEHICLES v
  where v->manufacturer->name = "Renault"
  ```

- Variables range restriction and type inference

- The filters of the language can be treated in the following way:

  ```sql
  select q(x_1, x_2, ..., x_n)
  from f_1() x_1
  f_2(x_1) x_2
  ...
  f_n(x_1, x_2, ..., x_{n-1}) x_n
  where p(x_1, x_2, ..., x_n)
  ```

  for each x_1 in f_1()
  for each x_2 in f_2(x_1)
  ...
  for each x_n in f_n(x_1, x_2, ..., x_{n-1})
  if p(x_1, x_2, ..., x_n)
  Result <- q(x_1, x_2, ..., x_n)

- Optimization