ODMG OQL
User Manual

Release 5.0 - April 1998

Who should read this manual

OQL is an object-oriented SQL-like query language, the ODMG standard. This manual describes how to use OQL as an embedded function in a programming language (e.g. O_2C, C, C++, or Java) or interactively as a query language. It assumes previous knowledge of the O_2 system.

Other documents available are outlined, click below.

See O2 Documentation set.

TABLE OF CONTENTS

This manual is divided into the following chapters:

TABLE OF CONTENTS

TABLE OF CONTENTS

1 Introduction

Congratulations! You are now a user of the object-oriented query language OQL.

O_2 is a revolutionary system that is particularly well adapted for developing large-scale client/server applications in both fields of business and technical software development.

This chapter introduces the O_2 system and the OQL query language.

The chapter is divided into the following sections :

1 Introduction

1.1 System Overview

The system architecture of O_2 is illustrated in Figure 1.1.

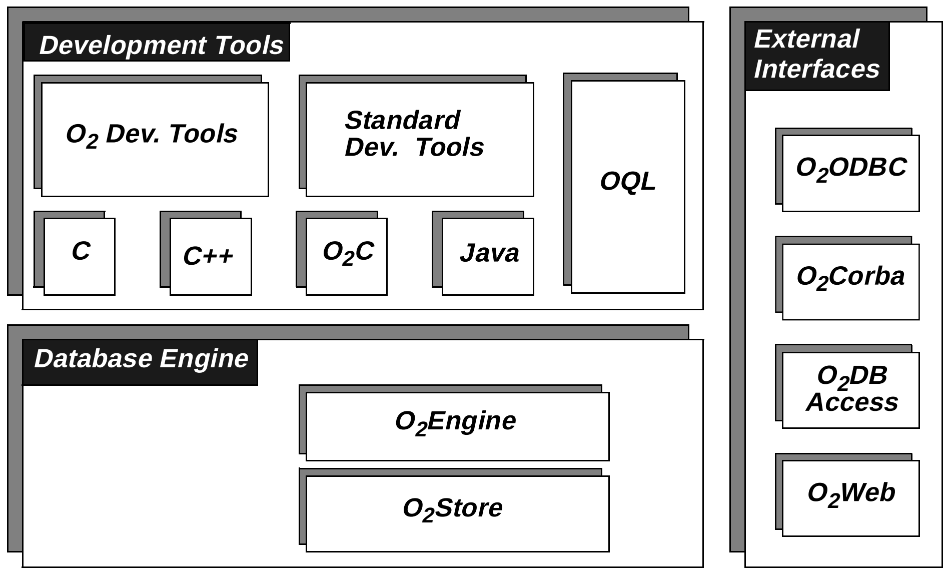


Figure 1.1: O_2 System Architecture

The O_2 system can be viewed as consisting of three components. The *Database Engine* provides all the features of a Database system and an object-oriented system. This engine is accessed with *Development Tools*, such as various programming languages, O_2 development tools and any standard development tool. Numerous *External Interfaces* are provided. All encompassing, O_2 is a versatile, portable, distributed, high-performance dynamic object-oriented database system.

Database Engine:

- O_2Store — The database management system provides low level facilities, through O_2Store API, to access and manage a database: disk volumes, files, records, indices and transactions.
- O_2Engine — The object database engine provides direct control of schemas, classes, objects and transactions, through O_2Engine API. It provides full text indexing and search capabilities with O_2Search and spatial indexing and retrieval capabilities with O_2Spatial. It includes a Notification manager for informing other clients connected to the same O_2 server that an event has occurred, a Version manager for handling multiple object versions and a Replication API for synchronizing multiple copies of an O2 system.

Programming Languages:

O_2 objects may be created and managed using the following programming languages, utilizing all the features available with O_2 (persistence, collection management, transaction management, OQL queries, etc.)

- C — O_2 functions can be invoked by C programs.
- C++ — ODMG compliant C++ binding.
- Java — ODMG compliant Java binding.
- O_2C — A powerful and elegant object-oriented fourth generation language specialized for easy development of object database applications.
- OQL — ODMG standard, easy-to-use SQL-like object query language with special features for dealing with complex O_2 objects and methods.

O_2 Development Tools:

- O_2Graph — Create, modify and edit any type of object graph.
- O_2Look — Design and develop graphical user interfaces, provides interactive manipulation of complex and multimedia objects.
- O_2Kit — Library of predefined classes and methods for faster development of user applications.
- O_2Tools — Complete graphical programming environment to design and develop O_2 database applications.

Standard Development Tools:

All standard programming languages can be used with standard environments (e.g. Visual C++, Sun Sparcworks).

External Interfaces:

- O_2Corba — Create an O_2/ Orbix server to access an O_2 database with CORBA.
- O_2DBAccess — Connect O_2 applications to relational databases on remote hosts and invoke SQL statements.
- O_2ODBC — Connect remote ODBC client applications to O_2 databases.
- O_2Web — Create an O_2 World Wide Web server to access an O_2 database through the internet network.

1 Introduction

OQL

OQL is an object-oriented SQL-like query language. OQL is the query language of the ODMG-93 standard[1]. It can be used in two different ways either as an embedded function in a programming language or as an ad hoc query language.

You can use OQL as a function called from O_2C, C, C++, Smalltalk or Java, in order to manipulate complex values and methods. Each construct produces a result which can then be used directly in the programming language. Methods can be triggered to modify the database. You will find that programming is easier because OQL can filter values using complex predicates whose evaluations are optimized by the OQL optimizer in O_2.

OQL can also be used interactively as an ad hoc query language allowing database queries from both technical and non-technical users. Interactive features include fast and simple browsing of the database.

Browser Interface

The browser interface you see depends on the operating system you are using.

- ***Unix***

In Unix, the O_2Look graphical user interface generator is used to generate the graphical form of OQL query results.

Figure 1.2 shows a typical query result in graphical form, as generated by O_2Look.

1. The Object Database Standard: ODMG - 93. Atwood, Barry, Duhl, Eastman, Ferran, Jordan, Loomis and Wade. Edited by R.G.G. Cattell. © 1996 Morgan Kaufman Publishers.

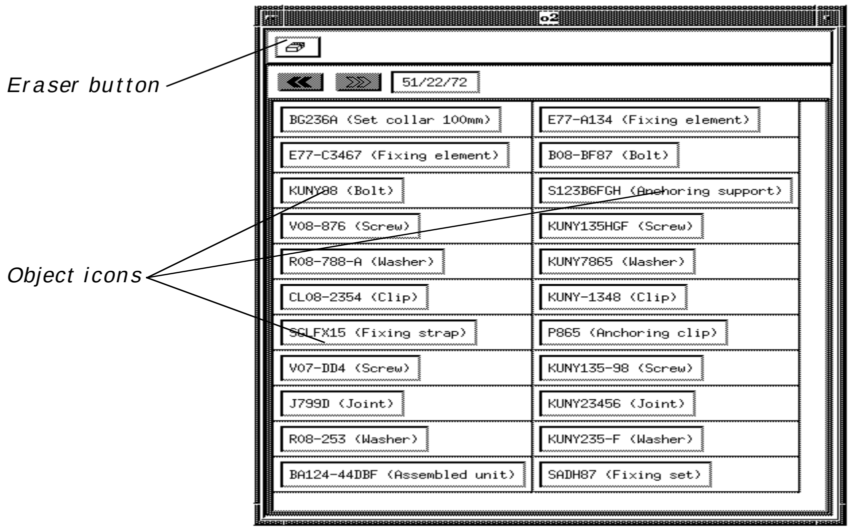


Figure 1.2: Typical OQL query result in graphical form, as generated in Unix

In addition to the usual Motif buttons a graphical query result has an Eraser button. Clicking on the Eraser button removes the graphical result. This query result consists of a number of objects. Each object has its own pop-up menu which is displayed by clicking the Object icon using the right mouse button. This pop-up menu can be used to access the public methods of each object.

- ***Windows NT***

In Windows NT, the query result is displayed in a window in textual form containing hypertext links. Each link represents a sub-object.

The label for a specific link may be obtained by applying the **`title`** method to the sub-object represented by the link.

Clicking on a hypertext link, with the right mouse button, replaces the contents of the window with a representation of the sub-object associated with the link.

Figure 1.3 shows a typical query result in graphical form, as generated in Windows NT.

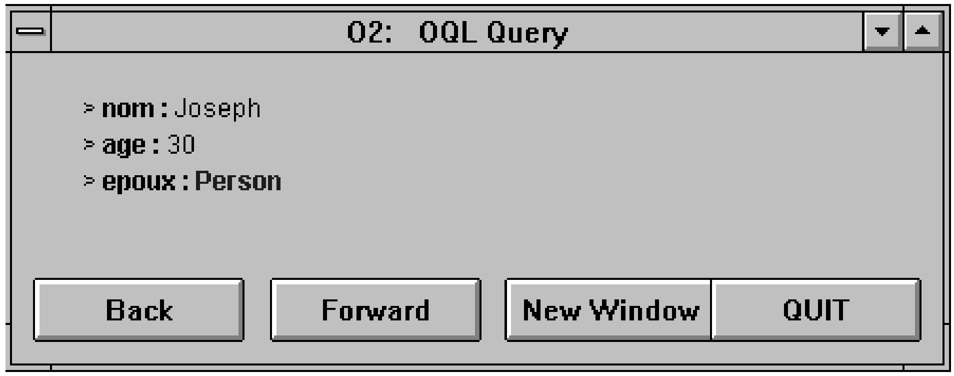


Figure 1.3: Typical OQL query result in graphical form, as generated in Windows NT

The browser shown in Figure 1.3 has the following buttons:

Back this button displays the previous object.

Forward this button displays the next sub-object. It is only valid if the Back button has been activated at least once.

New Window This button displays the current object in a new window. Each window is an independent browser.

Quit This button closes the active window.

The query result is an object of the Person class, which has a name, an age and a spouse. A spouse is also an object of the Person class, and thus appears in as a hypertext link. Left clicking displays the spouse object.

Note

The rest of this manual will only show graphical displays from the Unix platform.

1.2 Interactive and embedded query language

It is because OQL is so easy to use interactively that all kinds of users including non-technical users can browse the database quickly and efficiently to get the information they want. OQL can also be used as a function called from C, C++, Java, O_2C and O_2 Engine API.

Interactive OQL

The OQL interpreter can be triggered by the **query** command of **O2dba**, **O2dsa** or O_2 shells. The command interpreter prompts you with the following message:

type your command and end with ^D

To run OQL, type:

query

^D

You must type **^D** (Control - D) on a separate line. You now see:

Query Interpreter

type your query and end with ^D

Type your query, ending it with **^D**.

"this is a query"

^D

The answer is automatically displayed and the system returns to the OQL prompt:

type your query and end with ^D

To leave the query session type:

^D (or **quit)**

You are now back in the command interpreter and you see the message:

type your command and end with ^D

You can also use OQL in the O_2Tools programming environment (Refer to the O_2Tools User Manual).

Note

In a Windows environment **^Z** (Control - Z) is used instead of **^D** (Control - D).

Embedded OQL

Any valid query can be passed from O_2C code to OQL using the system supplied function **o2query**. This is detailed in the O_2C Reference manual.

Similarly, you can pass a query to a C++, C, Smalltalk or Java program. Refer to the respective manuals for details.

Finally an OQL function exists in O_2Engine and is described in the O_2Engine API Reference Manual.

1.3 Manual overview

This manual is divided up into the following chapters:

- Chapter 1 - Introduction

  This chapter introduces the O_2 system and the OQL query language.

  It outlines the concepts of the ad hoc query language that allows you to browse the database quickly and efficiently to get the information you want, and the embedded query language that you can call from inside your programs.

- Chapter 2 - OQL - Getting started

  This chapter introduces the OQL language so you can start to use OQL in order to obtain the exact information you want from your database.

  It describes and illustrates basic and “select..from..where” queries, details how to construct results and describes the use of operators and indexes. To fully understand this chapter, you must know the ODMG data model.

- Chapter 3 - OQL Rationale

  This chapter introduces the ODMG standard and describes the ODMG object model. It also gives an example based presentation of OQL.

- Chapter 4 - OQL Reference

  This chapter contains the ODMG reference manual for OQL 1.2. It is the same as the ODMG standard with added notes and explanations on how to use OQL with O_2.

  For each feature of the language, you get the syntax, in informal semantics, and an example. Finally, the formal syntax is given.

2 Getting Started

AN OBJECT-ORIENTED DATABASE QUERY LANGUAGE

So that you can obtain the exact information you want from your database, O_2 has an object oriented database query language OQL.

OQL is a powerful and easy-to-use SQL-like query language with special features for dealing with complex objects, values and methods.

This chapter introduces the OQL language and is divided up into the following sections:

- Basic queries
- Select ... from ... where
- Constructing results
- Operators
- Set operators
- Conversions
- Combining operators
- Indexes
- Chapter Summary

To understand this chapter you need to know the ODMG data model[1]. As an introduction to the data model you can refer to chapter 3 of this manual or the O_2C Beginner's Guide.

Experience of SQL, though not a prerequisite, will facilitate the OQL learning process.

1. The Object Database Standard: ODMG - 93, release 1.2. Edited by R.G.G. Cattell. © 1996 Morgan Kaufman Publishers.

2 Getting Started

2.1 Basic queries

All the examples shown below are based on the following O_2 schema:

- In O_2C

```
class o2_set_Employee public type
      unique set (Employee)
end;

class o2_list_Client public type
      list (Client)
end;

class Company public type
      tuple (   name: string,
                employees: o2_set_Employee,
                clients: o2_list_Client
            )
      method public title: string
end;

class Client public type
      tuple (   name: string,
                order: list (tuple (  what: string,
                                      price: real))
       )
end;

class Employee public type
      tuple (   name: string,
                birthday: Date,
                position: string,
                salary: real)
      method age: integer
end;
```

- In C++

```
class Company {
public:
          d_String name;
          d_Set<d_Ref<Employee> > employees;
          d_List<d_Ref<Client> > clients;
          char* title() {return name;}
};

class item { d_String what; double price;};

class Client {
public:
          d_String name;
          d_Array<item> order;
};

class Employee {
public:
          d_String name;
          d_Date birthday;
          d_String position;
          float salary;
          int age();
};
```

Two persistent roots are also defined: An object, **Globe** and a collection **the_employees**.

```
name Globe: Company;
constant name the_employees: o2_set_Employee;
```

Database entry points

To query any database you need various entry points.

In O_2 these are the named objects and named values.

For example, **`Globe`** is an entry point.

The simplest OQL query calls an entry point:

```
Globe
```

This returns:

In an O_2 database, named objects and values can either be values of any type, or objects of any class. Consequently, OQL allows you to query values or objects of any type or class.

Note

The query results shown below are all given in the Unix graphic form.

Simple queries

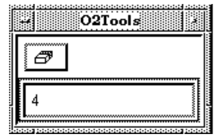
Simple queries can involve different types of values:

- ***Atomic values***

With atomic values you can carry out arithmetic calculations, e.g.,

```
2 * 2
```

This is a query which returns the integer 4.



- ***Struct values***

You can also consider the value of the object **Globe** of class **Company** as a struct (or tuple) value with three attributes.

The only operation you can carry out on a struct is extracting a field, e.g.,

```
Globe.name
```

This returns the name of the Globe Company.

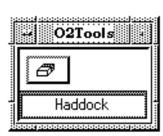
- ***List or array values***

A list is an ordered collection that allows duplicates and you can therefore extract any of its elements if you know their position.

For example, you can extract the first element of the list in **clients** as follows.

```
Globe.clients[0]
```

In OQL, you count list elements from 0.



For OQL, an array behaves the same way as a list.

- ***Call of a method***

To apply a method to an object is a base query, e.g.

```
Globe.title
```

This applies the method **`title`** to the object **`Globe`** and returns the result of the method **`title`**:

2.2 Select ... from ... where

The **`select from where`** clause enables you to extract those elements meeting a specific condition from a collection. O_2 collections include set, bag (a multi-set or set with duplicates), list (an insertable and dynamic array) or array.

The OQL query has the following structure:

`select`: defines the structure of the query result

`from`: introduces the collections against which the query runs.

`where`: introduces a predicate that filters the collection.

This section now describes how to use this clause.

Set

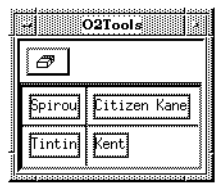
A set is a non-ordered collection.

The most frequent query on a set is a filter. This consists of extracting the elements of a set which have certain characteristics.

For example:

```
select e
from e in Globe.employees
where e.salary > 200.00
```

This query returns those employees working at the International Globe with a salary over 200:



The **select** clause defines the query result as the employees and the **from** clause gives the set on which to run the query. The variable **e** represents each of its elements in turn. The **where** clause filters the employees so that those earning more than 200 are extracted.

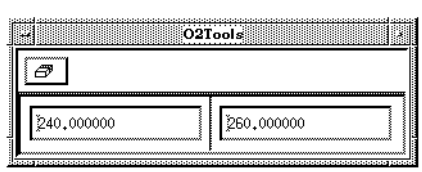
This query therefore builds a collection of employees.

This collection is in fact a bag as duplicates are accepted. You can also add the keyword **distinct** to eliminate any duplicates from the resulting bag and then produce a true set.

Moreover, you can access from **e** any attributes, e.g. **salary** and get a set of real numbers. For example:

```
select distinct e.salary
from e in Globe.employees
where e.position = "Reporter"
```

This gives a set of the salaries of the Reporters:

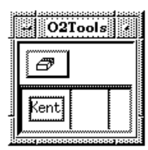


Join

You can also use a query to select from more than one collection:

```
select e
from e in Globe.employees,
      c in Globe.clients
where e.name = c.name
```

This query returns the set of employees who have the same name as a client. If there is a client called Kent and an employee called Kent, you see the following window:



Path expressions

Objects are related to other objects, and in order to get to the data it needs, a query can follow various paths that start from any O_2 object or collection. For example,

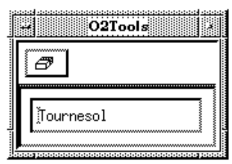
```
select distinct ord.what
from cl in Globe.clients,
     ord in cl.order
where cl.name = "Haddock"
```

You obtain the set of what the client(s) called Haddock bought:

Testing on nil

After your application has updated the database, you may find that some objects are now equal to nil. You can test for this using OQL. For example, you can test that a client exists and if so, which client has three orders:

```
select c.name
from c in Globe.clients
where c!=nil and count (c.order) = 3
```



To simplify programming, OQL skips nil objects when they are encountered. If a path expression contains a **nil** object, a predicate is always considered **false**. This means that the previous expression can be rewritten as follows:

```
select c.name
from c in Globe.clients
where count (c.order) = 3
```

List or array

A list or an array is an ordered collection that can contain duplicate elements.

Since it is ordered, you may extract any of its elements if you know their position. For example:

```
Globe.clients[2]
```

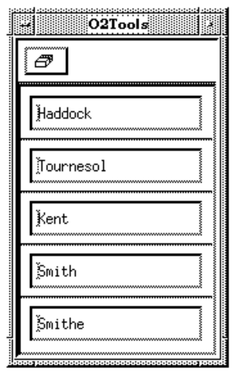
This extracts the third element of the list (the first element is at position 0).

As with sets you can filter a list.

For example: what are the names of the clients who buy the International Globe newspaper?

```
select e.name
from e in Globe.clients
```

The result of this query is a bag of the **name** of **Globe** clients:



Note

The query returns a *bag* and not a list. To return a list, you must define an order. See "Order by" on page 35.

You can also add the keyword **distinct** to a selection to eliminate any duplicates from the resulting set.

Note

You can manipulate very complex structures. A list can be made up of tuples which in turn can have a set attribute, etc. Consequently, you have access to all the embedded components of an object.

For more details, refer to Section 2.3 for constructing query results and Section 2.7 for combining operators.

2.3 Constructing results

The structure of a query result is very often implicit. For example, when you extract the **age** field of an employee, which is of type integer, you obtain an integer. When you filter a set, bag or list, you obtain a set, bag or list depending on what you select.

However, you can also construct a query result with an explicit structure using the **struct**, **set**, **bag**, **list** and **array** constructors.

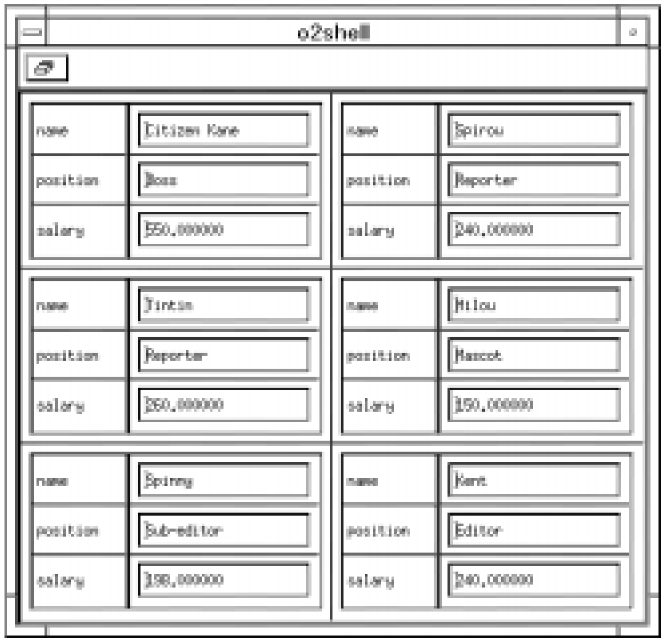
For example, using the struct constructor:

```
select struct (employee: e.name,
               position: e.position,
               salary: e.salary)
from e in Globe.employees
```

or simply:

```
select e.name, e.position, e.salary
from e in Globe.employees
```

This query gives the name, position and salary of the employees at the International Globe newspaper:



You can use the special "*" operator to select all attributes of the elements of a collection.

For example:

```
select * from Globe.employees
```

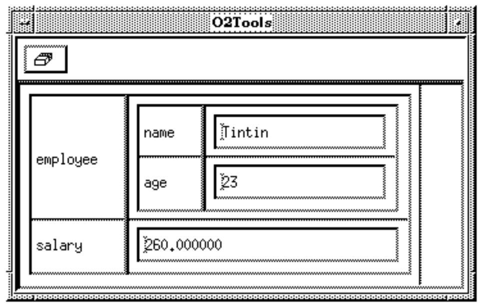
Note that in this example you do not need to define a variable with **from**.

You can also build up embedded structures simply by combining **struct** operators.

For example, to get the identities and salaries of all those employees working as reporters and older than 22.

```
select struct (employee: struct (name: e.name,
                                 age: e.age),
               salary: e.salary)
from   e in Globe.employees
where  e.position = "Reporter" and
       e.age > 22
```

This query gives a bag with one element:



Creating an object

You create values using **`struct`**, **`list`**, **`array`**, **`bag`** and **`set`**. In OQL, you can also create objects using the class name and by initializing the attributes of your choice. Any un-initialized attributes are set to the default value. For example, to create an object of the class **`Client`**:

```
Client (name: "Trent")
```

This creates a temporary object with the name attribute initialized to **`Trent`**.

You can then make the object persistent in the usual way (refer to the O_2C, C++ and Java manuals). The result of this query is the new object.

An object collection can be created in the same way. For example, use the following query to create an o2_list_Client collection.

```
o2_list_Client (list(Client(name:"John"),
                    (Client(name:"Jack")))
```

2.4 Operators

This section outlines the basic OQL operators you can use to query the database.

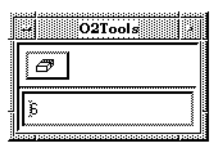
Count

You can query the database using the **count** clause.

For example, to find out how many employees there are at the International Globe newspaper:

```
count (Globe.employees)
```

This query returns an integer.



Other aggregate operators are **min**, **max**, **sum** and **avg**.

Define

You can name the result of a query using the **define** clause. For example,

```
define MyEmployees as
        select e
        from e in Globe.employees
        where e.name like "Sp*"
```

This names the result of the query and not the query itself.

The name **MyEmployees** can then be used in other queries. Named queries greatly improve the legibility of complex queries.

Note

You can only reuse these named queries in the same query session, i.e., up to a commit or abort point.

Element

When you have a set or a bag that contains a single element, you extract the element directly using the **element** operator. For example,

```
element ( select e
          from e in Globe.employees
          where e.name = "Tintin")
```

This query gives the result:

Exists

You can add a new persistent name to cover all the different companies that exist:

```
name TheCompanies: list (Company);
```

You can now carry out more complex queries, such as selecting which company has at least one employee under the age of 23:

```
select c.name
from c in TheCompanies
where exists e in c.employees: e.age < 23
```

The answer is a bag of names:

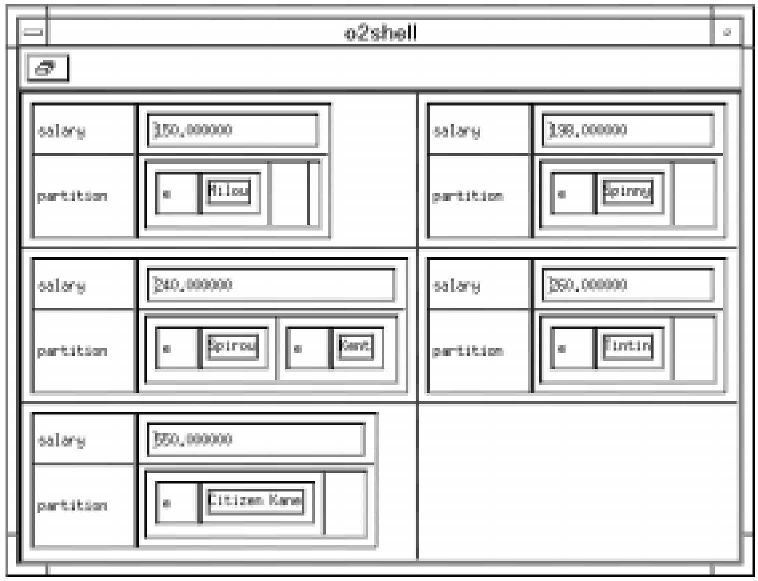
Group by

This operator groups together objects of a collection with the same value for particular attributes.

For example,

```
select *
from e in Globe.employees
group by e.salary
```

This groups the **employees** by salary giving a bag of two-attribute tuples:



The first attribute is the salary and is called **salary** as specified. The second is the set of objects (employees) with the same salary and is called **partition**.

Thus, the type of the result of this query is:

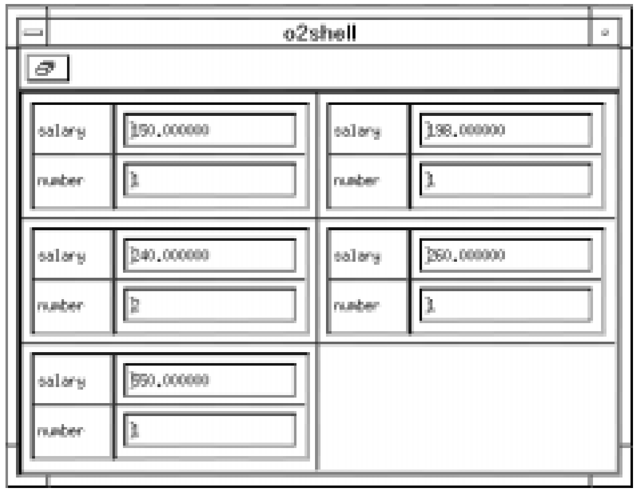
```
bag (struct (salary: real,
               partition: bag (struct (e:Employee))))
```

You can work on a partition value by computing statistics on each partition.

The following query returns a bag of two-attribute tuples with the salary and the number of employees earning each of these salaries:

```
select salary, number: count (partition)
from e in Globe.employees
group by e.salary
```

You get the following type of window:

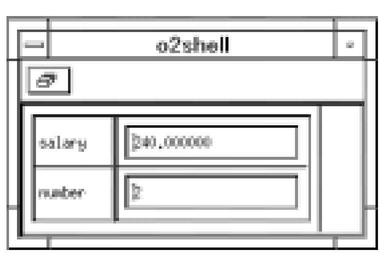


Finally you can filter the result of grouping by applying predicates on aggregative operations. You can select groups with conditions on average, count, sum, maximum and minimum values of partitions. You do this using the **having** clause.

For example, if you wish to select only groups with more than one salary:

```
select salary, number: count (partition)
from e in Globe.employees
group by e.salary
having count (partition) > 1
```

The following screen is displayed.



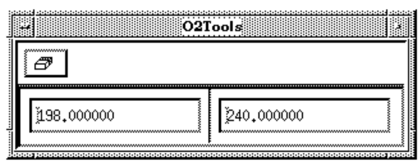
Like

The **like** operator allows you to test part of a character string. The "*" character stands for any string including the empty string.

The query:

```
select distinct e.salary
from e in Globe.employees
where e.name like "Sp*"
```

returns the salaries of all employees whose names begin with **Sp**:



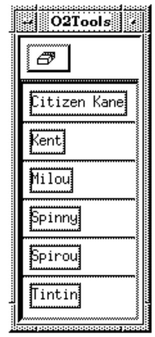
Order by

You can obtain a sorted list using the **order by** clause. For example, to sort the employees by name and by age:

```
select e from e in Globe.employees order by e.name, e.age
```

The result of an **order by** operation is always a list, even though the source of the objects to sort (the set **employees**, in this case) may be a set.

This query returns a list of employees; their order is alphabetical by name, and then by age:



2.5 Set operators

The standard set operations are defined on set and bag: **union**, **intersect** (intersection) and **except** (difference).

You can also write these operators as **+** (union), ***** (intersection) and **-** (difference).

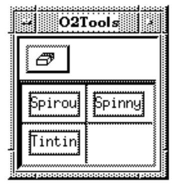
You can define another query **YourEmployees**:

```
define YourEmployees as
        select e
        from e in Globe.employees
        where e.name = "Tintin"
```

Now you can combine the queries by adding together two sets:

```
MyEmployees + YourEmployees
```

The simple addition (union) of the two sets of employees gives you a set containing the answer:



The **pick** operator is defined on a set or a bag. It returns an element of the collection, chosen arbitrarily.

For example:

```
pick (MyEmployees)
```

2.6 Conversions

List to set

To convert a list or array to a set you use the **listtoset** operator.

Example:

```
listtoset (Globe.clients) intersect
listtoset (TheCompanies[2].clients)
```

Set to list

To convert a set or bag to a list you must order it.

For example:

```
select e from e in the_employees order by e.salary
```

returns a list sorted by salary.

You can also use "*" to build a list. This avoids a real sort algorithm and should be used when the final order of the list is unimportant.

```
select e from e in the_employees order by *
```

returns a list of all employees in random order.

Flatten

To convert a collection of collections into a flattened collection you use the **flatten** operator.

For example:

```
flatten (select distinct c.clients
         from c in TheCompanies)
```

returns a set of clients.

2.7 Combining operators

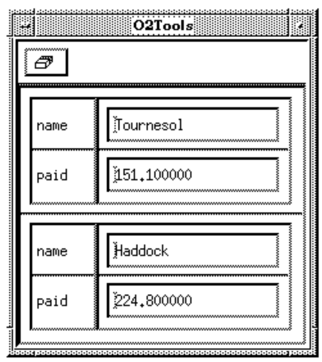
OQL is a complete functional language in that every operator can be combined with any other operator.

You can use combine and build up operators, universal and existential quantifiers, wild-card operators, standard set operators as well as list concatenation, ordering and grouping operators on sets, bags and lists.

For example:

```
select cl.name, paid: sum (select p.price from p in
                              cl.order)
from cl in Globe.clients
where count (cl.order) >2
order by sum (select p.price from p in cl.order)
```

This sorts all the clients, with more than two orders, by how much they have paid to the company:



2.8 Indexes

When OQL extracts one or more elements from a collection using a specified predicate or order operation, it must scan the whole collection to find the required elements.

You can improve performance if the system is able to directly access the matching elements. This is done by establishing an index on a collection.

An index maps a key to one or more elements of a named collection.

Whenever a program searches for elements of the collection using the key, the system uses the index to quicken the search.

This entire process is totally transparent to you as the programmer. The absence or presence of an index has no effect on program code, only on system performance.

The benefits of indexes include the following:

- Complete logical and physical independence

You do not have to change your query to use indexing. Indexes are created by administration commands.

- High performance during use and maintenance

Access from an index means constant time access irregardless of the size of the collection.

Example:

- Defining an index for all employees:

```
create index the_employees on salary;
```

- The following query will then be optimized:

```
select e
from e in the_employees
where e.salary ≥ 1000 and e.salary ≤ 5000
```

Display index

The "display index" query allows you to see how OQL will use existing indexes in queries you will make. To stop this feature, execute "display index" again.

Note

Please refer to the System Administration Guide for details on how to create and manage indexes.

2.9 Chapter Summary

This chapter has covered the following points:

- ***Basic queries***

  To query any database you need various entry points. In O_2 these are the named instances — i.e. named objects and named values.

  Simple queries include: calling an entry point, applying a method to a named object, extracting a field, etc.

- ***Select..from..where***

  The **`select ... from ... where`** clause enables you to extract those elements meeting a specific condition from a list or set.

- ***Constructing results***

  The structure of a query result is very often implicit. However, you can also construct a query result with an explicit structure using the **`struct`**, **`set`** and **`list`** constructors.

- ***Operators***

  OQL operators include **`define`**, **`element`**, **`order by`**, **`count`**, **`exists`**, **`group by`** and **`like`**. They can be combined for complex queries.

- ***Indexes***

  When OQL extracts one or more elements from a set or list it scans the whole collection to find the desired elements. You can improve performance if you tell the system exactly where to look.This is done by establishing an index on a collection. An index maps a key to one or more elements of a named collection.

2

Getting Started

3 OQL Rationale

Most commercial object database systems now have a common data model based on the OMG object model. This data model is defined in the ODMG 93 report. Based on this ODMG model, the query language OQL was defined and adopted by the ODMG group.

This chapter is divided as follows:

3 OQL Rationale

3.1 The ODMG standard

The ODMG standard covers the following points:

1. an object model
2. an object definition language for this model, with its own syntax, ODL or its expression through C++ and Smalltalk syntax
3. an object query language for this model, OQL
4. a C++ binding allowing C++ programs to operate on a database compliant to the object model
5. a Java binding allowing Java programs to operate on a database compliant to the object model

3.2 The ODMG model

The ODMG object model supports the notion of classes, of objects with attributes and methods, of inheritance and specialization. It offers the classical types to deal with string, date, time, time interval and timestamp. And finally, it supports the notions of relationships and collections.

ODMG-93 introduces a set of predefined generic collection classes: **`Set<T>`**, **`Bag<T>`** (a multi-set, i.e., a set with repeated elements), **`Varray<T>`** (a variable size array), **`List<T>`** (a variable size and insertable array).

An object refers to another object through a Ref. A Ref behaves as a C++ pointer, but with more semantics: it is a persistent pointer but referential integrity can be expressed in the schema and maintained by the system. This is done by declaring the relationship as symmetric.

Combining relationships and collections, an object can relate to more than one object through a relationship. Therefore, 1-1 relationships, 1-n relationships and n-m relationships can be supported with the same guarantee of referential integrity.

ODMG-93 enables explicit names to be given to any object or collection. From a name, an application can directly retrieve the named object and then operate on it or navigate to other objects following the relationship links.

Let us now present the model through a complete example. We use here C++ syntax for our object definition language, following the ODMG C++ ODL binding (i.e., the way of defining an ODMG schema using the standard C++ language).

```
class Person{
   d_String name;
   d_Date   birthdate;
   d_Set  < d_Ref<Person> > parents
         inverse children;
   d_List < d_Ref<Person> > children
         inverse parents;

   d_Ref<Apartment> lives_in
       inverse is_used_by;

              Methods

  Person();      Constructor: a new Person is born
  int age();     Returns an atomic type

  void marriage( d_Ref<Person> spouse);
                            This person gets a spouse
  void birth( d_Ref<Person> child);
                            This person gets a child
  d_Set< d_Ref<Person> > ancestors;;
                        Set of ancestors of this Person
  virtual d_Set<d_String> activities();
                            A redefinable method
};

class Employee: Person{
                             A subclass of Person

  float salary;
                               Method
  virtual d_Set<d_String> activities();
                            This method is redefined
};
```

```
class Student: Person{

  d_String grade;

  virtual d_Set<d_String> activities();

};
```

A subclass of Person

Method

The method is redefined

```
class Address{
   int number;
   d_String street;
};

class Building{
  Address address;
            A complex value Address embedded in this object
  d_List< <d_Ref<Apartment> > apartments
             inverse building;
                                                Method
  d_Ref<Apartment> less_expensive();
};

class Apartment{
  int number;
  d_Ref<Building> building;
  d_Ref<Person> is_used_by
             inverse lives_in;
};

d_Set<  d_Ref<Person> >    Persons;
                              All persons and employees
d_Set<  d_Ref<Apartment> > Apartments;
                              The Apartement class extent
d_Set<  d_Ref<Apartment> > Vacancy;
                              The set of vacant appartements
d_List< d_Ref<Apartment> > Directory;
         The list of appartements ordered by their number of rooms
};
```

3.3 OQL by example

Let us now turn to an example based presentation of OQL. We use the database described in the previous section, and instead of trying to be exhaustive, we give an overview of the most relevant features.

Path expressions

As explained above, one can enter a database through a named object, but more generally as soon as one gets an object (which comes, for instance, from a C++ expression), one needs a way to “navigate” from it and reach the right data one needs. To do this in OQL, we use the “**.**” (or indifferently “**->**”) notation which enables us to go inside complex objects, as well as to follow simple relationships. For instance, given a Person **p** to know the name of the street where this person lives, we use the following OQL query:

```
p.lives_in.building.adddress.street
```

This query starts from a **Person**, traverses an **Apartment**, arrives in a **Building** and goes inside the complex attribute of type **Address** to get the street name.

This example treated 1-1 relationship, let us now look at n-p relationships. Assume we want the names of the children of the person **p**. We cannot write: **p.children.name** because **children** is a List of references, so the interpretation of the result of this query would be undefined. Intuitively, the result should be a collection of names, but we need an unambiguous notation to traverse such a multiple relationship and we use the **select-from-where** clause to handle collections just as in SQL.

```
select c.name
from c in p.children
```

The result of this query is a value of type Bag<String>. If we want to get a Set, we simply drop duplicates, like in SQL by using the **distinct** keyword.

```
select distinct c.name
from c in p.children
```

Now we have a means to navigate from any object to any other object following any relationship and entering any complex subvalues of an object.

For instance, we want the set of addresses of the **children** of each **Person** of the database. We know the collection named **Persons** contains all the persons of the database. We have now to traverse two collections: **Persons** and **Person::children**. Like in SQL, the **select-from** operator allows us to query more than one collection. These collections then appear in the **from** part. In OQL, a collection in the **from** part can be derived from a previous one by following a path which starts from it, and the answer is:

```
select c.lives_in.building.address
from p in Persons,
     c in p.children
```

This query inspects all children of all persons. Its result is of the type Bag<Address>.

- ***Predicate***

Of course, the **where** clause can be used to define any predicate which then serves to select the data matching the predicate. For instance, to restrict the previous result to the people living on Main Street, and having at least 2 children who do not live in the same apartment as their parents, the query is:

```
select c.lives_in.building.address
from p in Persons,
     c in p.children
where
   p.lives_in.building.address.street
          = "Main Street" and
   count(p.children) >= 2 and
   c.lives_in != p.lives_in
```

- ***Join***

In the **from** clause, collections which are not directly related can also be declared. As in SQL, this allows us to compute “joins” between these collections. For instance, to find the people living in a street and having the same name as this street, we do the following: the **Building** extent is not defined in the schema, so we have to compute it from the **Apartments** extent. To compute this intermediate result, we need a **select-from** operator again. So the join is done as follows:

```
select p
from p in Persons,
     b in (select distinct a.building
           from a in Apartments)
where p.name = b.address.street
```

This query highlights the need for an optimizer. In this case, the inner select subquery must be computed once and not for each person!

Data manipulation

A major difference between OQL and SQL is that an object query language must manipulate complex values. OQL can therefore create any complex value as a final result, or inside the query as intermediate computation.

To build a complex value, OQL uses the constructors **struct**, **set**, **bag**, **list** and **array**. For example, to obtain the addresses of the children of each person, along with the address of this person, we use the following query:

```
select struct(me:  p.name,
     my_address:
       p.lives_in.building.address,
     my_children:
      (select struct(
         name: c.name,
         address:
           c.lives_in.building.address)
       from c in p.children))
from p in Persons
```

This gives, for each person, the name, the address, and the name and address of each child. The type of the result is a bag of the following struct:

```
struct{
     String me;
     Address my_address;
     Bag<struct{String name;
                  Address address}> my_children;
}
```

OQL can also create complex objects. For this purpose, it uses the name of a class as a constructor. Attributes of the object of this class can be initialized explicitly by any valid expression.

For instance, to create a new building with 2 apartments, if there is a type name in the schema, called **List_apart**, defined by:

tydedef List<<Ref<Apartment> > List_apart;

the query is:

```
Building(
   address:
       Address (number: 10,
                street: "Main street"),
   apartments:
     List_apart(list(Apartment(number: 1),
                     Apartment(number: 2))))
```

Method invoking

OQL allows method calls with or without parameters anywhere the result type of the method matches the expected type in the query. In case the method has no parameter, the syntax for method call is the same as for accessing an attribute or traversing a relationship. If the method has parameters, these are given between parenthesis. This flexible syntax frees the user from knowing whether the property is stored (an attribute) or computed (a method). For instance, to get the age of the oldest child of **“Paul”**, we write the following query:

```
select max(select c.age
           from c in p.children)
from p in Persons,
where p.name = "Paul"
```

Of course, a method can return a complex object or a collection and then its call can be embedded in a complex path expression. For instance, inside a building **b**, to know who inhabits those least expensive apartment, we use the following path expression:

```
b.less_expensive.is_used_by.name
```

Although **less_expensive** is a method we “traverse” it as if it were a relationship.

Polymorphism

A major contribution of object technology is the possibility of manipulating polymorphic collections, and thanks to the “late binding” mechanism, to carry out generic actions on the elements of these collections. For instance, the set **Persons** contains objects of class **Person**, **Employee** and **Student**. So far, all the queries against the **Persons** extent dealt with the three possible classes of objects of the collection. A query is an expression whose operators operate on typed operands. It is correct if the type of operands matches those required by the operators. In this sense, OQL is a typed query language. This is a necessary condition for an efficient query optimizer. When a polymorphic collection is filtered (for instance **Persons**), its elements are statically known to be of that class (for instance **Person**). This means that a property of a subclass (attribute or method) cannot be applied to such an element, except in two important cases: late binding to a method, or explicit class indication.

- ***Late binding***

To list the activities of each person, we use the following query:

```
select p.activities
from p in Persons
```

activities is a method which has 3 incarnations, one for **Student**, one for **Employee** and one for generic **Person**. Depending on the kind of person of the current **p**, the right incarnation is called.

- ***Class indicator***

To go down the class hierarchy, a user may explicitly declare the class of an object that cannot be inferred statically. The interpreter then has to check at runtime, that this object actually belongs to the indicated class (or one of its subclasses).

For example, assuming we know that only “students” spend their time in following a course of study, we can select those persons and get their grade. We explicitly indicate in the query that these persons are students:

```
select ((Student)p). grade
from p in Persons
where  "course of study" in p.activities
```

Operator composition

OQL is a purely functional language: all operators can be composed freely as long as the type system is respected. This is why the language is so simple and its manual so short. This philosophy is different from SQL, which is an ad-hoc language whose composition rules are not orthogonal to the type system. Adopting a complete orthogonality, makes the language easier to learn without losing the SQL style for simple queries. Among the operators offered by OQL but not yet introduced, we can mention the set operators (**union**, **intersect**, **except**), the universal (**forall**) and existential quantifiers (**exists**), the **order by** and **group by** operators and the aggregative operators (**count**, **sum**, **min**, **max** and **avg**).

To illustrate this free composition of operators, let us write a rather elaborate query. We want to know the name of the street where the set of employees living on that street and have the smallest average salary, compared to the sets of employees living in other streets. We proceed step by step and use the **define** OQL instruction to evaluate temporary results.

OQL by example : Operator composition

1. Build the extent of class **Employee** (not supported directly by the schema)

```
define Employees as
     select (Employee) p
     from p in Persons
     where "has a job" in p.activities
```

2. Group the employees by street and compute the average salary in each street

```
define salary_map as
select street,
       average_salary: avg (select p.e.salary
                                   from partition p)

from e in Employees
group by e.lives_in.building.address.street
```

The **group by** operator splits the employees into partitions, according to the criterion (the name of the street where this person lives). The **select** clause computes, in each partition, the average of the salaries of the employees belonging to this partition.

The result of the query is of type:

```
Bag<struct{String street;
           float average_salary;}>
```

3. Sort this set by salary

```
define sorted_salary_map as
             select s from s in salary_map
             order by s.average_salary
```

The result is of type:

```
List<struct{String street;
            float average_salary;}>
```

4. Now get the smallest salary (the first in the list) and take the corresponding street name. This is the final result.

```
sorted_salary_map[0].street
```

In a single query, we could have written:

```
(select street,
        average_salary: avg (select p.e.salary
                             from partition p)
from e in (select (Employee) p
           from p in Persons
           where "has a job" in p.activities)

group by e.lives_in.building.address.street
order by avg (select p.e.salary from partition p))
[0]. street
```

4 OQL Reference

This chapter gives the full referencial information of the object query language OQL.

The chapter is divided into the following sections:

- Introduction
- Principles
- Language Definition
- Syntactical Abbreviations
- OQL BNF

The information given below is the same as that of the ODMG standard[1] with notes added on how to use this language with O_2.

1. The Object Database Standard: ODMG - 93. Atwood, Duhl, Ferran, Loomis and Wade. Edited by R.G.G. Cattell. © 1996 Morgan Kaufman Publishers.

4

4.1 Introduction

In this chapter, a formal and complete definition of the language is given. For each feature of the language, we give the syntax, its semantics, and an example. Alternate syntax for some features are described in Section 4.4, which completes OQL in order to accept any syntactical form of SQL.

The chapter ends with the formal syntax which is given in Section 4.5

4.2 Principles

Our design is based on the following principles and assumptions:

- OQL relies on the ODMG object model.
- OQL is a superset of the standard SQL part which allows you to query a database. Thus, any *select* SQL sentence which runs on relational tables, works with the same syntax and semantics on collections of ODMG objects. Extensions concern Object Oriented notions, like complex objects, object identity, path expression, polymorphism, operation invocation, late binding etc...
- OQL provides high-level primitives to deal with sets of objects but does not restrict its attention to this collection construct. Thus, it also provides primitives to deal with structures, lists, arrays, and treats all such constructs with the same efficiency.
- OQL is a functional language where operators can freely be composed, as soon as the operands respect the type system. This is a consequence of the fact that the result of any query has a type which belongs to the ODMG type model, and thus can be queried again.
- OQL is not computationally complete. It is an easy to use query language which provides easy access to an object database.
- Based on the same type system, OQL can be invoked directly from within programming languages for which an ODMG binding is defined, e.g., C++. Conversely, OQL can invoke operations programmed in these languages.
- OQL does not provide explicit update operators but rather can invoke operations defined on objects for that purpose, and thus does not breach the semantics of an Object Database which, by definition, is managed by the "methods" defined on the objects.
- OQL provides declarative access to objects. Thus OQL queries can be easily optimized by virtue of this declarative nature.
- The formal semantics of OQL can easily be defined.

4.3 Language Definition

OQL is an "expression" language. A query expression is built from typed operands composed recursively by operators. We will use the term *expression* to designate a valid query in this section.

4.3.1 Query Program

A query program consists of a (possibly empty) set of query definition expressions followed by an expression, which is evaluated as the query itself. The set of query definition expressions is non recursive (although a query may call an operation which issues a query recursively).

For example:

```
define jones as select distinct x from Students x
   where x.name = "Jones";
select distinct  student_id from jones
```

This defines the set **jones** of students named Jones, and evaluates the set of their **student_ids**.

O_2 note

With the O_2 query interpreter you use CTRL-D (on Unix) or CTRL-Z (On Windows) between two queries rather that ";".

4.3.2 Named Query Definition

If **q** is an identifier and **e** is a query expression, then **define q as e** is a query definition expression which defines the query with name **q**.

Example:

```
define Does as select x from Student  x
where x.name ="Doe"
```

This statement defines **Does** as a query returning a bag containing all the students whose name is Doe.

```
define Doe as element(select x from Student  x
where x.name="Doe")
```

This statement defines **Doe** as a query which returns the student whose name is Doe (if there is only one, otherwise an exception is raised).

O_2 note

- define operation is available only with the interactive query interpreter. It has no meaning for OQL embedded in programming languages (C++, Smalltalk, O_2C) because standard programming language variables can be used for that purpose.
- A defined name is valid up to the next commit or abort
- You can get the list of current defined queries by typing the query: display queries

4.3.3 Elementary Expressions

4.3.3.1 Atomic Literals

If l is an atomic literal, then l is an expression whose value is the literal itself.

Literals have the usual syntax:

- Object Literal: nil
- Boolean Literal: false, true
- Integer Literal: sequence of digits, e.g, 27
- Float Literal: mantissa/ exponent. The exponent is optional, e.g., 3.14 or 314.16e-2
- Character Literal: character between simple quotes, e.g., 'z'
- String Literal: character string between double quote, e.g.,"a string"

4.3.3.2 Named Objects

If **e** is a named object, then **e** is an expression. It defines the entity attached to the name.

Example:

```
Students
```

This query defines the set of students. We have assumed here that the name **Students** exists which corresponds to the extent of objects of the class **Student**.

4.3.3.3 Iterator Variable

If **x** is a variable declared in a from part of a select-from-where..., then **x** is an expression whose value is the current element of the iteration over the corresponding collection.

4.3.3.4 Named Query

If **define q as e** is a query definition expression, then **q** is an expression.

Example:

```
Doe
```

This query returns the student with name Doe. It refers to the query definition expression declared in Section 4.3.2.

4.3.4 Construction Expressions

4.3.4.1 Constructing Objects

If **t** is a type name, $p_1, p_2, ..., p_n$ are properties of **t**, and $e_1, e_2, ..., e_n$ are expressions, then $t\ (p_1: e_1 ..., p_n: e_n)$ is an expression.

This defines a new object of type **t** whose properties $p_1, p_2, ..., p_n$ are initialized with the expressions $e_1, e_2, ..., e_n$. The type of e_i must be compatible with the type of p_i.

If **t** is a type name of a collection and **e** is a collection literal, then **t(e)** is a collection object. The type of **e** must be compatible with **t**.

Examples:

```
Employee (name: "Peter", boss: Chairman)
```

This creates a mutable Employee object.

```
vectint (set(1,3,10))
```

This creates a mutable set object (assuming that *vectint* is the name of a class whose type is Bag<int>).

4.3.4.2 Constructing Structures

If p_1, p_2, ..., p_n are property names, and e_1, e_2, ..., e_n are expressions, then

struct (p_1: e_1, p_2: e_2, ..., p_n: e_n)

is an expression. It defines the structure taking values e_1, e_2, ..., e_n on properties p_1, p_2, ..., p_n.

Note that this dynamically creates an instance of the type **struct(p_1: t_1, p_2: t_2, ..., p_n: t_n)** if t_i is the type of e_i.

Example:

```
struct(name: "Peter", age: 25);
```

This returns a structure with two attributes **name** and **age** taking respective values **Peter** and **25**.

See also abbreviated syntax in some contexts, in Section 4.4.1.

4.3.4.3 Constructing Sets

If e_1, e_2, ..., e_n are expressions, then **set(e_1, e_2, ..., e_n)** is an expression. It defines the set containing the elements e_1, e_2, ..., e_n. It creates a set instance.

Example:

```
set(1,2,3)
```

This returns a set consisting of the three elements 1, 2, and 3.

4.3.4.4 Constructing Lists

If e_1, e_2, ..., e_n are expressions, then

list(e_1, e_2, ..., e_n) or simply **(e_1, e_2, ..., e_n)**

are expressions. They define the list having elements e_1, e_2, ..., e_n. They create a list instance.

If min, max are two expressions of integer or character types, such that min < max, then

`list(min .. max)` or simply **`(min .. max)`**

are expressions whose value is: list(min, min+1, ... max-1, max)

Example:

```
list(1,2,2,3)
```

This returns a list of four elements.

Example:

```
list(3 .. 5)
```

This returns the list(3,4,5)

O_2 note

In O_2 the keyword list is mandatory.

4.3.4.5 Constructing Bags

If **e_1, e_2, ..., e_n** are expressions, then **bag(e_1, e_2, ..., e_n)** is an expression. It defines the bag having elements **e_1, e_2, ..., e_n**. It creates a bag instance.

Example:

```
bag(1,1,2,3,3)
```

This returns a bag of five elements.

4.3.4.6 Constructing Arrays

If **e_1, e_2, ..., e_n** are expressions, then **array(e_1, e_2, ..., e_n)** is an expression. It defines an array having elements **e_1, e_2, ..., e_n**. It creates an array instance.

Example:

```
array(3,4,2,1,1)
```

This returns an array of five elements.

4.3.5 Atomic Types Expressions

4.3.5.1 Unary Expressions

If **e** is an expression and **<op>** is a unary operation valid for the type of **e**, then **<op> e** is an expression. It defines the result of applying **<op> to e**.

Arithmetic unary operators are: +, -, abs

Boolean unary operator is: not.

Example:

```
not true
```

This returns **false**.

4.3.5.2 Binary Expressions

If e_1 and e_2 are expressions and **<op>** is a binary operation, then **e_1<op>e_2** is an expression. It defines the result of applying **<op>** to e_1 and e_2.

Arithmetic integer binary operators are: +, -, *, / , mod (modulo)

Floating point binary operators are: +, -, *, /

Relational binary operators are: =, !=. <. <=, >, >=

These operators are defined on all atomic types.

Boolean binary operators are: and, or

Example:

```
count(Students) - count(TA)
```

This returns the difference between the number of students and the number of TAs.

4.3.5.3 String Expressions

If s_1 and s_2 are expressions of type string, then

s_1 || s_2, and s_1 + s_2

are equivalent expressions of type string whose value is the concatenation of the two strings.

O_2 note

In O_2 the operator || is not accepted. To concatenate 2 strings use "+".

If **c** is an expression of type character, and **s** an expression of type string, then

```
c in s
```

is an expression of type boolean whose value is true if the character belongs to the string, else false.

If **s** is an expression of type string, and **i** is an expression of type integer, then

```
s[i]
```

is an expression of type character whose value is the i+1th character of the string.

If **s** is an expression of type string, and **low** and **up** are expressions of type integer, then

```
s[low:up]
```

is an expression of type string whose value is the substring of **s** from the low+1 th character up to the up+1 th character.

If **s** is an expression of type string, and **pattern** a string literal which may include the wildcard characters: "**?**" or "_", meaning any character, and "*****" or "**%**", meaning any substring including an empty substring, then

```
s like pattern
```

is an expression of type boolean whose value is true if **s** matches the pattern, else false.

Example:

```
'a nice string'  like '%nice%str_ng'
```

is true.

O_2 note

In O_2 the only supported wildcard is "*".

4.3.6 Object Expressions

4.3.6.1 Comparison of Mutable Objects

If e_1 and e_2 are expressions which denote mutable objects (objects with identity) of the same type, then

$e_1 = e_2$ and $e_1 \mathrel{!=} e_2$

are expressions which return a boolean. The second expression is equivalent to **not(e_1 = e_2)**.

$e_1 = e_2$ is true if they designate the same object.

Example:

```
Doe = element(select s from Students s
where s.name = "Doe")
```

is true.

4.3.6.2 Comparison of Immutable Objects

If e_1 and e_2 are expressions which denote immutable objects (literals) of the same type, then

$e_1 = e_2$ and $e_1 \mathrel{!=} e_2$

are expressions which return a boolean. the second expression is equivalent to

not(e_1 = e_2).

$e_1 = e_2$ is true if the value e_1 is equal to the value e_2.

4.3.6.3 Extracting an Attribute or Traversing a Relationship from an Object

If **e** is an expression, if **p** is a property name, then **e->p** and **e.p** are expressions. These are alternate syntax to extract the property **p** of an object **e**.

If **e** happens to designate a deleted or a non existing object, i.e. **nil**, an attempt to access the attribute or to traverse the relationship raises an exception. However, a query may test explicitly if an object is different from **nil** before accessing a property.

Example:

```
Doe.name
```

This returns **Doe**.

Example:

```
Doe->spouse != nil and Doe->spouse->name = "Carol"
```

This returns **true**, if Doe has a spouse whose name is Carol, or else **false**.

O_2 note

According to a recent evolution of the ODMG standard, OQL does not now raise an exception when it traverses a path which contains a **nil**. Instead of this, a predicate involving such a path is always **false**. This means that OQL now skips such elements and thus the explicit test to **nil** is not yet mandatory.

4.3.6.4 Applying an Operation to an Object

If **e** is an expression, if **f** is an operation name, then

e->f and **e.f**

are expressions. These are alternate syntax to apply on operation on an object. The value of the expression is the one returned by the operation or else the object **nil**, if the operation returns nothing.

Example:

```
jones->number_of_students
```

This applies the operation **number_of_students** to **jones**.

4.3.6.5 Applying an Operation with Parameters to an Object

If **e** is an expression, if e_1, e_2 , ..., e_n are expressions, if **f** is an operation name, then

e->f(e_1, e_2, ..., e_n) and **e.f**(e_1, e_2, ..., e_n)

are expressions that apply operation **f** with parameters e_1, e_2, ..., e_n to object **e**. The value of the expression is the one returned by the operation or else the object nil, if the operation returns nothing.

In both cases, if e happens to designate a deleted or a non existing object, i.e. **nil**, an attempt to apply an operation to it raises an exception.

However, a query may test explicitly if an object is different from **nil** before applying an operation.

Example:

```
Doe->apply_course("Maths", Turing)->number
```

This query calls the operation **apply_course** on class Student for the object Doe. It passes two parameters, a string and an object of class Professor. The operation returns an object of type Course and the query returns the number of this course.

4.3.6.6 Dereferencing an Object

If **e** is an expression which denotes an object with identity (a mutable object), then ***e** is an expression which delivers the value of the object (a literal).

Example:

Given two variables of type Person, **p1** and **p2**, the predicate

p1 = p2

is true if both variables refer to the same object, while

***p1 =*p2**

is true if the objects have the same values, even if they are not the same objects.

4.3.7 Collections Expressions

4.3.7.1 Universal Quantification

If **x** is a variable name, e_1 and e_2 are expressions, e_1 denotes a collection and e_2 a predicate, then

for all x in e_1**:** e_2

is an expression. It returns **true** if all the elements of collection e_1 satisfy e_2 and **false** otherwise.

Example:

```
for all x in Students: x.student_id > 0
```

This returns **true** if all the objects in the **Students** set have a positive value for their **student_id** attribute. Otherwise it returns **false**.

4.3.7.2 Existential Quantification

If **x** is a variable name, if $\mathbf{e_1}$ and $\mathbf{e_2}$ are expressions, $\mathbf{e_1}$ denotes a collection and $\mathbf{e_2}$ a predicate, then

exists x in $\mathbf{e_1}$**:** $\mathbf{e_2}$

is an expression. It returns **true** if there is at least one element of collection $\mathbf{e_1}$ that satisfies $\mathbf{e_2}$ and **false** otherwise.

Example:

```
exists x in Doe.takes: x.taught_by.name = "Turing"
```

This returns **true** if at least one course Doe takes is taught by someone named Turing.

If **e** is a collection expression, then

exists(e) and **unique(e)**

are expressions which return a boolean value. The first one returns true if there exists at least one element in the collection, while the second one returns true, if there exists only one element in the collection.

Notice that these operators allow the acceptance of the SQL syntax for nested queries such as:

select ... from col where exists (select ... from $\mathbf{col_1}$ **where predicate)**

The nested query returns a bag to which the operator exists is applied. This is of course the task of an optimizer to recognize that it is useless to compute effectively the intermediate bag result.

O_2 note

In O_2 these two last operations are not supported. Only the form "exists x in e1: e2" is valid.

4.3.7.3 Membership Testing

If $\mathbf{e_1}$ and $\mathbf{e_2}$ are expressions, $\mathbf{e_2}$ is a collection, $\mathbf{e_1}$ has the type of its elements, then

$\mathbf{e_1}$ in $\mathbf{e_2}$

is an expression. It returns **true** if element $\mathbf{e_1}$ belongs to collection $\mathbf{e_2}$.

Example:

```
Doe in Does
```

This returns **true**.

4.3.7.4 Aggregate Operators

If **e** is an expression which denotes a collection, if **<op>** is an operator from **{min, max, count, sum, avg}**, then **<op>(e)** is an expression.

Example:

```
max (select salary from Professors)
```

This returns the maximum salary of the Professors.

4.3.8 Select From Where

If e_1, e_2, ..., e_n are expressions which denote collections, and x_1, x_2, ..., x_n are variable names, if **e'** is an expression of type boolean, and if *projection* is an expression or the character *, then

select *projection* **from** e_1 **as** x_1, e_2 **as** x_2 .., e_n **as** x_n **where e'**

and

select distinct *projection* **from** e_1 **as** x_1, e_2 **as** x_2 ..., e_n **as** x_n **where e**

are expressions.

The result of the query is a set for a **select distinct** or a bag for a **select.**

If you assume e_1, e_2, ..., e_n are all set and bag expressions, then the result is obtained as follows: take the cartesian product[1] of the sets e_1, e_2, ..., e_n; filter that product by expression **e**' (i.e., eliminate from the result all objects that do not satisfy boolean expression **e'**); apply the *projection* to each one of the elements of this filtered set and get the result. When the result is a set (distinct case) duplicates are automatically eliminated.

The situation where one or more of the collections e_1, e_2, ..., e_n is an indexed collection is a little more complex. The select operator first converts all these collections into sets and applies the previous operation. The result is a set (distinct case) or else a bag. So, in this case, we simply transform each of the e_i's into a set and apply the previous definition.

4.3.8.1 Projection

Before the projection, the result of the iteration over the *from* variables is of type

1. The cartesian product between a set and a bag is defined by first converting the set into a bag, and then getting the resulting bag which is the cartesian product of the two bags.

bag< struct(x_1: type_of(e_1 elements), ... x_n: type_of(e_n elements)) >

The projection is constructed by an expression which can then refer implicitly to the "current" element of this bag, using the variables x_i. If for e_i neither explicit nor implicit variable is declared, then x_i is given an internal system name (which is not accessible by the query anyway).

By convention, if the projection is simply "*", then the result of the selection is the same as the result of the iteration.

If the projection is "distinct *", the result of the select is this bag converted into a set.

In all other cases, the projection is explicitly computed by the given expression.

Example:

```
select couple(student: x.name, professor: z.name)
from Students as x,
     x.takes as y,
     y.taught_by as z
where z.rank = "full professor"
```

This returns a bag of objects of type **couple** giving student names and the names of the full professors from which they take classes.

Example:

```
select  *
from Students as x,
      x.takes as y,
      y.taught_by as z
where z.rank = "full professor"select  *
```

This returns a bag of structures, giving for each student "object", the section object followed by the student and the full professor "object" teaching in this section:

bag< struct(x: Student, y: Section, z: Professor) >

4.3.8.2 Iterator Variables

A variable, x_i, declared in the *from* part ranges over the collection e_i and thus has the type of the elements of this collection. Such a variable can be used in any other part of the query to evaluate any other expressions (see the Scope Rules in Section 4.3.15). Syntactical variations are accepted for declaring these variables, exactly as with SQL. The *as* keyword may be omitted. Moreover, the variable itself can be omitted, and in this case, the name of the collection itself serves as a variable name to range over it.

Example:

```
select couple(student: Students.name, professor: z.name)
from Students,
      Students.takes y,
      y.taught_by z
where z.rank = "full professor"
```

O_2 note

In O_2 an additional syntax is allowed to declare a variable x:

"... from x in collection ...".

This syntax will also be included in the next release of the ODMG standard.

4.3.8.3 Predicate

In a select-from-where query, the *where* clause can be omitted, with the meaning of a true predicate.

4.3.9 Group-by Operator

If *select_query* is a select-from-where query, *partition_attributes* is a structure expression and *predicate* a boolean expression, then

select_query **group by** *partition_attributes*

is an expression and

select_query **group by** *partition_attributes* **having** *predicate*

is an expression.

The cartesian product visited by the select operator is split into partitions. For each element of the cartesian product, the partition attributes are evaluated. All elements which match the same values according to the given partition attributes, belong to the same partition. Thus the partitioned set, after the grouping operation is a set of structures: each structure has the valued properties for this partition (the valued *partition_attributes)*, completed by a property which is conventionally called *partition* and which is the bag of all objects matching this particular valued partition.

If the partition attributes are:

att_1: e_1, att_2: e_2, ... , att_n: e_n ,

then the result of the grouping is of type

```
set< struct(att1: type_of(e1), att2: type_of(e2),...,
            attn: type_of(en),
             partition: bag< type_of(grouped elements) >)
```

The type of grouped elements is defined as follows.

If the *from* clause declares the variables **v_1** on collection **col_1, v_2** on **col_2, ..., v_n** on **col_n**, the grouped elements form a structure with one attribute "**v_k**" for each collection having the type of the elements of the corresponding collection.

```
partition: bag< struct(v1: type_of(col1 elements), ... ,
                       vn: type_of(coln elements))>.
```

If a collection **col_k** has no variable declared the corresponding attribute has an internal system name.

This partitioned set may then be filtered by the predicate of a *having* clause. Finally, the result is computed by evaluating the *select* clause for this partitioned and filtered set.

The *having* clause can thus apply aggregate functions on *partition*, likewise the *select* clause can refer to *partition* to compute the final result. Both clauses can refer also to the partition attributes.

Example:

```
select  *
    from  Employees e
group by  low:     e.salary < 1000,
          medium:  e.salary >= 1000 and salary < 10000,
          high:    e.salary >= 10000
```

This gives a set of three elements, each of which has a property called **partition** which contains the bag of employees that enter in this category. So the type of the result is:

```
set<struct(low: boolean, medium: boolean, high: boolean,
           partition: bag<struct(e: Employee)>)>
```

The second form enhances the first one with a *having* clause which enables you to filter the result using aggregative functions which operate on each partition.

Example:

```
select  department,
       avg_salary: avg(select p.e.salary from partition p)
from Employees e
group by department: e.deptno
having avg(select p.e.salary from partition p) > 30000
```

This gives a set of couples: department and average of the salaries of the employees working in this department, when this average is more than 30000. So the type of the result is:

bag<struct(department: integer, avg_salary: float)>

O_2 note

In O_2 the syntax of *partition_attributes* does not accept the keyword struct and thus is always given as a list of criteria separated by commas. See Section 4.4.1.

4.3.10 Order-by Operator

If *select_query* is a select-from-where or a select-from-where-group_by query, and if e_1, e_2, ..., e_n are expressions, then

select_query **order by** e_1, e_2, ..., e_n

is an expression. It returns a list of the selected elements sorted by the function e_1, and inside each subset yielding the same e_1, sorted by e_2, ... , and the final subsub...set, sorted by e_n.

Example:

```
select p from Persons p order by p.age, p.name
```

This sorts the set of persons on their age, then on their name and puts the sorted objects into the result as a list.

Each sort expression criterion can be followed by the keyword asc or desc, specifying respectively an ascending or descending order. The default order is that of the previous declaration. For the first expression, the default is ascending.

Example:

```
select * from p in Persons
order by p.age desc, p.name asc, p.department
```

4.3.11 Indexed Collection Expressions

4.3.11.1 Getting the i-th Element of an Indexed Collection

If e_1 and e_2 are expressions, e_1 is a list or an array, e_2 is an integer, then $e_1[e_2]$ is an expression. This extracts the e_2**+1 th** element of the indexed collection e_1. Notice that the first element has the rank 0.

Example:

```
list (a,b,c,d) [1]
```

This returns **b.**

Example:

```
element (select x
         from Courses x
         where x.name = "math" and
               x.number ="101").requires[2]
```

This returns the third prerequisite of Math 101.

4.3.11.2 Extracting a Subcollection of an Indexed Collection.

If e_1**,** e_2, and e_3 are expressions, e_1 is a list or an array, e_2 and e_3 are integers, then $e_1[e_2:e_3]$ is an expression. This extracts the subcollection of e_1 starting at position e_2 and ending at position e_3.

Example:

```
list (a,b,c,d) [1:3]
```

This returns **list (b,c,d)**.

Example:

```
element (select x
         from Courses x
         where x.name="math" and
               x.number="101").requires[0:2]
```

This returns the list consisting of the first three prerequisites of Math 101.

4.3.11.3 Getting the First and Last Elements of an Indexed Collection

If **e** is an expression, if **<op>** is an operator from **{first, last}**, **e** is a list or an array, then **<op>(e)** is an expression. This extracts the first and last element of a collection.

Example:

```
first(element(select x
      from Courses x
      where x.name="math" and
            x.number="101").requires)
```

This returns the first prerequisite of Math 101.

4.3.11.4 Concatenating Two Indexed Collections

If $\mathbf{e_1}$ and $\mathbf{e_2}$ are expressions, if $\mathbf{e_1}$ and $\mathbf{e_2}$ are both lists or both arrays, then $\mathbf{e_1+e_2}$ is an expression. This computes the concatenation of $\mathbf{e_1}$ and $\mathbf{e_2}$.

Example:

```
list (1,2) + list( 2,3)
```

This query generates **list (1,2,2,3)**.

4.3.12 Binary Set Expressions

4.3.12.1 Union, Intersection, Difference

If e_1 and e_2 are expressions, if **<op>** is an operator from **{union, except, intersect},** if e_1 and e_2 are sets or bags, then e_1 **<op>** e_2 is an expression. This computes set theoretic operations, union, difference, and intersection on e_1 and e_2, as defined in Chapter 2.

When the collection kinds of the operands are different (bag and set), the set is converted into a bag beforehand and the result is a bag.

Examples:

```
Student except Ta
```

This returns the set of students who are not Teaching Assistants.

```
bag(2,2,3,3,3) union bag(2,3,3,3)
```

This bag expression returns **bag(2,2,3,3,3,2,3,3,3)**

```
bag(2,2,3,3) intersect bag(2,3,3,3)
```

The intersection of 2 bags yields a bag that contains the minimum for each of the multiply values. So the result is: **bag(2,3,3)**

```
bag(2,2,3,3,3) except bag(2,3,3,3)
```

This bag expression returns bag(2)

4.3.12.2 Inclusion

If e_1 and e_2 are expressions which denote sets or bags, if <op> is an operator from {<, <=, >, >=}, then e_1 <op> e_2 is an expression whose value is a boolean.

When the operands are different kinds of collections (bag and set), the set is first converted into a bag.

e_1 < e_2 is true if **e_1** is included into **e_2** but not equal to **e_2**

e_1 <= e_2 is true if **e_1** is included into **e_2**

Example:

```
set(1,2,3) < set(3,4,2,1)
```

is true.

4.3.13 Conversion Expressions

4.3.13.1 Extracting the Element of a Singleton

If **e** is a collection-valued expression, **element(e)** is an expression. This takes the singleton **e** and returns its element. If **e** is not a singleton this raises an exception.

Example:

```
element(select x from Professors x
where x.name ="Turing")
```

This returns the professor whose name is **Turing** (if there is only one).

4.3.13.2 Turning a List into a Set

If **e** is a list expression, **listtoset(e)** is an expression. This converts the list into a set, by forming the set containing all the elements of the list.

Example:

```
listtoset (list(1,2,3,2))
```

This returns the set containing 1, 2, and 3.

O_2 note

To carry out the reverse operation (set to list) you use the order by operator. If you are not interested in a given order you can use "*" as shown in the following query:

```
select e from e in aSet order by *
```

4.3.13.3 Removing Duplicates

If **e** is an expression whose value is a collection, then

```
distinct(e)
```

is an expression whose value is the same collection after removing the duplicated elements. If **e** is a bag, **distinct(e)** is a set. If **e** is an ordered collection, the relative ordering of the remaining elements is preserved.

4.3.13.4 Flattening Collection of Collections

If **e** is a collection-valued expression, **flatten(e)** is an expression. This converts a collection of collections of **t** into a collection of **t**. So this flattening operates at the first level only.

Assuming the type of **e** to be $\mathbf{col_1}$**<**$\mathbf{col_2}$**<t>>,**

the result of **flatten(e)** is:

- If $\mathbf{col_2}$ is a set (resp. a bag), the union of all $\mathbf{col_2}$**<t>** is done and the result is a **set<t>** (resp. **bag<t>**)
- If $\mathbf{col_2}$ is a list (resp. an array) and $\mathbf{col_1}$ is a list (resp. an array) as well, the concatenation of all $\mathbf{col_2}$**<t>** is done following the order in $\mathbf{col_1}$ and the result is $\mathbf{col_2}$**<t>**, which is thus a list (resp. an array). Of course duplicates, if any, are maintained by this operation.
- If $\mathbf{col_2}$ is a list or an array and $\mathbf{col_1}$ is a set or a bag, the lists or arrays are converted into sets, the union of all these sets is done and the result is a **set<t>**, therefore without duplicates.

Examples:

```
flatten(list(set(1,2,3), set(3,4,5,6), set(7)))
```

This returns the set containing 1,2,3,4,5,6,7.

```
flatten(list(list(1,2), list(1,2,3)))
```

This returns **list(1,2,1,2,3).**

```
flatten(set(list(1,2), list(1,2,3)))
```

This returns the set containing 1,2,3.

4.3.13.5 Typing an Expression

If **e** is an expression, if **c** is a type name, then **(c)e** is an expression. This asserts that **e** is an object of class type **c.**

If it turns out that it is not true, an exception is raised at runtime. This is useful to access a property of an object which is statically known to be of a superclass of the specified class.

Example:

```
select ((Employee) s).salary
from Students s
where s in (select sec.assistant from Sections sec)
```

This returns the set of salaries of all students who are teaching assistants, assuming that **Students** and **Sections** are the extents of the classes **Student** and **Section**.

4.3.14 Function Call

If f is a function name, if e_1 , e_2 , ..., e_n are expressions, then

f() and f(e1, e2, ... , en)

are expressions whose value is the value returned by the function, or the object **nil**, when the function does not return any value. The first form allows you to call a function without a parameter, while the second one calls a function with the parameters e_1, e_2, ..., e_n.

OQL does not define in which language the body of such a function is written. This feature allows you to smoothly extend the functionality of OQL without changing the language.

4.3.15 Scope Rules

The *from* part of a select-from-where query introduces explicit or implicit variables to range over the filtered collections. An example of an explicit variable is:

select ... from Persons p ...

while an implicit declaration would be:

select ... from Persons ...

The scope of these variables reaches all parts of the select-from-where expression including nested sub-expressions.

The *group by* part of a select-from-where-group_by query introduces the name *partition* along with possible explicit attribute names which characterize the partition. These names are visible in the corresponding *having* and *select* parts, including nested sub-expressions within these parts.

Inside a scope, you use these variable names to construct path expressions and reach properties (attributes and operations) when these variables denote complex objects. For instance, in the scope of the first from clause above, you access the age of a person by **p.age**.

When the variable is implicit, as in the second from clause, you use the name of the collection directly, **Persons.age**.

However, when there is no ambiguity, you can use the property name directly as a shortcut, without using the variable name to open the scope (this is made implicitly), writing simply: **age**. There is no ambiguity when a property name is defined for one and only one object denoted by a visible variable.

To summarize, a name appearing in a (nested) query is looked up in the following order:

- a variable in the current scope, or
- a named query introduced by the *define* clause, or
- a named object, i.e., an entry point in the database, or
- an attribute name or an operation name of a variable in the current scope, when there is no ambiguity, i.e., this property name belongs to only one variable in the scope.

Example:

Assuming that in the current schema the names Persons and Cities are defined.

```
select  scope1
from Persons,
     Cities c
     where exists(select scope2 from children as child)
           or count (select scope3, (select scope4 from
                                         partition)
                       from  children p,
                           scope5 v
                     group by age: scope6
                     )
```

In *scope1*, we see the names: Persons, c, Cities, all property names of class Person and class City as soon as they are not present in both classes, and they are not called "Persons", "c", nor "Cities".

In *scope2*, we see the names: child, Persons, c, Cities, the property names of the class City which are not property of the class Person. No attribute of the class Person can be accessed directly since they are ambiguous between "child" and "Persons".

In *scope3*, we see the names: age, partition, and the same names from scope1, except "age" and "partition", if they exist.

In *scope4*, we see the names: age, partition, p, v, and the same names from scope1, except "age", "partition", "p" and "v", if they exist.

In *scope5*, we see the names: p, and the same names from scope1, except "p", if it exists.

In *scope6*, we see the names: p, v, Persons, c, Cities, the property names of the class City which are not property of the class Person. No attribute of the class Person can be accessed directly since they are ambiguous between "child" and "Persons".

O_2 note

Implicit attribute scope is not available with O_2. You must always access an attribute with the dot notation: **`v.att`**.

4.4 Syntactical Abbreviations

OQL defines an orthogonal expression language, in the sense that all operators can be composed with each others as soon as the types of the operands are correct. To achieve this property, we have defined a functional language with simple (like +) or composite operators (like select from where group_by order_by) which always deliver a result in the same type system and which thus can be recursively operated with other operations in the same query.

In order to accept the whole DML query part of SQL, as a valid syntax for OQL, OQL is added some ad-hoc constructions each time SQL introduces a syntax which cannot enter in the category of true operators. This section gives the list of these constructions that we call "abbreviations", since they are completely equivalent to a functional OQL expression which is also given. Doing that, we thus give at the same time the semantics of these constructions, since all operators used for this description have already been defined.

4.4.1 Structure Construction

The structure constructor was introduced in Section 4.3.4.2. Alternate syntax are allowed in two contexts: select clause and group-by clause.

In both contexts, the SQL syntax is accepted, along with the one already defined.

select *projection* {, *projection*} ...

select ... group by *projection* {, *projection*}

where *projection* is in one of the following forms:

- (i) expression as identifier
- (ii) identifier: expression
- (iii) expression

This is an alternate syntax for:

struct(identifier: expression {, identifier: expression})

If there is only one *projection* and the syntax (iii) is used in a select clause, then it is not interpreted as a structure construction but rather the expression stands as it is. Furthermore, a (iii) expression is only valid if it is possible to infer the name of the corresponding attribute (the identifier). This requires that the expression denotes a path expression (possibly of length one) ending in a property whose name is then chosen as the identifier.

Example:

```
select p.name,  salary,  student_id

    from Professors p,  p.teaches
```

This query returns a bag of structures:

bag<struct(name: string, salary: float, student_id: integer)>

O_2 note

O_2 accepts the 3 alternatives of the *projection* syntax in the **select** part, as well as the **struct** syntax. In the **group by** part, O_2 accepts the 3 alternatives but does not accept the **struct** syntax.

4.4.2 Aggregate Operators

These operators were introduced in Section 4.3.7.4. SQL adopts a notation which is not functionnal for them. So OQL accepts this syntax too.

If we define *aggregate* as one of min, max, count, sum and avg,

select count(*) from ...

is equivalent to: **count(select * from ...)**

select *aggregate***(query) from ...**

is equivalent to: *aggregate***(select query from ...)**

select *aggregate***(distinct query) from ...**

is equivalent to: *aggregate***(distinct(select query from ...)**

O_2 note

O_2 does not support Aggregate Operator abbreviations.

4.4.3 Composite Predicates

If $\mathbf{e_1}$ and $\mathbf{e_2}$ are expressions, $\mathbf{e_2}$ is a collection, $\mathbf{e_1}$ has the type of its elements, if *relation*

is a relational operator (=, !=, <, <=, > , >=), then

$\mathbf{e_1}$ *relation* **some** $\mathbf{e_2}$ and $\mathbf{e_1}$ *relation* **any** $\mathbf{e_2}$ and $\mathbf{e_1}$ *relation* **all** $\mathbf{e_2}$

are expressions whose value is a boolean.

The two first predicates are equivalent to:

exists x in $\mathbf{e_2}$**:** $\mathbf{e_1}$ *relation* **x**

The last predicate is equivalent to:

for all x in $\mathbf{e_2}$**:** $\mathbf{e_1}$ *relation* **x**

Example:

```
10 < some (8,15, 7, 22)
```

is true

O_2 note

In O_2 Composite Predicate abbreviations are not supported.

4.4.4 String Literal

OQL accepts simple quotes as well to delimit a string (see Section 4.3.3.1), as SQL does. This introduces an ambiguity for a string with one character which then has the same syntax as a character literal. This ambiguity is solved by context.

O_2 note

In O_2 a string must be delimited by double quotes.

4.5 OQL BNF

The OQL grammar is given using a BNF-like notation.

- { symbol } means a sequence of 0 or more symbol(s).
- *[*symbol*]* means an optional symbol. Do not confuse with the separators [ ]
- **keyword** is a terminal of the grammar. Keywords are not case sensitive.
- xxx_name has the syntax of an identifier
- xxx_literal is self explanatory, e.g., "a string" is a string_literal
- bind_argument stands for a parameter when embedded in a programming language, e.g., $3i.

The non terminal **`query`** stands for a valid query expression. The grammar is presented as recursive rules producing valid queries. This explains why most of the time this non terminal appears on the left side of ::=. Of course, all operators expect their "query" operands to be of the right type. Type constraints were discussed in the previous sections.

These rules must be completed by the priority of OQL operators which is given after the grammar. Some syntactical ambiguities are solved semantically from the types of the operands.

4.5.1 Grammar

4.5.1.1 Axiom (see Sections 4.3.1, 4.3.2)

```
query_program ::={define_query;} query
define_query ::=define identifier as query
```

4.5.1.2 Basic (see Section 4.3.3)

```
query ::= nil
query ::= true
query ::= false
query ::= integer_literal
query ::= float_literal
query ::= character_literal
query ::= string_literal
query ::= entry_name
query ::= query_name
query ::= bind_argument[1]
query ::= from_variable_name
query ::= (query)
```

4.5.1.3 Simple Expression (see Section 4.3.5)

```
query ::= query + query[2]
query ::= query - query
query ::= query * query
query ::= query / query
query ::= - query
query ::= query mod query
query ::= abs (query)
query ::= query || query
```

4.5.1.4 Comparison (see Section 4.3.5)

```
query ::= query comparison_operator query
query ::= query like string_literal
comparison_operator ::= =
comparison_operator ::= !=
comparison_operator ::= >
comparison_operator ::= <
comparison_operator ::= >=
```

1. A bind argument allows to bind expressions from a programming language to a query when embedded into this language (see Chapters on language bindings).
2. The operator + is also used for list and array concatenation.

```
comparison_operator ::= <=
```

4.5.1.5 Boolean Expression (see Section 4.3.5)

```
query ::= not query
query ::= query and query
query ::= query or query
```

4.5.1.6 Constructor (see Section 4.3.4)

```
query ::= type_name ([query] )
query ::= type_name (identifier:query {, identifier: query})
query ::= struct (identifier: query {, identifier: query})
query ::= set ([query {, query}])
query ::= bag ([query {,query}])
query ::= list ([query {,query}])
query ::= (query, query {, query})
query ::=[ list](query .. query)
query ::= array ([query {,query}])
```

4.5.1.7 Accessor (see Sections 4.3.6, 4.3.11, 4.3.14, 4.3.15)

```
query ::= query dot attribute_name
query ::= query dot relationship_name
query ::= query dot operation_name
query ::= query dot operation_name( query {,query} )
dot     ::= . | ->
query ::= * query
query ::= query [query]
query ::= query [query:query]
query ::= first (query)
query ::= last (query)
query ::= function_name( [ query {,query} ] )
```

4.5.1.8 Collection Expression (see Sections 4.3.7, 4.4.3)

```
query ::= for all identifier in query: query
query ::= exists identifier in query: query
query ::= exists(query)
query ::= unique(query)
query ::= query in query
query ::= query comparison_operator quantifier query
quantifier ::=  some
quantifier ::=  any
quantifier ::=  all
query ::= count (query)
```

```
query ::= count (*)
query ::= sum (query)
query ::= min (query)
query ::= max (query)
query ::= avg (query)
```

4.5.1.9 Select Expression (see Sections 4.3.8, 4.3.9, 4.3.10)

```
query ::= select [ distinct ] projection_attributes
          from variable_declaration {, variable_declaration}
          [where query]
          [group by partition_attributes]
          [having query]
          [order by sort_criterion {, sort_criterion}]
projection_attributes ::= projection {, projection}
projection_attributes ::= *
projection ::= query
projection ::= identifier: query
projection ::= query as identifier
variable_declaration ::= query [ [ as ] identifier]
partition_attributes ::= projection {, projection}
sort_criterion ::= query [ordering]
ordering ::= asc
ordering ::= desc
```

4.5.1.10 Set Expression (see Section 4.3.12)

```
query ::= query intersect query
query ::= query union query
query ::= query except query
```

4.5.1.11 Conversion (see Section 4.3.13)

```
query ::= listtoset (query)
query ::= element (query)
query ::= distinct(e)
query ::= flatten (query)
query ::= (class_name) query
```

4.5.2 Operator Priorities

The following operators are sorted by decreasing priority. Operators on the same line have the same priority and group left-to-right.

() [] . ->

OQL BNF : Operator Priorities

not **-** (unary) **+** (unary)

in

* / mod intersect

+ **-** union except ||

< > <= >= < some < any < all (etc ... for all comparison operators)

= != like

and exists for **all**

or

.. :

,

(identifier) this is the cast operator

order

having

group by

where

from

select

4

INDEX

Symbols

A

B

C

D

Q

R

S

T

U

V

W