Object Query Language

Syntax

Arūnas Janeliūnas

Object Databases

Example database

Classes (with attributes and methods):

Person: name, surname, birthDate, address, age()

Student: studentId, courses

Employee: department, salary()

Lecturer: title, courses

Course: title, id, lecturers, students

Address: city, street, house, flat

Storage roots:

People: Bag <Person>

Students: Set <Student>

Employees: Set <Employee>

Lecturers: Set <Lecturer>

Courses: List <Course>

Dean: Lecturer

Data access

Any name returns its value(-s):

```
Dean; // some object of the class Lecturer

Employees; // a set of Employee objects

1 + 1; // the result is … 2 :)
```

Unary path expressions

Like in any object oriented programming language:

```
Dean.name;

Dean.address.street;

Dean.salary();
```

Constructors

One can create objects/values *ad hock*:

```
struct ( street: “Lokio g–vė”,
         city: “Meškai” );

Address ( house: 5,
           flat: 14,
         street: “Partizanų”,
           city: “Joniškėlis” );

set (1, 3, 5, 7, 9);

array (1, 3, 5, 7, 9);

list (1, 2, 3, 4, 5, 6, 7, 8, 9);

list (1..9);
```

Iterators

SELECT is just a sort of the query, meaning iteration:

```
Select s
  from s in Students
 where s.name = “Sigitas”

Select struct ( name: s.name,
                age: age() )
  from s in Students

Select c.lecturer.address.city
  from c in Courses
 where c.title = “Object Databases”

Select s.name
  from s in Students,
       l in Lecturers
 where s.name = l.name
```

N-nary path expressions

```
select l.surname
  from l in Lecturers,
       c in l.courses,
       s in c.students
 where s.name = “Sigitas”
```

Matematikos
ir informatikos
fakultetas

Pointer join

```
select struct (student: s.surname,
               lecturer: l.surname)
  from l in Lecturers,
       c in l.courses,
       s in c.students
```

Matematikos
ir informatikos
fakultetas

Arūnas Janeliūnas

Methods

It can be used anywhere:

```
Dean.salary();

select p.name
  from p in Persons
 where p.age() > 21
```

Collections

It can be used anywhere as well:

```
select c.title
  from c in Dean.courses

select struct ( lecturer: l.surname,
                DB_courses: select c
                              from c in l.courses
                             where c.title like “*DB*”
              )
 from l in Lecturers
where count(select c
             from c in l.courses
            where c.title like “*DB*”) > 0
```

Sorting

```
select s
  from s in Students
order by s.age() asc,
         s.name
```

Alternative syntax, seen in some DBMSes:

```
sort s in Students
  by s.age() asc, s.name
```

Grouping

```
select e
  from e in Employees
group by       rich: e.salary() > 2000,
          moderate: e.salary() > 500 and e.salary() <= 2000,
              poor: e.salary() <= 500
having avg( select p.salary() from p in partition ) > 1111
```

The result of this query is like this:

```
{
  [ rich: false, moderate: false, poor: true,  partition: { e11,e12,...} ],
  [ rich: false, moderate: true,  poor: false, partition: { e21,e22,...} ],
  [ rich: true,  moderate: false, poor: false, partition: { e31,e32,...} ],
}
```

Agreagation etc.

```
max ( select e.salary()
        from e in Employees )


element ( select c
            from c in Courses
           where c.title like “*DB*” and
                 c.id = 101 );
```

```
first ( element( select c
                   from c in Courses
                  where c.title like “*DB*” and
                        c.id = 101
                ).lecturers
      );

listtoset (Dean.courses);

flatten ( select l.courses
            from l in Lecturers )
```

Set operations

```
Persons except Students;

Students union set(Dean);

Courses || list ( Course( title: “ODB”,
                          id: 13,
                          lecturers: set(Dean),
                          students: Students )
)
```

Quantums

```
select s.name
  from s in Students
 where for all c in s.courses : c.title like “*DB*”;

select s.name
  from s in Students
 where exists c in s.courses :
            (exists l in c.lecturers :
                l.name = “Janeliūnas”)
```

Named queries

```
Define Sigitai as
  select distinct s
    from s in Students
   where s.name = “Sigitas”
```

```
Select ss.address.city
  from ss in Sigitai

Undefine Sigitai
```

Summary

1. **c** - any constant;
2. **n** - names (data storage roots);
3. **x** - iteration variables;
4. constructors **struct**, **set**, **list**, **bag**, **array** :

struct (name$_1$: expr$_1$, ... , name$_n$: expr$_n$)
set (expr$_1$, ... , expr$_n$)
list (expr$_1$, ... , expr$_n$)
list (expr$_1$ .. expr$_2$)
bag (expr$_1$, ... , expr$_n$)
array (expr$_1$, ... , expr$_n$)

Summary

5. Operations:

numerical: `+ , – , * , /, mod, abs(expr)`
logical: **`not, and, or`**
structural: `.` (attribute extractor)
set: **`except, union, intersect, flatten, element, distinct, count,`**
`sum, avg, min, max, count`
list: `||`, **`first, last, listtoset`** and set operations
bag: **`distinct`** and set operations
object: `.` (message sending)

Summary

6. predicates:

$expr_1$ Θ $expr_2$, $\Theta \in$ { **=** , **!=**, **<**, **>**, **<=**, **>=** }

```
for all x in col: boolexpr(x)
exists x in col: boolexpr(x)
```

Summary

7. iterations:

```
select [distinct] expr1(x1 , ... , xn),
                        . . .
                  exprm(x1 , ... , xn)
  from x1 in col1 ,
            . . .
       xn in coln
[where boolexpr (x1 , ... , xn)]
[group by name1 : expr1 , ... , namen : exprm ]
[having boolexpr(name1 , ... , namen) ]
[order by expr1 [asc, desc], ... , exprq  [asc, desc]]
```

Summary

8. naming:

```
define name as expr

undefine name
```

9. comments:

```
// single line comments
/* block comments,
   having as many lines
   as you decide is necessary */
```