O_2 ODBC User Manual

Release 5.0 - April 1998

Who should read this manual

This manual presents O_2Tools, a complete graphical programming environment for the design and development of O_2 applications. It describes the browsers and editors available, as well as how to customize O_2Tools screens.

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

1 Introduction

Congratulations! You are now a user of O_2ODBC - the standard interface for accessing data in an heterogeneous environment of both relational and object database systems.

The O_2ODBC interface handles client application requests to a database and returns the database server's response.

This introductory chapter is divided as follows:

- System overview
- ODBC
- Manual Overview

1 Introduction

1.1 System overview

he system architecture of O_2 is illustrated in Figure 1.1.

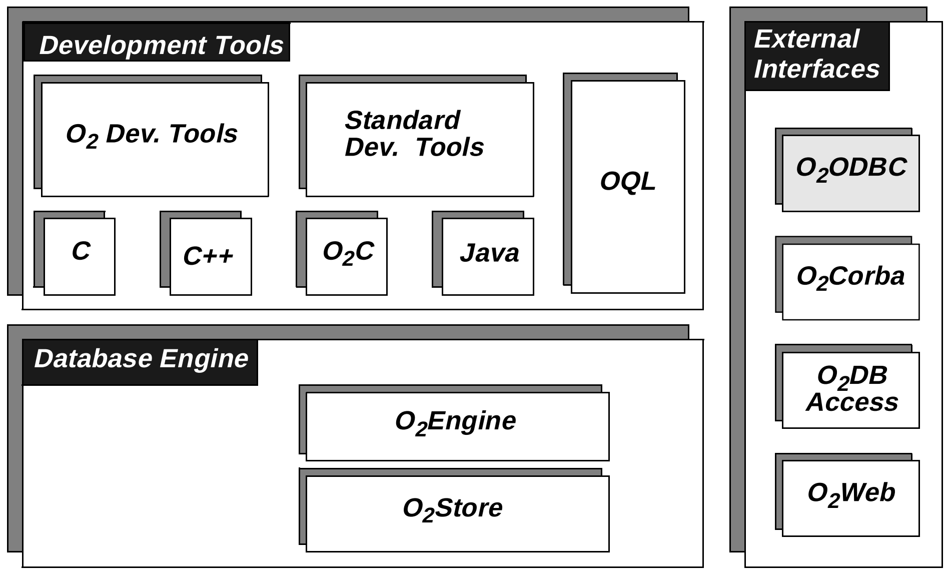


Figure 1.1: O_2 System Architecture

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.

System overview :

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.

1.2 ODBC

ODBC (Open Database Connectivity) is a standard interface for accessing data in an heterogeneous environment of relational and non-relational database management systems. Many existing tools use ODBC to access a database, e.g. Word, Excel, Delphi, etc.

An ODBC client application uses the ODBC API to request and/or send data to a database server. The ODBC driver translates client requests and server answers into a format that the DBMS server and the ODBC client can understand. The ODBC API defines a set of core functions, that correspond to the functions in the X/Open and SQL Access Group Call Level Interface specification, together with two extended sets of functionality. ODBC defines a standard SQL grammar, which drivers translate to the native SQL grammars used by various DBMSs.

1.3 O_2 and ODBC

O_2 ODBC is an ODBC driver built on top of O_2. It allows existing ODBC applications to run on top of an O_2 database and new ODBC applications to be defined on top of O_2 through the ODBC API. O_2 ODBC works on a relational view of an O_2 base.

Starting from an existing O_2 base, the ODBC application can retrieve information about the relational view derived for that base: tables, columns, primary and foreign keys, etc.

The relational view derivation is performed by a special tool and can be customized by the user through a configuration file. Many different views can be defined for a given O_2 base. O_2 ODBC implements the core ODBC API and some Level 1 and Level 2 extensions (e.g. retrieve catalog and parameter information). In addition, it supports the core SQL grammar and part of the extended grammar level (e.g. procedure calls and long data).

SQL queries (**SELECT-FROM-WHERE**) formulated on the relational view, and sent through O_2 ODBC are translated into the corresponding OQL queries. Data update SQL commands (**INSERT-UPDATE-DELETE**) are interpreted by the O_2 ODBC driver, which performs updates on the corresponding O_2 data through the O_2 API interface.

SQL catalog commands (**CREATE TABLE**, for instance) are also interpreted by the O_2 ODBC driver, which updates the O_2 database schema accordingly. Tables and views can be therefore created from scratch rather than being derived from existing O_2 classes. The two kinds of tables (system-derived and application-defined) can be combined in an ODBC application.

1.4 Manual Overview

This manual is divided into the following chapters:

- Chapter 1

  Introduces O_2 ODBC.

- Chapter 2

  Describes how to install O_2 ODBC.

- Chapter 3

  Gives an overview of O_2 ODBC.

- Chapter 4

  Describes how O_2 schemas and O_2 data are translated into equivalent SQL entities.

- Chapter 5

  Describes how to use the O_2 ODBC driver, its features and limitations.

- Chapter 6

  Show how programmers can use the **o2_Odbc** class to build their own O_2 ODBC servers.

- Chapter 7

  A reference guide for O_2ODBC.

Two appendixes complete this manual:

- Appendix A

  Gives the complete syntax for writing configuration files used for view customizing by the **o2sql_export** tool.

- Appendix B

  Gives the values returned by the **SQLGetInfo** ODBC API function for all possible **fInfoType** input argument values.

1.5 Background

We asume the reader is familiar with ODBC and O_2. The following references provide essential information:

- *ODBC SDK 2.1 Programmer's Reference*, Microsoft Development Library.
- *O_2 System version 5.x Admistration Manuals*, O_2Technology.
- *Understanding the new SQL: a complete guide*, J. Melton and A. R. Simon, Morgan Kaufamann, 1993.

1 Introduction

2 O_2ODBC Installation

This chapter addresses the installation of an O_2 ODBC driver and details the contents of the O_2 ODBC distribution package.

The reader should be familiar with the ODBC environment and related concepts.

2

O2ODBC Installation

2.1 Hardware and Software Requirements

The O_2 ODBC driver requires the following hardware:

- IBM-compatible PC
- 8 MB RAM required
- Hard Disk Space: 1.5 MB for the installation.
- The O_2 ODBC driver requires the following software:
- O_2 DBMS

  In order to access data in an O_2 database with the O_2 ODBC driver, you must have the O_2 DBMS version 5.x. For information on software and hardware requirements for the O_2 DBMS version 5.x, refer to the O_2 System Administration Manuals.

- Microsoft Windows 95 or Microsoft Windows NT
- Network software

  A network is required to connect the platforms on which the O_2 ODBC client and O_2 ODBC server reside. For information on the software and hardware required by your network, see its documentation.

2.2 O_2 ODBC Distribution Package

The O_2 ODBC distribution package contains the following:

- The Dynamic Link Libraries (DLL) **libo2dri.dll** and **libo2com.dll.**
- The **setup.exe** program.
- The **o2open_dispatcher** program.
- The **o2odbc_dump_base** program.
- The **o2odbc_server** program.

- The **o2sql_export** program.
- The **o2sql_query** program.
- The **o2_Odbc.hxx** include files.
- The **libsql.so** and **libo2odbc_svr.so** libraries.

  These libraries are used by the different O_2 ODBC related tools and are necessary to build a user specific O_2 ODBC server.

2.3 Setting up the O_2 ODBC Driver

The installation procedure described below assumes that you have dumped the contents of the O_2 ODBC distribution package to the disk of the Windows 95 or Windows NT station where the driver is to be installed. The structure of the O_2 ODBC distribution package is the following:.

```
o2odbc
   include
      o2_Odbc.hxx
   install
     o2driver
         libo2com.dll
         libo2dri.dll
      odbc32
         setup.exe
         odbcad32.exe
         ...
   lib
      libsql.so
      libo2odbc_svr.so
    bin
      o2open_dispatcher
      o2odbc_dump_base
      o2odbc_server
      o2sql_export
      o2sql_query
   doc
      o2odbc_manual.pdf
```

2 O2ODBC Installation

Installing the driver

Go to the sub-directory **o2odbc/install/odbc32** and run the program **setup.exe**. This program will prompt you for confirmation and then install the O_2 ODBC driver by copying all the ODBC components needed to run the driver in the system directories.

At the end of the installation process, the setup program prompts you to declare new data sources on installed drivers. You can declare O_2 data sources at this point or, if you prefer, you will be able to manage your data sources using the ODBC administrator program **odbcad32.exe** located in the same directory.

Declaring the o2 open-dispatcher

The declaration of the **o2open_dispatcher** is a two steps process:

- Declare in the **O2OPEN_DISPATCHER** variable the name of the machine on which the dispatcher is running.

  On Windows NT, open the control panel program, choose the system icon and select the “**environment**” pane. You can then add the new variable.

  On Windows 95, declare the variable in your **autoexec.bat** file by adding the following line: **set O2OPEN_DISPATCHER=<machine name>**.

- Declare in the services file the port on which the dispatcher is reachable.

  On Windows NT, edit the file **<WINDIR>/system32/drivers/etc/services** and add the following line: **o2opendispatcher <port number>/tcp**.

  On Windows 95, edit the file **<WINDIR>/services** and add the following line: **o2opendispatcher <port number>/tcp**.

Installing the tools

Once the driver and dispatcher have been declared on the client side, the O_2 ODBC tools must be installed on the server side, i.e. on the machine where the O_2 database system is installed. Assuming the environment variable **O2HOME** denotes the O_2 installation directory, the following completes the installation of O_2 ODBC:

```
cp o2odbc/include/* $O2HOME/include;

cp o2odbc/bin/* $O2HOME/<platform>/bin;

cp o2odbc/lib/* $O2HOME/<platform>/lib;
```

2.4 Adding, Modifying and Deleting O_2 ODBC Data Sources

On the client side, an O_2 data source is added, modified and deleted using the standard ODBC Administrator tool. In the Data Sources dialog box of this tool, a new data source can be added by clicking on the Add button. Assuming the O_2 ODBC driver has been already installed, it can be selected from the Installed ODBC Drivers list that is displayed in the Add Data Source dialog box. An O_2 ODBC Setup dialog box is displayed to allow the option values to be set and the data source definition to be completed on the client side.

Modification and deletion of O_2 data sources are carried out in an analogous way, by following the appropriate options from the Data Sources dialog box of the ODBC Administrator tool.

On the O_2 ODBC server side, a data source corresponds to an O_2 base on which a view schema generated with the **o2sql_export** tool from the schema of the base has been generated. For more details on data sources, see Section 5.2.

2 O2ODBC Installation

3 O_2 ODBC Overview

This chapter is an introduction to the main O_2 ODBC concepts. It gives an overview of the driver architecture and describes its main components.

This chapter provides an overview of the O_2 ODBC driver architecture and the way it works.

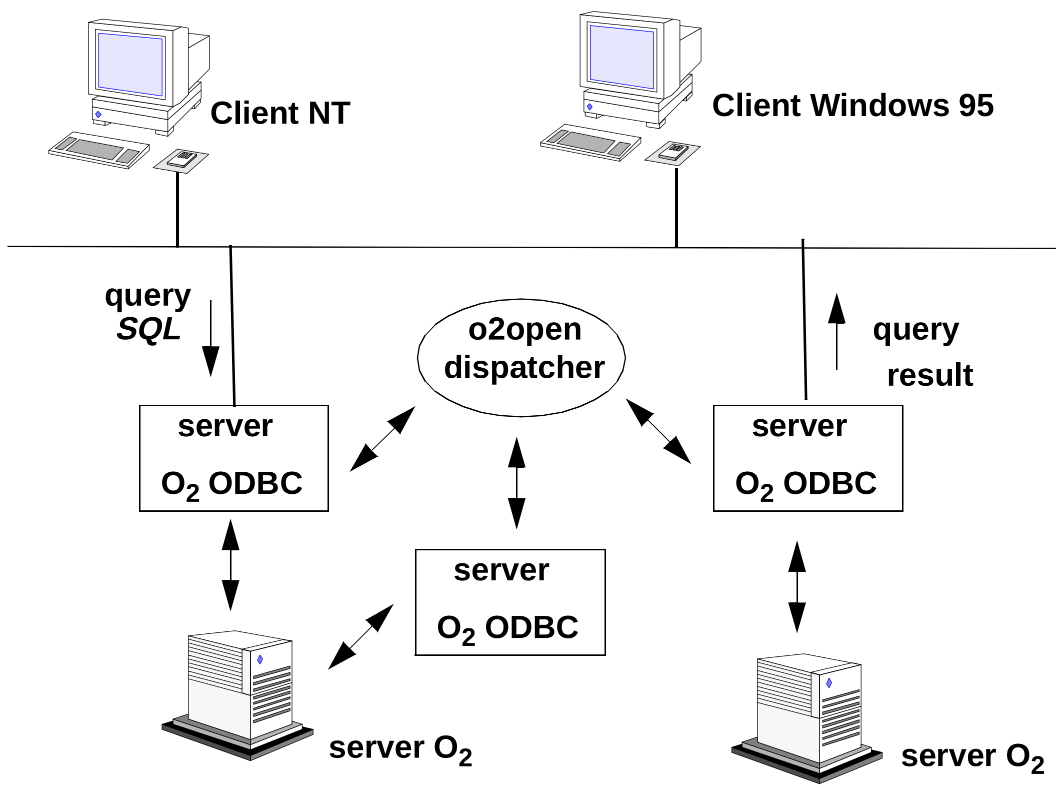
We assume the reader is familiar with the ODBC environment and related concepts and with O_2 general architecture and related concepts.

3

O2 ODBC Overview

3.1 O_2 ODBC Architecture

The architecture of the O_2 ODBC product is depicted in the Figure below:



We identify the following main elements in this architecture:

- an O_2 server

  This is the standard **o2server** program.

- an O_2 ODBC server

  This is the **o2odbc_server** program which is connected to an O_2 server.

- an O_2 OpenDispatcher dispatcher

  This is the standard **o2open_dispatcher** program.

Outline of the O_2 ODBC driver activity

An O_2 ODBC driver works in the following way:

1. An ODBC client requests a connection to an O_2 ODBC data source.
2. The O_2 ODBC client library connects (through SQLConnect or SQLDriverConnect ODBC API functions) to an O_2OpenAccess dispatcher running on the local area network.
3. The O_2 OpenAccess dispatcher tells the ODBC client which O_2 ODBC server to connect to.
4. The ODBC client connects to the appropriate O_2 ODBC server.
5. Once the connection has been established, the ODBC client will use the ODBC API appropriate functions (e.g. **`SQLExecute`**) to access data in the data source to which it is connected.
6. The O_2 ODBC server processes the client requests. It is connected to an O_2 server and performs query translation and execution.
7. The O_2 ODBC server returns dataquery result data to the client on demand (e.g. **`SQLBind`**, **`SQLFetch`**, **`SQLGetData`**).

Advantages of the O_2 ODBC architecture

The architecture of the O_2 ODBC driver provides numerous features that enhance the applications performance:

- Multi-threading

  O_2 ODBC allows an application to use multiple threads in order to concurrently perform different treatments. The module provides some synchronization functions that allow client application developers to use multiple threads in the client part while protecting the application from forbidden resources access violation.

- Load-balancing

  The dispatcher module is an independent module used to route connections from an ODBC client to an ODBC server and to preserve an efficient load-balancing (static and dynamic load-balancing) among the network. Its role is to manage a pool of ODBC servers available throughout the network to answer clients requests.

- Flexible deployment

  O_2 ODBC allows to distribute the application among multiple machine if necessary, thus offering an easy way to support scalability. Multiple ODBC servers can be run on different machines, the user load being distributed among these machines according to criteria like current load, machine characteristics, etc. This location-independent model makes it easy to change deployment schemes as the application grows. As demand grows, other O_2 ODBC servers can be added on other machines, and the demand can be distributed among those servers without any code changes.

In the remaining of this chapter, we give an overview of the two main components of the O_2 ODBC driver architecture, namely the O_2 SQL component and the O_2 ODBC server.

3.2 O_2 SQL

We denote by O_2 SQL the module of the O_2 ODBC architecture implementing the schema and query translation capabilities of the driver. This module is basically composed of the O_2 SQL library **`libsql.so`** together with two development tools **`o2sql_export`** and **`o2sql_query`** that can be used independently of the O_2 ODBC driver.

The O_2 SQL library is used by the O_2 ODBC driver server component. It implements the schema and query translation services necessary to allow O_2 data to be accessed through SQL. O_2 SQL is built on top of OQL and the O_2 Engine.

The **`o2sql_export`** tool is used to derive relational views from O_2 schemas. Such a view must be derived prior to any access to O_2 with ODBC. The activation of a relational view on an O_2 base allows such base to be seen as a relational database. Objects stored in O_2 are perceived as tuples in relational tables (an object can spawn more than one tuple in more than one table). It should be noted that such tables exist only virtually in the virtual database resulting from a view activation on an O_2 base.

The **`o2sql_query`** tool is an interactive shell allowing SQL commands to be executed on a virtual database. This can be a useful tool for quickly inspecting view schemas and databases and tunning SQL applications running on O_2 .

Given the separation between the O_2 SQL and the O_2 ODBC driver implementation, it is possible to see and query O_2 data as relational data through SQL without using an ODBC interface. An API function **`o2_sql`**, analogous to the standard O_2 API function **`o2_oql`** can be used to execute SQL queries from a given O_2 Engine API program.

O_2 SQL is detailed in Chapter 4.

3.3 O_2 ODBC Server

The O_2 ODBC server is built on top of O_2 SQL.

The server processes client requests. These requests are issued through the ODBC API. SQL queries sent by a client to be executed on a O_2 data source are translated by the server into an equivalent OQL query and executed on the O_2 base to which the client is connected.

A server can handle requests of different clients working on different data sources. Before processing the request of a given client, the server activates the client's data source, i.e. it activates the appropriate view on the O_2 base to which the client is connected.

A server uses the query translation services implemented in the O_2 SQL library. It performs, in addition, all the ODBC specific processing (data conversions, cursor management, catalog data retrieval, etc) necessary to respond to ODBC clients requests.

The O_2 ODBC server is detailed in Chapter 5.

3 O2 ODBC Overview

4 O_2 SQL

The O_2 SQL library and related tools are defined as a separate product and used by the O_2 ODBC server.

O_2 SQL provides two main services for applications wanting to access O_2 databases through SQL: a schema and a query translator.

This chapter presents how to define sophisticated SQL views of O_2 data instead of using the default view derivation rules, so as to adapt the relational structure to the needs of a particular application.

To customize the way a relational view of an O_2 schema is derived, Section 4.1 and, in particular, Section “View customization” provide useful information.

To formulate complex queries and updates operations on O_2 data through the SQL interface, Section 4.2 gives some hints on how to perform and optimize such operations. In particular, Section "Schema Update Commands" should be read by those wanting to populate an O_2 schema through SQL (with table creation commands), whereas Section "Data Update Commands" provides useful information for those wanting to create and update O_2 data through SQL.

All users wishing to access O_2 data through SQL should read Section 4.3.

We assume the reader is familiar with the O_2 and relational data models, as well as with SQL related concepts in general.

4 O2 SQL

4.1 Schema Translation

An object model captures semantics of application domains in a more elaborate way and it promotes adherence to normal forms. Relational schemas derived from a richer object model *tend to be* in third normal form. The database designer can thus benefits from the expressiveness and extensibility, among others, of an object model and still implements such a model in terms of well normalized relational tables.

The structure of data stored in an O_2 base is defined in the schema of that base. The schema contains class and type definitions, as well as data entry points (names that play the role of roots of persistence).

As SQL queries can be formulated on relational data only, it is necessary to provide a *relational view* of O_2 data to be able to query such data with SQL.

A relational view schema is derived from an O_2 schema with the **o2sql_export** tool. We say that the O_2 schema is exported to SQL. When a view is derived, its definition is kept by the O_2 system in an internal structure called the SQL catalog (see “View creation tool o2sql_export” on page 61). Many different view schemas can be derived for a given O_2 schema. All views derived for a given O_2 schema can be activated on every base instance of that schema.

4.1.1 SQL View Schema

We recall below the main features of the O_2 and relational schemas, before considering the translation of an instance of the former into one of the latter.

O_2 Schema

An O_2 schema is composed of a set of class definitions. A class can inherit from a number of classes. To each class a type is associated. Valid type constructors include *set*, *list* and *tuple* constructors, that can be applied recursively to define arbitrarily complex types from class types (each class defines a type) and atomic types (*integer*, *real*, *char*, *string*, *boolean*). Methods can be defined in a class to be applied on its instances.

Relational Schema

A relational schema is composed of a set of table definitions. Each table is composed of a set of columns, each of a given atomic type. A subset of the columns of a table can be declared as a primary key serving to uniquely identify rows in the table. Foreign keys can point to primary keys, allowing references to be established among rows in tables.

Since data are not stored on the relational database, performance is not an issue and we do not pay attention to table fragmentation (vertical partitioning). Nevertheless, we try to reduce the number of collection tables in order to simplify query formulation. In that sense, we decided to unnest tuple attributes instead of generating collection tables corresponding to *weak entities* in the relational schema.

Example

Let us consider an example of schema translation before considering the different aspects of the schema translation process in turn.

The O_2 schema **document** given in Figure 4.1 models information about articles, their authors and respective affiliations. The relational schema obtained from the O_2 schema in Figure 4.1 is given in Figure 4.2.

Each class is mapped to an homonymous table. For table **Article**, attribute **title** is a primary key and **date_title** is a foreign key pointing to table **Date**. Attributes of the nested tuple attribute **address** in class **Author** are unnested in the corresponding table. Set and list attributes are mapped to the so called collection tables **Article_authors** and **Article_sections**. Such tables associate to each instance of `Article` the corresponding set of authors and list of sections respectively. For the list attribute, an additional attribute pos represents the position of elements in the list.

An SQL client knowing the relational schema above can formulate queries on it. Such queries are translated into OQL queries that are evaluated on an O_2 base instance of the original schema.

Class translation

Each O_2 class is mapped into a so called *class table*. A column **title** of default type **LONGVARCHAR** is defined by default in the corresponding table and declared as a primary key, unless a different logical key is declared for the class in the configuration file.

A class must have an associated homonymous named set defined in the schema and modeling its extent in the O_2 base. If, however, such a named set is not explicitly defined in the O_2 schema, a virtual extent (i.e. an OQL query giving a set of object of the class as a result) must be provided in a configuration file used to derive the view schema, as it will be described later.

.

```
schema document;
class Article type tuple(
   title:string,
   authors:set(Author),
   sections:list(Section),
   date:Date)
end;

class Author type tuple(
   name:string,
   address:tuple(institute:Institute,email:string))
end;

class Institute type tuple(
   name:string,
   country:string)
end;

class Section type tuple(
   title:string,
   contents:string)
end;

class Date type tuple(
   day:integer,
   month:integer,
   year:integer)
end;

name Articles:set(Article);
```

Figure 4.1: O_2 schema **document**

Primary Key Definition

When mapping object structures to relational tuples, we must define primary keys so as to be able to uniquely identify objects when they are queried through SQL in their tuple form. In O_2 , object identifiers are not available to the user. They are used internally by the system at the object store level and are not externalized.

In O_2 , each class defines a method **`title`**, inherited from class **`Object`**, which by default gives the name of the class of the object. This method can be redefined in a subclass as a method or an attribute that gives a different value for each object of the class, therefore playing the role of *logical identifier* of the object.

.

```
CREATE SCHEMA document

CREATE TABLE Article( title LONGVARCHAR
       date_title LONGVARCHAR PRIMARY KEY ( title )
       FOREIGN KEY ( date_title ) REFERENCES Date )

CREATE TABLE Author( title LONGVARCHAR name LONGVARCHAR
       address_institute_title LONGVARCHAR
       address_email LONGVARCHAR
       PRIMARY KEY ( title )
       FOREIGN KEY ( address_institute_title )
       REFERENCES Institute )

CREATE TABLE Date(
       title LONGVARCHAR day INTEGER month INTEGER
       year INTEGER PRIMARY KEY ( title ) )

CREATE TABLE Section(
       title LONGVARCHAR title LONGVARCHAR ~
       contents LONGVARCHAR PRIMARY KEY ( title ) )

CREATE TABLE Institute(
       title LONGVARCHAR name LONGVARCHAR
       country LONGVARCHAR PRIMARY KEY ( title ) )

CREATE TABLE Article_authors(
       Article_title LONGVARCHAR
       authors_title LONGVARCHAR
       FOREIGN KEY ( Article_title )
       REFERENCES Article
       FOREIGN KEY ( authors_title )
       REFERENCES Author )

CREATE TABLE Article_sections(
       Article_title LONGVARCHAR
       pos INTEGER
       sections_title LONGVARCHAR
       PRIMARY KEY ( Article_title, pos )
       FOREIGN KEY ( Article_title )
       REFERENCES Article
       FOREIGN KEY ( sections_title )
       REFERENCES Section )
```

Figure 4.2: Relational schema **document**

The **title** method or attribute is thus translated by default as a primary key in the corresponding table but a different attribute or method can be declared to be used as a key in the configuration file.

Remark 1: It is up to the object schema designer to guarantee that the value returned by such a method or attribute remains constant, at least during an SQL section on the object database. Such identifier should be completely independent on changes to the object value and on physical location. In practice, we require the value of a logical identifier to depend on constant attributes, i.e. attributes that are not likely to change after object creation, and, in our framework, attributes that are not likely to be updated by an SQL statement.

Remark 2: The choice between an attribute or a method key is an important one, as SQL queries matching rows based on their primary key columns will be mapped into OQL queries retrieving objects based on the corresponding key attributes or methods. If attributes are used instead of methods to identify objects in O_2 , then indexes on such attributes can be defined to optimize the query evaluation.
Remark 3: The method `title`, or any other method declared as a logical key in the configuration file, can return the *external identifier* of the object on which it is applied. This identifier can be provided by O_2 Engine on user’s request.

Attribute Translation

Attributes in a tuple-typed class represent relationship among objects and values. These can be one-to-one, one-to-many or many-to-many relationships. The simpler case, that of atomic and object attributes, correspond to one-to-one relationships and are directly translated as columns in the corresponding derived table. We consider them first before looking at complex attributes (tuple and collection attributes).

Atomic and Class Attributes

Each attribute having an atomic or class type in the O_2 class becomes a column in the corresponding table with a type given by the type mapping defined below.

OQL type t	$T_{SQL}(t)$
`integer`	INTEGER
`real`	REAL
`char`	CHARACTER
`string`	LONGVARCHAR
`bytes`	LONGVARBINARY
`boolean`	CHARACTER
`class`	LONGVARCHAR

The column takes its name from the attribute name, unless a renaming is defined in the configuration file.

An attribute pointing to another object (attribute of class type) becomes a foreign key referencing the table corresponding to the class of the pointed object.

For attributes having a complex type we consider two possibilities: collection and tuple attributes.

Tuple Attributes

Tuple types are unnested and have their attributes incorporated to the table corresponding to the type structure where the tuple type occurs. Tuple attributes are renamed before being merged, i.e. the outer tuple attribute name is prefixed to each attribute name of the nested tuple to avoid naming conflicts. Such attributes can also be renamed by the user through the configuration file.

Merging attribute tuples with their pointing objects leads to relational queries that are easier to formulate. In addition, tuple attributes are values and, opposite to object attributes, cannot be shared, and placing them in an external auxiliary table would be pointless in that sense.

Collection Attributes

A collection type attribute is translated into a so-called collection table. The type of the elements of the collection are translated recursively as columns in the collection table (or other collection tables, for collections nested in collections).

A set attribute models a one-to-many or a many-to-many relationship. A one-to-many relationship can be merged with a participating object (all objects in the set can point to the composite object). Although the choice on how to map collection attributes could have been let to the user, we decided to model all such aggregations as distinct tables, for expediency of implementation of the query translator.

The collection table corresponding to a set attribute is named according to the name of the class where the attribute is defined and the name of the attribute itself, unless it is explicitly renamed in the configuration file. A column *class_name* + *_title* is defined to hold the logical identifier of the composite object (the object holding the set). The other columns of the collection table are derived based on the type of elements of the set. For a set **s** of atomic values, a column **s** with the corresponding element type is defined to hold elements of the set. For a set **s** of objects, an attribute **`s_title`** is defined to hold the logical identifiers of elements. For a set **s** of tuples, each attribute a_i of the tuple is treated recursively and merged to the collection table as column **`s_ai`**, as for ordinary tuple attributes. For nested collections, an extra collection table is derived recursively. Default key attribute names for the key columns generated in the collection tables can be renamed through the configuration file as usual.

List attributes are also mapped to collection tables, as for set attributes, but an additional column *pos* of type *INTEGER* is defined to hold the position of each element in the list.

Remark: In the ODMG C++ binding, collections are imported as O_2 classes having a collection type. In order to allow C++ applications to use the O_2 / SQL binding, such classes will be treated by the query translator as special classes to allow a direct access to the encapsulated collections. For instance, a class K which encapsulates an O_2 list will be mapped into two tables: TABLE K and TABLE K_List. TABLE K holds the object itself, while TABLE K_List holds its value.

Inherited Attributes

What about inheritance? We consider two alternatives: (1) merging inherited attributes with attributes defined locally in the subclass to derive the corresponding relation; (2) deriving a relation with only locally defined attributes plus a foreign key pointing to the corresponding tuple in the table derived for each superclass. Again, our choice is dictated by the specificities of the problem in hand: since derived tables are not (at least in principle) used to store data, but exist only virtually, we decided to collapse inherited attributes in the table derived from a given subclass. The resulting tables are not normalized but are far easier to query.

Data Retrieval Methods

In addition to the columns derived to hold the values of attributes defined in the tuple type of a class, columns can be derived to hold values returned by *data retrieval methods*. The choice of which such methods to import is left to the user, as they must be explicitly declared in the configuration file.

As far as visibility is concerned, only read and public attributes (and methods) should be derived as columns in the corresponding table, in order to preserve data encapsulation and rules out unauthorized access through the relational interface.

4.1.2 View Customization

The relational schema in Figure 4.2 results from a so-called *default mode* translation.

In the default mode, tables and columns are named from their corresponding class and attribute names and the existence of a default logical **`title`** method or attribute is assumed for every class. In addition, for query translation purposes, for each class **C**, the system assumes the existence of an homonymous named set **C** modelling the extent of the class.

Customized translation

In a *customized mode*, the schema translator takes into account some user-supplied information used for the generation of the view schema (and consequently for the translation of SQL queries into OQL).

View customization includes the following possibilities:

- hiding of classes

  By default, every class defined in the O_2 schema is derived as a table in the view schema, unless it is hidden in the configuration file used to derive the view.

- hiding of attributes

  By default, every attribute of a tuple typed class is derived as column (or possibly as a collection table if it is a collection attribute) in the view schema, unless it is hidden in the configuration file.

- renaming of classes

  If a class renaming is not specified in the configuration file, a table takes the same name as the class from which it is derived.

- renaming of columns

  If an attribute renaming is not specified in the configuration file, a column takes the same name as the attribute from which it is derived.

- virtual class extents

  If a named set is not explicitly defined in the O_2 schema, a virtual extent (i.e. an OQL query giving a set of object of the class as a result) can be provided in the configuration file. This is not mandatory, unless the table is to be used in the **FROM** clause of a given SQL query.

- export of data retrieval methods as columns

  Data retrieval methods are methods without input parameter and with an output parameter. Such methods can be translated into table columns as if they were attributes.

- definition of alternative logical keys

By default, the **title** method or attribute is exported as a primary key, but an alternative attribute and/or method can be declared as a logical key for a given class in the configuration file.

- data update authorization

  The configuration file can also be used to authorize data updates on tables generated from O_2 classes. By default, such updates (i.e. insertion, updates and deletions) are not authorized.

- stored procedures

  O_2 C and C++ functions can be declared as stored procedures in the configuration file to be called through the SQL interface.

- redefinition of collection tables

  The default naming rules used in the derivation of collection tables from collection attributes can be redefined to allow different table and column names to be used.

A view schema can be customized through a configuration file provided to the schema export tool at view creation or update. Appendix A gives the complete syntax used to specify configuration files.

Remark: Hiding and renaming of classes and attributes together with the selective importing of methods allow the entire object structure to be customized to meet the needs of a given SQL application. Many different views of the same schema can be defined, allowing different virtual databases to be derived from a given O_2 base.

Example 4.2.1 The configuration file shown in Figure 4.3 is specified using the syntax given in Appendix A.

Attributes that are not hidden nor redefined are exported using the default translation rules. Methods declared in a **method** clause are exported as virtual attributes (e.g. method **year** in class **Article** is exported as **annee** in the corresponding table). The name of exported methods is redefined through the **redefine** clause. The resulting view schema is given in Figure 4.4 below.

.

```
view schema french_document from document;

hide Section, Date, Institute;

stored procedure articles_from_author;

export class Article
     hide
         sections, date;
     redefine
         title as titre,
         year as annee;
     method
         year;
     extent
         "Articles";
     with insert,update,delete;
end;

export class Author as Auteur
     hide address.institute;
     redefine
         name as nom,
         address_email as adresse_eletronique;
     extent
         "select x
         from Articles a,
         a->authors x";
end;

export collection Article.authors as Auteurs
     redefine
         authors_title as auteur,
         Article.Article_title as article;
end;
```

Figure 4.3: Configuration file for schema translation

```
CREATE SCHEMA french_document

CREATE TABLE Article(
       titre LONGVARCHAR
       annee INTEGER
       PRIMARY KEY ( title ) )

CREATE TABLE Auteur(
       title LONGVARCHAR nom LONGVARCHAR
       adresse_eletronique LONGVARCHAR
       PRIMARY KEY ( title ) )

CREATE TABLE Auteurs(
       article LONGVARCHAR auteur LONGVARCHAR
       FOREIGN KEY (article) REFERENCES Article
       FOREIGN KEY (auteur) REFERENCES Auteur )
```

Figure 4.4: Relational schema **french_document**

4.2 Query Translation

4.2.1 Schema Update Commands

In this section, we consider the translation of schema update SQL commands into O_2 schema update commands.

Schema update SQL commands are commands for table, view and index creation, deletion and modification. The translation of such commands corresponds to an update of the underlying O_2 schema.

Table creation command

Currently, a simple translation mechanism is used for generating, for each newly created table, a corresponding class in the O_2 schema with the same attributes and using a default type mapping. Simple reference attributes, i.e. one-to-one relationships, are generated in the the O_2 class by taking primary and foreign key definitions into account, as the information provided in the table creation command is not enough for allowing the system to infer one-to-many or many-to-many relationships.

Example 4.3.1 Let us consider the creation of tables **Proceedings** and **Proceedings_articles** in schema document:

```
CREATE TABLE Proceedings(
                  title LONGVARCHAR,
                  editor LONGVARCHAR,
                  date LONGVARCHAR
                  PRIMARY KEY ( title ),
                  FOREIGN KEY ( date )
                  REFERENCES Date )

CREATE TABLE Proceedings_articles(
                  proceedings LONGVARCHAR,
                  pos INTEGER NOT NULL,
                  article LONGVARCHAR
                  PRIMARY KEY ( proceedings, pos ),
                  FOREIGN KEY ( proceedings
                  REFERENCES Proceedings,
                  FOREIGN KEY ( article )
                  REFERENCES Article )
```

The two table creation commands above are translated into the following class and name creation commands in the O_2 schema:

```
class SQL_Proceedings type tuple(
   title:string,
   editor:string,
   date:Date)
end;

class SQL_Proceedings_articles type tuple(
  proceedings:SQL_Proceedings,
  pos:integer,
  article:Article);

name SQL_Proceedings:unique set(SQL_Proceedings);

name SQL_Proceedings_articles:set(
                  SQL_Proceedings_articles);
```

The name of the O_2 class generated is defined as follows: the prefix **SQL_** is added to the table name. Also, for each created table, a name (the same identifier as for the class name is used) is created to model the table extent. Each time an insertion is performed in such a table, an object of the corresponding class is created and inserted into the corresponding named collection.

Definitions prefixed with **SQL_** in an O_2 schema should not be modified through O_2 but only through the SQL interface. Changes to class **SQL_Proceedings**, for instance, should be performed only indirectly through the **ALTER TABLE** command. Modifications to such classes and names can be nevertheless performed (i.e. the O_2 system does not prevent them) at the risk of making the SQL catalog inconsistent.

References to a class (in a hide or export clause) whose name is prefixed by **SQL_** are not taken into account in the configuration file. In other words, the configuration file cannot be used to customize classes generated via a **CREATE TABLE** command, as these classes are not exported in the same way as O_2 classes are exported.

Constraints associated to column definitions (e.g. **NOT NULLX**, **DEFAULT**, **CHECK**, etc) are automatically checked at insertions and updates.

Remark: **NULL** values are not supported. All class and collection tables have the **NOT NULL** constraint added systematically to all columns. In addition, the null predicate (**IS NULL**) always evaluates to false (respectively, **IS NOT NULL** always evaluates to true).

View table creation command

SQL view table definitions are recorded in the SQL catalog. At query translation time references to a view table are replaced by the query used in the view table definition.

Example 4.3.2 Let us consider the following view table defined in schema **document**:

```
CREATE VIEW Recent_Articles(
        title LONGVARCHAR, year INTEGER ) AS
   SELECT art.title, dat.year
   FROM Article art
        Date dat
   WHERE art.date_title = dat.title
     AND dat.year > 90
```

Now let us consider the following query on this view and its corresponding OQL query:

SQL query:	OQL query:

```
SQL query:                      OQL query:
SELECT art.title                select struct(title:art.title)
FROM Recent_Article art         from
WHERE art.year = 95                  (select tuple(title:art.title,
                                                  year:a.date.year)
                                      from Articles art
                                      where art.date.year > 90) art
                                 where art.year = 95
```

An optimized version of the OQL query above that eliminates the nested selection will be generated by the OQL query interpreter at runtime.

Table deletion command

When a table is dropped through the **DROP TABLE** command, the corresponding class and name are both deleted from the O_2 schema.

A table cannot be dropped if there are indexes defined on it or if other tables reference it through a foreign key.

View deletion command

When a view table is dropped through the **DROP VIEW** command, the corresponding view definition is removed from the SQL catalog.

Remark: The options **CASCADE** and **RESTRICT** in the **DROP TABLE** and **DROP VIEW** commands are not supported.

Index creation command

An SQL index is translated into an equivalent O_2 index or set of indexes.

Example 4.3.3 Let us consider the following index created on table **Proceedings** defined above:

```
CREATE INDEX i1 ON Proceedings( title );
```

The index creation command above is translated into the following O_2 index creation command in the O_2 base:

```
index SQL_Proceedings on title;
```

Table modification command

The **ALTER TABLE** command can be used to add columns to an existing table.

Example 4.3.4 Let us consider the following modification of table **Proceedings** defined above:

```
ALTER TABLE Proceedings ADD code INTEGER;
```

The command above is translated into the following O_2 class update command in the O_2 schema:

```
attribute code:integer in class SQL_Proceedings;
```

Table Types

We distinguish four different types of tables in a view schema:

- User Tables

  These are defined through the **SQL CREATE TABLE** command.

- View Tables

  These are defined through the **SQL CREATE VIEW** command.

- Class Tables

  These are generated by the export of an existing O_2 class through the **o2sql_export** tool.

- Collection Tables

  These are generated by the export of an O_2 collection attribute through the **o2sql_export** tool.

We assume that user tables belong to the SQL application and therefore all operations on them are allowed (deletion, index creation, modification), whereas class tables belong to O_2 , so that modifications to them are allowed only through the configuration file. The complete list of restrictions associated to each type of table is given below.

- SQL command **DROP TABLE**

  Only user and view tables can be dropped via the **DROP TABLE** command. Class and collection tables can be dropped indirectly via the **hide** command in the configuration file.

- SQL command **`ALTER TABLE`**

  Only user tables can be modified with the **`ALTER TABLE`** command. Although the syntax defined for the core level ODBC SQL does not allow constraints to be associated to a column added via **`ALTER TABLE`**, the constraints **`NOT NULL`** and **`DEFAULT`** are automatically associated to newly added columns. The default value is the corresponding O_2 default value for the attribute generated for the column. For instance, numeric columns have a zero default value, whereas character columns have the empty string as default value.

  Class and collection tables can be modified indirectly by modyfying the corresponding O_2 data types in the O_2 schema. For instance, adding an attribute to an exported O_2 class entails the addition of a new column to the corresponding table, unless the new attribute is hidden in the configuration file and one reruns **`o2sql_export`** to update the view scheme definition.

- SQL command **`CREATE INDEX`** and **`DROP INDEX`**

  Indexes can be created on user tables only. This is translated as the creation of an index on the corresponding system generated named collection. Indexes on collections used as class table extents (declared in the configuration file through the clause **`extent`**) can be defined directly in O_2 . Only indexes created through the SQL command **`CREATE INDEX`** can be dropped via **`DROP INDEX`**.

4.2.2 Data Update Commands

There are three types of update operations: row insertion, row deletion and row modification. With SQL, an update operation is issued on a table and is performed on a set of tuples (rows) which are selected through a query.

OQL does not dispose of a set of update commands analogous to those of SQL. In O_2 , updates to objects can be performed through application programs or by calling methods or functions from an OQL query.

In the current version, updates to user tables are performed by the SQL engine in a generic way, so that no extra O_2 C functions or methods need to be defined. User tables can thus be freely updated.

Class tables can be updated only if an update clause is declared for the corresponding class in the configuration file and a number of conditions hold. For instance, if a column in a table is derived from a method, then this column cannot be updated. Also, the ability to insert or remove rows in/from a table will depend on the nature of the corresponding table extent declared in the configuration file. If it is a named collection, insertions/deletions can be straightforwardly mapped into corresponding O_2 insertion/deletion operations, but if a class extent is given by a selection query, for instance, then insertions can be performed only through stored procedures, these procedures corresponding to user-defined O_2 C or C++ imported functions (see Section 4.2.5).

View tables cannot be updated and collection tables can be updated only indirectly through stored procedures defined to that end.

Insert commands

The insertion of tuples into tables is translated as the creation and initialization of the corresponding O_2 objects and the attachment of such objects to the root of persistence modeling the table extent in the O_2 base.

No restriction is imposed on the insertion of rows into user tables.

To be able to insert rows into a given class table through the `INSERT` command, however, an *insert clause* must be declared in the configuration file for the corresponding class.

Insertion from an associated named collection extent

The extent of class **`Article`** declared in the configuration file corresponds to a named collection defined in the original O_2 schema. The SQL insertion is automatically translated as an insertion of the newly created object into the corresponding O_2 class extent. The following insert clause must be declared in the configuration file to tell the system that insertions are allowed on table **`ARTICLE`**: .

```
export class Article
...
extent "Articles";
with insert;
end;
```

Let us consider the following **`INSERT SQL`** command:

```
INSERT INTO Article (title,date_title)
VALUES ('New Article','12/10/1995')
```

When the SQL command above is issued, the SQL engine inserts a newly created object into the corresponding user defined class extent after initializing the corresponding attributes. Object attributes are initialized with the column values given in the insertion command.

Insertion and foreign keys

When one inserts a row containing a foreign key value into a table, the corresponding row must exist in the referenced table, otherwise a referential integrity constraint is violated and the insertion is refused. If insertions are allowed in the referenced table, then the referred row must be inserted before the referring row is inserted. Finally, if direct insertions into the referenced table are not allowed, then insertions can be achieved indirectly, through a user-defined stored procedure.

In the example above, the inserted row contains the foreign key **date_title**, that points to an entry in table **Date**. If the corresponding date already exists in the Date table, the insertion of the article will be performed and the newly created **Article** object will point to the corresponding **Date** object. If the referenced date does not exist, the insertion will be refused by the update engine.

If, however, the user wants a new date with the corresponding key to be inserted whenever it does not already exist, the following O_2 C function can be defined and declared as a stored procedure in the configuration file to be called through the SQL interface. .

```
function body
Insert_Article(title:string,date_title:string):integer
{
  o2 Article obj = new Article;
  o2 Date obj_date;
  obj->sql_update_title(title);
  obj_date = select_Date(date_title);
  if (obj_date==nil) {
      obj_date= new Date(0,0,0);
      obj_date->to_date(tuple(mode:'a',
                              s_date:date_title));
  }
  obj->sql_update_date(obj_date);
  Articles += set(obj);
  return(0);
};
```

In the example above, we assume that the following method computes the logical key of an instance of class **Date**:

```
method body title:string in class Date {
  return(self->to_string(tuple(mode:'a')));
};
```

Assuming that the function **`Insert_Article`** is declared as a stored procedure, insertions into table **`Article`** can be performed through the SQL interface with the following SQL command:

```
CALL Insert_Article('A1','12/10/1995')
```

Insertion and computed extents

For class tables with an associated computed extent, the complex semantics of an insertion into such a table can be encapsulated into a user-defined function to be called by the user as a stored procedure.

In our example, an author depends, as a date, on an article to exist in the database, as it becomes persistent through the path leading from the root **`Articles`** to the attribute **`authors`**. But, contrary to a date, however, an author is not directly pointed to by an article. Instead, it is related to more articles through the collection table **`Article_authors`**. An entry in such table can be inserted only if the article and the author being related already exist, as they are pointed to by its foreign keys.

Remark: Class and collection tables can be updated through a stored procedure call even if an update clause is not declared for them in the configuration file.

Let us consider another example. The extent of table **`Author`** is given by a complex OQL query rather than by a named collection and insertions can therefore not be performed directly by the update engine into this table. Instead, in order to allow new authors to be inserted in the database, a function performing the insertion must be declared as a stored procedure in the configuration file.

When a row is inserted into table **`Author`**, the corresponding new object created must be attached to the attribute **`authors`** of a given **`Article`**. This implies that an article must be provided if one wants to insert a new author in the database.

Given the considerations above, the following function can be defined to be called as a stored procedure and perform the insertion of a row into table **`Author`**:

.

```
function body Insert_Author_of_Article(

   (name:string,
   address_institute_title:string,
   address_email:string,
   Article_title:string):integer {
  o2 Author obj = new Author;
  o2 Article a;
  obj->set_name(name);
  obj->set_address_institute(select_Institute(
                      address_institute_title));
  obj->set_address_email(address_email);
  a = select_Article(Article_title);
  if (a!=nil) {
    a->authors += set(obj);
    return 1;
  }
  return 0;
};
method body set_name(name:string) in class Author {
  self->name = name;
};
method body
set_address_institute(address_institute:Institute)
            in class Author {
  self->address.institute = address_institute;
};
method body set_address_email(email:string) in class
Author {
```

In the example, the **Insert_Author_of_Article** procedure performs the insertion of a new row into **Author** and of a new row into **Article_Author** that relates the article identified by **Article_title** to the newly inserted author.

Given the above, the insertion of a new author can be performed through the following SQL command:

```
CALL Insert_Author_of_Article("John Smith",
                              "ICS","smith@ics.fr","A1");
```

Many different stored procedures can be declared by the user for the different paths leading from a root of persistence to the instances of a given class.

Insertion with nested queries

If a query is specified in the body of an `INSERT` command, this query is translated to its equivalent OQL query, which must in turn return a set of tuples of atomic attributes corresponding to the attributes specified in the `column-identifier` list. The update engine iterates on the result of this nested queries to perform the insertion of the corresponding rows.

Delete commands

The deletion of tuples from a table must be translated as the disconnection of the corresponding O_2 objects from the root(s) of persistence to which they are attached in the O_2 base.

No restriction is imposed on the deletion of rows from user tables.

To be able to delete rows from a given class table through the **DELETE** command, however, a *delete clause* must be declared in the configuration file for the corresponding class.

Deletion from an associated named collection extent

The SQL deletion from table **ARTICLE** can be automatically translated as the removal of the corresponding object from the named collection extent. The following delete clause must be declared in the configuration file to tell the system that deletions from table **ARTICLE** are allowed:.

```
export class Article
   ...
   extent "Articles";
   with delete;
end;
```

Let us consider the following **DELETE** command:

DELETE FROM Article WHERE title = ‘Old Article’

The SQL engine will first select all objects corresponding to the rows to be deleted and then delete them from the class extent.

Deletion and foreign keys

Cascading deletions can be implemented through stored procedures, as for insertions.

In the example above, the deleted row contains the foreign key **date_title**, that points to an entry in table **Date**. Suppose that one wants the pointed date to be deleted from the corresponding table whenever a pointing article is deleted. As a row in table **Date** exists only as long as at least one row in table **Article** points to it, then the removal of the last pointing article from the database would automatically entail the removal of the pointed date. If however, the pointed objects can be reached through another path from a given persistence root, then the cascading deletion can be performed through stored procedures.

In addition, if the class associated to the pointed table has no extent clause associated to it, or if the associate extent expression is computed rather than given by a named collection, then the cascading deletion is performed by default.

Supposing that a named set **Dates** is defined in the O_2 schema and declared as the extent of class **Date**, then the deletion of an article would not entail the deletion of the pointed date. The cascading removal could be achieved by explicitly performing the removal of the pointed date from the named collection **Dates** in the function implementing a stored procedure used to remove articles from the database.

Deletion from a class table with an associated computed extent

Let us now consider the deletion of a row from the table **AUTHOR**. Such deletions cannot be automatically performed by the system and a delete clause should not be associated to the class **Author**. Instead, the following stored procedure can be used: .

```
function body
Delete_Author_of_Article(author_title:string,
                         Article_title:string):integer {
  select_Article(Article_title)->authors -=
                set(select_Author(author_title));
  return 0;
};
```

The **Delete_Author_of_Article** procedure performs the deletion of the **AUTHOR** row corresponding to the key passed as parameter and of the corresponding entry in table **Article_Author** that relates the article identified by **article_title** to the deleted author.

The deletion of a given author can be performed through the following SQL command:

```
CALL Delete_Author_of_Article("John Smith","A1");
```

As for insertion, many different stored procedures can be declared by the user for the different paths leading from a root of persistence to the instances of a given class.

Update commands

The update of tuples in a class table must be translated as the update of the corresponding O_2 objects in the O_2 base.

No restriction is imposed on the updateof rows from user tables.

To be able to update rows in a given class table, an *update clause* must be declared in the configuration file for the corresponding class, as illustrated below:.

```
 export class Article
   ...
   extent "Articles";
   with update;
end;
```

Let us consider the following **UPDATE** command:

UPDATE Articles SET title = 'New Article' WHERE title = 'Old Article'

The SQL engine will first select all objects corresponding to the rows to be updated and then update their attributes with the new corresponding column values.

Update and foreign keys

The update of foreign key columns is similar to the insertion of new rows with foreign key columns. When one updates a foreign key row column, the corresponding row must exist in the referenced table, otherwise a referential integrity constraint is violated and the update is refused. If insertions are allowed in the referenced table, then the referred row must have been inserted before the referring row is updated. If however, it is not possible to explicitly insert a row into the referenced table, then an insertion into this table can be achieved through a *cascading update* of the referencing table implemented by a stored procedure.

To conclude this section, we recall that the use of stored procedures can be generalized to overcome the limitations on update operations on class and collection tables.

4.2.3 Data Retrieval Commands

Data retrieval commands correspond to the **SELECT-FROM-WHERE** SQL queries.

The query translator builds up on the implementation of the OQL query interpreter. Starting from the OQL version 1.2 interpreter, the original OQL query parser was replaced by an SQL parser building an OQL query tree. The construction of an OQL syntax tree from a given SQL query is based on information collected by the schema translation and kept in the SQL catalog. The generated OQL tree is further optimized by applying standard optimization techniques.

Query translation is thus integrated to current OQL interpretation and is performed in two main steps:

1. The text of an SQL query is parsed and the abstract syntax tree of the equivalent OQL query is constructed, based on information collected by the schema translator.
1. The rewritten query (i.e. its corresponding syntax tree) is passed to the next phases of OQL query interpretation: query graph construction, optimization and evaluation.

As the syntax of the OQL language version 1.2 is very close to that of SQL, the OQL tree construction is straightforward. This first step produces the syntax tree of an OQL query that is already semantically equivalent to the original SQL query, i.e. this tree can be used by the OQL interpreter *as it is* in the subsequent phases of the standard OQL query processing, namely graph construction, optimization and evaluation, to produce the expected result *without no further intervention of the query translator*. In other words, the first step captures the semantics of the SQL query into an equivalent OQL query already and the second step is standard OQL engine activity.

During the construction of the tree, the translator searches well defined access patterns to perform query rewriting. When a given pattern is matched, an action on the corresponding subtree is performed. This action entails the replacement of one or more subtrees by other equivalent subtrees as well as the inclusion of new variables in the corresponding **from** clause. The intuition is that the new subtrees are semantically equivalent to the ones they replace, but the corresponding subqueries can be evaluated in a more efficient way.

The derivation of a query graph from the syntax tree is performed as for ordinary OQL queries. In particular, standard optimization techniques are applied as for ordinary OQL query graphs.

Remark: Only the percent character (matching zero or more of any character) is supported in pattern values used in the **LIKE** predicate and in parameters to some API functions (e.g. **szTableName** parameter of **SQLTables**). The underscore character (matching one character) is not supported.

Below, we give some examples of query translation to illustrate the process. The examples are based on the view schema given in Figure 4.2.

Example 4.3.5 Retrieve the name of all articles:

SQL query:	OQL query:
SELECT title	select struct(title:x0.title)
FROM Article	from Articles x0

Example 4.3.6 Retrieve the name of all articles using a column alias:

SQL query:	OQL query:
SELECT title AS article_name	select struct(article_name:x0.title)
FROM Article	from Articles x0

Example 4.3.7 Retrieve all columns of all rows of table Article:

SQL query:	OQL query:
SELECT *	select struct(title:x0.title,date_title:x0.date.title)
FROM Article	from Articles x0

Example 4.3.8 Retrieve the names of all authors:

SQL query:	OQL query:
SELECT name	select struct(name:x1.name)
FROM Author	from Articles x0,x0.authors x1

Example 4.3.9 Retrieve the names of all authors of an article whose title is “The Article”:

SQL query:	OQL query:

SELECT name
FROM Author
WHERE title IN
   (SELECT Author_title
   FROM Article_authors
   WHERE Article_title IN
      (SELECT title
      FROM Article
      WHERE title = "The Article"))
	select struct(name:x1.name)
	from Articles x0,x0.authors x1
	where x0.title == "The Article"

Example 4.3.10 Retrieve the name of all authors of article "Article 1":

SQL query:	OQL query:

SELECT y.name
FROM Article x,Author y,
     Article_authors z
WHERE x.title = "Article 1"
  AND x.title = z.Article_title
  AND y.title = z.authors_title
	select struct(name:x1.name)
	from Articles x0,
	x0.authors x1
	where x0.title == "Article 1" and
	      x1 in x0.authors

Example 4.3.11 Retrieve the electronic addresses of all authors:

SQL query:	OQL query:
SELECT address_email	select struct(address_email:x0.address.email)
FROM Author	from Articles x0,x0.authors x1

Example 4.3.12 Retrieve the institute_title of author "Author 1":

SQL query:	OQL query:
SELECT address_institute_title AS i struct(i:a.address.institute.title)	select
FROM Author	from Articles x0, x0.authors x1
WHERE name = "Author 1"	where x1.name == "Author 1"

Example 4.3.13 Retrieve the name of all authors having written at least two different articles:

SQL query:	OQL query:
SELECT name	select struct(name:x1.name)
FROM Author, Article_authors	from Articles x0, x0.authors x1
WHERE title = Authors_title AND	group by name
GROUP BY name	having count(partition) > 1
HAVING COUNT(*) > 1	

4.2.4 Granting privileges

Privileges are defined through the **GRANT** and **REVOKE** commands. A privilege definition is, together with table and view definition commands, a basic relational schema element.

When a grant command is passed as an ordinary SQL statement through the SQL interface, the system records information about granted update privileges. In the current version, however, update privileges are not checked by the system.

Remark: The keyword **USER** represents the constant string "**USER**" instead of the name of the current user.

4.2.5 Stored Procedures

Stored procedures are declared in the configuration file, as illustrated below:

```
stored procedure Insert_Article
      "This will insert a tuple into table Article and a tuple
      into table Date if the referenced date does not exist.",
            C++:Process_Complex_Update
      "This will perform something by calling a C++ imported
            function.";
```

For each procedure, a text may be associated to it, in addition to the procedure name. This allows a brief description of the semantics of the procedure to be stored in the SQL catalog and to be retrieved when the stored procedures are inspected through the o2sql_query tool or through the ODBC API function **SQLStoredProcedure**.

O2C procedures

By default, stored procedures correspond to an O_2 C function with the same name defined in the O_2 schema.

The following is a call to the procedure **Insert_Article** declared above:

CALL Insert_Article('A1','12/10/1995')

Such call is straightforwardly translated into the following OQL query:

Insert_Article('A1','12/10/1995')

C++ procedures

If the prefix "**C++:**" is added to the procedure name, then the procedure will correspond to an imported C++ member function of the imported class **SQLStoredProcedureHandler**.

C++ procedures allow functions defined by a C++ appplication to be called through the SQL interface instead of using O_2 C functions.

To be able to call C++ functions, the application must perform the following steps:

- to import a C++ class named **SQLStoredProcedureHandler** into the O_2 schema. This class should be defined by the user to group all functions that are to be called as stored procedures through the SQL interface.

- to define the root of persistence **`SQLStoredProcedureHandler`** in the O_2 schema with type **`SQLStoredProcedureHandler`**.

- to create an instance of class `SQLStoredProcedureHandler` and attach it to the root of persistence `SQLStoredProcedureHandler`.

  Let us consider a call to the procedure **`Process_Complex_Update`** (we assume that this procedure takes no parameter):

```
CALL Process_Complex_Update()
```

  Such call is translated into the following OQL query:

```
SQLStoredProcedureHandler->Process_Complex_Update()
```

  The standard **`o2odbc_server`** program (see command “o2odbc_server”, page 7-109) is able to automatically execute O_2C functions declared as stored procedures.

Linking C++ functions with the “sql” library

Stored procedures implemented by C++ functions cannot be execute through the **`o2sql_query`** shell nor through the **`o2odbc_server`** program. This is so because the library containing the implementation of such functions is not linked to **`o2sql_query`** nor to the **`o2odbc_server`**.

A C++ application wishing to call C++ functions as stored procedures through the SQL interface must then be linked at least with the **`sql`**, the **`oql`** and the **`o2cppruntime`** libraries (as well as other O_2 and general purpose libraries necessary to build the application). This is detailed in Chapter 6.

Typing restrictions

The following conditions must hold on the O_2 C and C++ imported functions declared as stored procedures:

- input arguments, if any, must have an atomic type;

- the result type, if the function returns a result, must be of one of the following:

- an atomic type;

  The procedure has an output parameter and must be called with the syntax **`? = call proc-name(arg1,..,argn)`**, i.e. the result can be retrieved as an output parameter.

- a collection of tuples of atomic type attributes;
  The procedure returns a result set as a **select-from-where** query.

- a class whose type is a collection of tuples of atomic type attributes;
  The procedure returns a result set, as in the previous case.

The conditions above are checked by the **o2sql_export** tool when the configuration file is loaded and an error is reported and the view generation abort if they do not hold on all declared procedures.

4.3 Development Tools

4.3.1 View creation tool o2sql_export

The **o2sql_export** tool allows view schemas to be created and modified. It takes two mandatory arguments: a schema name and a view name, through arguments **-schema** and **-view** respectively.

An optional configuration file can be provided through the argument **-config**. All classes in the input schema, if any, are exported into the view associated to this schema as relational tables, unless they are hidden in the configuration file.

If no configuration file is given, a default translation is performed (no hiding nor renaming of classes and/or attributes take place).

The complete usage of **o2sql_export** is given in Chapter 7.

A view schema generated with **o2sql_export** can be inspected at any time with the tool **o2sql_query** or through the ODBC API, by calling the appropriate catalog functions (e.g. **SQLTables**, **SQLColumns**, etc).

Remark: A view can be created on an empty O_2 schema. This schema can be further populated through **CREATE TABLE** commands.

Modifying existing views

Views generated with the **o2sql_export** tool can be further deleted and updated. Update is performed through the **o2sql_export** tool, i.e. running **o2sql_export** on an existing view allows the view to be modified. This will be usually performed to associate a new configuration file to an existing view (changing hidings, redefinitions, stored procedure declarations, etc). The tool prompts the user for confirmation of the view update.

The deletion of a view schema can be performed through the **o2sql_query** tool, as it will be described in the sequel.

The SQL catalog

O_2 keeps an SQL catalog as part of its system catalog. An entry in this SQL catalog is associated to each view schema created with the **o2sql_export** tool.

SQL user definitions such as view tables and integrity constraints associated to user tables are kept in internal structures of the SQL catalog. The information provided in the configuration file is also stored in the SQL catalog.

The SQL catalog is thus accessed when a view schema is created, updated or deleted and it is automatically updated when SQL operations updating the view schema (e.g. **CREATE TABLE**, **CREATE INDEX**) are performed on the database.

The SQL catalog can be inspected through a number of display functions, which are detailed below.

Each O_2 schema keeps a list of SQL catalog structures, one per view schema created on it. Entries in the SQL catalog are removed when the corresponding view schemas or the O_2 schema are deleted.

4.3.2 SQL shell tool o2sql_query

This tool allows views to be activated, deleted and inspected. It is an interactive shell allowing SQL commands and some special maintenance commands to be executed on an activated view.

There is no mandatory argument, but if a base and a view are provided as arguments, the view is activated on that base when the shell is launched. Otherwise, a view can be activated once the shell has been launched with the command **set view schema** that will be described below.

The main uses of the **o2sql_query** tool are:

- To quickly test some queries on the database before writing an O_2 / ODBC complete application. It actually uses the **o2_sql** function in order to evaluate the SQL queries submitted by the user through the standard input.

- The deletion of existing view schemas through the **delete view schema** command.

- The inspection of the SQL catalog through a number of display commands: **display config file**, **display view schema**, **display tables**, etc. This can be particularly useful for tunning up configuration files and the resulting view schemas so as to adapt them to the needs of a given application.

The complete usage of **o2sql_query** is given in Chapter 7.

If an output file is specified through argument **-output** then the result of SQL selection queries and of general view inspection commands is dumped into this file.

Once the shell is launched, the following prompt is displayed:

TYPE YOUR QUERY ENDED BY ';':

Different kinds of commands can be submitted to the shell. These are considered in turn.

SQL commands

Standard SQL commands using the syntax defined for the core ODBC SQL level in the appendix C of the ODBC SDK Programmer's Reference;

These are standard SQL commands which include "Data Retrieval Commands" on page 55, "Data Update Commands" on page 47, "Schema Update Commands" on page 42 and "Stored Procedures" on page 59.

Remark: SQL commands with input and/ or output parameters cannot be submitted to the **o2sql_query** shell.

Transaction commands

As far as transactions are concerned, the default behavior of the **o2sql_query** tool is similar to that of the o2shell tool. In other words, when the shell is launched, a transaction is implicitly started. At any moment the following transaction commands can be executed:

- **commit work**

  This will commit all updates to data and to the currently active view schema by performing a commit on O_2 .

- **rollback work**

This will perform an abort on O_2 and rollback all modification to data and to the currently active schema.

If the user quits the tool (by typing "`;`") without commiting or aborting, then a commit is implicitly performed.

Alternatively, the user can run the shell in an auto-commit mode (option `auto_commit`). In this case, a commit is automatically performed after each command is executed. When running in auto-commit mode, the transaction commands described above are not allowed.

View inspection commands

The following view inspection commands are available:

- **`display view schemas;`**

  This will list the names of the different views defined on the currently active base.

- **`display view schema;`**

  This will display all the definitions (tables, indexes and procedures) in the currently active view schema.

- **`display tables;`**

  This will display all tables in the currently active view schema. These are the user, class, collection and view tables.

- **`display table <table-name>;`**

  This will display the definition of the table **`<table-name>`** in the currently active view schema.

- **`display procedures;`**

  This will list the names of all stored procedures declared in the configuration file for the currently active view schema.

- **`display procedure <proc-name>;`**

  This will display the definition of the stored procedure **`<proc-name>`** in the currently active view schema.

- **`display indexes;`**

  This will list the names of all indexes created through the **`SQL CREATE INDEX`** command.

- **`display index <index-name>;`**

This will display the definition of the index **<index-name>** in the currently active view schema.

- **display config file;**

  This will display the contents of the config file used to derive the currently active view through the **o2sql_export** tool, if any. The configuration file used to derive a view is needed only at view creation time. If a change to the view needs to be performed by editing an existing configuration file, the contents of the file used to derive the view can be retrieved through the **display config file** command and dumped to a file to be edited. This releases users from keeping backups of configuration files on their disk.

View management commands

- view activation command;

  set view schema (<base-name>,<view-name>);

  This will activate the view schema **<view-name>** on base **<base-name>.**

- view deletion command;

  delete view schema (<schema-name>,<view-name>);

  This will delete the view schema **<view-name>** on defined on schema **<schema-name>**. The view must not be the currently active view.

4 O2 SQL

5 O_2 ODBC

This chapter describes how to use the O_2 ODBC driver and write or use ODBC application that access O_2 data sources.

Section 5.2 describes various data sources and details how an ODBC client application can connect to an O_2 data source. Section 5.3 gives information on the ODBC API functions implemented by the O_2 ODBC driver.

In order to read this chapter, it is assumed you are familiar with the ODBC environment and related concepts.

5 O2 ODBC

5.1 Conformance Levels

The O_2 ODBC driver has the following conformance levels:

- API Conformance Level: **`Level 1`**
- SQL Conformance Level: **`Core`**

Note: The O_2 ODBC driver also supports some of the functions in the level 2 API conformance level and part of the grammar in the extended SQL conformance level.

5.2 O_2 Data Sources

An O_2 data source is defined by:

- an O_2 system
- an O_2 database
- optionally, an ODMG C++ application to which the client must connect
- a query kind mode (currently only SQL is supported)
- an SQL view for SQL kind connections,
-

5.2.1 Connection to Data Sources

Connection to a data source is performed by **`SQLConnect`** or **`SQLDriverConnect`** calls.

When the function **`SQLConnect`** is used, information in the **`ODBC.ini`** file (or registry information) is used to perform the connection, whereas with **`SQLDriverConnect`**, a connection string (or prompted information) is used.

5.2.2 Configuring Data Sources with `ODBC.ini`

An O_2 data source specification section in the **`ODBC.ini`** file will introduce 4specific keywords : **`System`**, **`Database`**, **`Application`**, and **`View`**. Its format is given below: .

```
[data-source-name]
Driver=driver-DLL-path
System=system-name
Database=database-name
[Application=application-name]
View=SQL-view-name
```

5.2.3 Connection String

A connection string used by **`SQLDriverConnect`** and by the **`o2odbc_dump_base tool`** (Section 5.4.4) has the following syntax:

```
connection-string : := empty-string [" ; " ] | list-of-
attributes  [" ; " ]

list-of-attributes   : := attribute | attribute " ; "
list-of-attributes

attribute : := DRIVER "  = { "  attribute-value " } " |
attribute-keyword "  = "  attribute-value | specific-
attribute

attribute-keyword : := DSN | UID  | PWD

specific-attribute: :=  SYSTEM "  = "  attribute-value
| DATABASE "  = "   attribute-value | APPLICATION  "  =
"  attribute-value | VIEW  "  = "  attribute-value

attribute-value : := character-string
```

The DSN keyword is the only keyword necessary to connect to a data source from a Windows 95/ NT client, as information about the O_2 system, base and view are part of the data source definition.

When using the O_2 ODBC client library to connect to an O_2 ODBC server without passing through an ODBC Driver Manager, however, the connection string for the O_2 ODBC driver must use the keywords:

Keyword	Description
SYSTEM	The name of the O_2 system.
BASE	The name of the O_2 base.
APPLICATION	The name of a C++ application (optional).
VIEW	The name of the SQL view.

5.3 ODBC API Functions

All Core and Level 1 API ODBC functions are supported. Some functions in level 2 are also supported. The list of all functions implemented by the O_2 ODBC Driver can be retrieved with the **`SQLGetFunction`** ODB API function.

The main restrictions in the API concern the extended cursors (scrolls, updates), and positioned update or delete statements, which are not supported.

The Level 2 functions implemented by the O_2 ODBC driver are:

- **`SQLNumParams`**
- **`SQLNativeSql`**
- **`SQLExtendedFetch`**
- **`SQLForeignKeys`**
- **`SQLPrimaryKeys`**
- **`SQLProcedures`**

The functions implemented by the O_2 ODBC driver are grouped by type of task in the sequel. Specificities of the O_2 ODBC driver regarding some of these functions are given whenever necessary.

5.3.1 Connecting to a data source

- **SQLAllocConnect**

- **SQLAllocEnv**

- **SQLConnect**

- **SQLDriverConnect**

5.3.2 Obtaining information about a driver and a data source

- **SQLGetInfo**

  Appendix B gives the values returned by the **SQLGetInfo** ODBC API function for all possible **fInfoType** input argument values.

- **SQLGetTypeInfo**

5.3.3 Setting and retrieving driver options

- **SQLSetConnectOption**

  This function sets a connection statement option. No specific driver options have been defined. The connection options that can be set with this function are:

  —**SQL_AUTOCOMMIT**

  This option defines the transaction mode. To set this option value, the connection must not be opened, otherwise the driver returns **SQL_ERROR**.

  The two possible values for this option are:

  —**SQL_AUTOCOMMIT_ON**

  If the value is set to **SQL_AUTOCOMMIT_ON** (auto-commit mode), the driver commits each statement immediately after it has been executed. This is the default value (accordingly, an **o2odbc_server** is launched by default in auto-commit mode).

—SQL_AUTOCOMMIT_OFF

If the value is set to **SQL_AUTOCOMMIT_OFF** (manual-commit mode), it is up to the application to explicitly commit or roll back transactions with **SQLTransact**.

Remark: An O_2 ODBC server running on manual mode must be declared to the dispatcher to allow the connection to the data source to be performed when option **SQL_AUTOCOMMIT** is set to **SQL_AUTOCOMMIT_OFF**.

—SQL_ACCESS_MODE

This option defines the access mode:

—SQL_MODE_READ_WRITE

This is the default mode.

—SQL_MODE_READ_ONLY

This value is supported but not used in this current driver version.

—SQL_TXN_ISOLATION

Sets the transaction isolation level. If a transaction is open, the driver returns **SQL_ERROR**.

SQL_TXN_SERIALIZABLE (serializable transactions plus locking) is the default and the only valid option value in the current O_2 ODBC driver version.

—SQL_ODBC_CURSORS

This is relative to **SQLExtendedFetch** calls. To set this option value, the connection must not be opened, otherwise the driver returns **SQL_ERROR**.

The value **SQL_CUR_USE_ODBC** means that the driver manager will use the Microsoft ODBC cursor library for cursor scrolling. Currently, this is the only valid option value. The application must set this option to **SQL_CUR_USE_ODBC** if it wants to use **SQLExtendedFetch**.

The following options are not supported:

—SQL_PACKET_SIZE

—SQL_QUIET_MODE

—SQL_CURRENT_QUALIFIER

—SQL_OPT_TRACE

—SQL_OPT_TRACEFILE

• **SQLGetConnectOption**

—SQLSetStmtOption

Sets a statement option value. The statement options that can be set with this function are:

—SQL_ASYNC_ENABLE

The two possible values are:

—SQL_ASYNC_ENABLE_ON

The following functions can be executed asynchronously : **SQLGetTypeInfo**, **SQLPutData**, **SQLParamData**, **SQLExecDirect**, **SQLPrepare**, **SQLExecute**, **SQLFetch**, **SQLGetData**, **SQLNumResultCols**, **SQLDescribeCol**, **SQLColAttributes** and all catalog functions.

—SQL_ASYNC_ENABLE_OFF

Disable asynchronous function executions.

Changing this option value is allowed at any time, because no asynchronous functions can be still executing for this statement. This induces immediate effect for subsequent calls. The default value is **SQL_ASYNC_ENABLE_OFF**.

—SQL_NOSCAN

Scanning or not SQL string for escape clauses. Escape clauses are allowed only in SQL statement strings for extended ODBC procedure calls. The two possible values are:

—SQL_NOSCAN_OFF

The driver will scan SQL strings for escape clause.

—SQL_NOSCAN_ON

The driver does not scan and sends directly the statement to the data source.

Changing this value will takes effect for the next calls to **SQLExecDirect** or SQLPrepare. The default value is **SQL_NOSCAN_OFF**.

—SQL_MAX_LENGTH

This gives the maximum amount of data returned by the driver for a character or binary column. If the value is 0, the driver attempts to return all available data. For any other value greater than 254 bytes, if the length of available data is greater than **SQL_MAX_LENGTH**, data retrieved with **SQLFetch** or **SQLGetData** are truncated without error or warning messages.

In the current version, the only valid value is the default one, i.e. 0, meaning all available data is retrieved whenever possible.

—SQL_QUERY_TIMEOUT

Number of seconds to wait for an SQL statement to execute before returning to the application. If the value is 0, the time-out is disabled (no time out). If the value exceeds the maximum time-out in the data source, 600 seconds, or is smaller than the minimum, 60 seconds, the driver substitutes that value by this maximum or minimum value and returns **SQL_SUCCESS_WITH_INFO**.

Changing the value is allowed any time and is taken into account for subsequent executions. The default value is 0 (no time out).

—SQL_ROWSET_SIZE

Defines the number of rows returned by an **SQLExtendedFetch**. Any value is supported. Changing this value is allowed even if cursors are opened, specially between two **SQLExtendedFetch**. The value will take effect for the next **SQLExtendedFetch** calls. The default value is 1.

—SQL_MAX_ROWS

This defines the maximum number of rows to return to the application for a **SELECT** statement. If the value is 0, the driver returns all rows. Any another value is allowed. The default value is 0 meaning all rows.

—SQL_BIND_TYPE

Two types of value define the bind type to be used by **SQLExtendedFetch**. The default and only possible value is **SQL_BIND_BY_COLUMN**.

—SQL_RETRIEVE_DATA

Two values for retrieving data in **SQLExtendedFetch** calls:

—SQL_RD_ON

In **SQLExtendedFetch** calls, data are retrieved.

—SQL_RD_OFF

SQLExtendedFetch positions the cursor to the specified location but data are not retrieved. For example, this option value allows an application to call **SQLExtendedFetch** only to verify existence of rows and check global errors.

Changing this value is allowed even if cursors are opened, especially between two calls to **SQLExtendedFetch**. The new value takes effect for the next **SQLExtendedFetch** calls. The default value is **SQL_RD_ON**.

—SQL_CONCURRENCY

Specifies the cursor concurrency. To set this value, the cursor must not be opened and the statement not prepared. The default and only value supported by the O_2 ODBC driver is **SQL_CONCUR_READ_ONLY**, meaning that the cursor is read-only and no updates are allowed. If another value is specified, the driver substitutes this value by the default one and returns **SQL_SUCCESS_WITH_INFO**.

—SQL_CURSOR_TYPE

Specifies the cursor type. To set this value the cursor must not be opened and the statement not prepared. The default and only value supported by the O_2 ODBC driver is **SQL_CURSOR_FORWARD_ONLY**, meaning that the cursor only scrolls forward. If an other value is specified, the driver substitutes this value by the default one and returns **SQL_SUCCESS_WITH_INFO**.

The following options are not supported:

—SQL_KEYSET_SIZE

—SQL_SIMULATE_CURSOR

—SQL_USE_BOOKMARKS.

- **SQLGetStmtOption**

Besides the options used with **SQLSetStmtOption**, for which the driver returns the current setting, another option can be retrieved:

—SQL_ROW_NUMBER

This allows the number of the current row in the result set to be retrieved. If the current row cannot be determined or if there is no current row, the driver returns 0. To get this option value, a cursor must be opened and not positioned before or after the result set.

5.3.4 Preparing SQL requests

- **SQLAllocStmt**

- **SQLNativeSql**

- **`SQLPrepare`**
- **`SQLBindParameter`**
- **`SQLGetCursorName`**

  Cursor names are used by positioned update or delete statements. Even if those statements are not supported by the O_2 ODBC driver, the functions **`SQLSetCursorName`** and **`SQLGetCursorName`** are implemented.

- **`SQLSetCursorName`**

5.3.5 Submitting requests

- **`SQLExecute`**
- **`SQLExecDirect`**
- **`SQLNumParams`**
- **`SQLParamData`**
- **`SQLPutData`**

5.3.6 Retrieving results and information about results

- **`SQLRowCount`**
- **`SQLNumResultCols`**
- **`SQLDescribeCol`**
- **`SQLColAttributes`**
- **`SQLBindCol`**
- **`SQLFetch`**

• **SQLExtendedFetch**

• **SQLGetData**

5.3.7 Catalog functions

The following restrictions apply to catalog functions:

- result sets are not ordered (e.g. by table name for **SQLTable**);
- only the percent character (matching zero or more of any character) is supported in search patters;
- table qualifiers and owners are not supported.

• **SQLColumns**

• **SQLForeignKeys**

• **SQLPrimaryKeys**

• **SQLProcedures**

Returns the list of procedure names and characteristics for a specific data source. These are the procedures declared in the configuration file used to derive the view associated to the data source.

• **SQLSpecialColumns**

• **SQLStatistics**

Only statistics giving the number of rows of a table will be performed. For indexes information, no data will be returned in the result set.

If the argument **fAccuracy** is **SQL_ENSURE**, the number of rows in the table is unconditionally retrieved which means that a **COUNT** request is performed on the table in the O2 data source. If **fAccuracy** is **SQL_QUICK**, this number is only retrieved if it is readily available from the server.

• **SQLTables**

If the argument **szTableType** is **%** and the argument **szTableName** is an empty string, the result set contains the list of valid table types for the data source (all others columns contain **NULL**). Valid table types are: **O2 CLASS TABLE**, **O2 COLLECTION TABLE**, **USER TABLE**, **VIEW TABLE**. For more details on the different types of tables, see Chapter 4, "Schema Update Commands" on page 42.

If a qualifier or owner is specified, **SQL_ERROR** is returned.

5.3.8 Terminating a statement

- **SQLFreeStmt**
- **SQLCancel**
- **SQLTransact**

If the connection is in auto-commit mode, an O_2 transaction is started each time an SQL statement that can be contained within a transaction is executed against the current data source. The driver validates this transaction after each execution.

Executing a **SELECT** statement will imply, for the O_2 data source, starting an O_2 transaction, processing, opening a scan on the result and validating the transaction. For the O_2 ODBC client, a cursor is opened.

An **SQLFreeStmt** with **SQL_CLOSE** option value will close, for the O_2 data source, the scan and delete pending results, and, for the O_2 ODBC client, close the cursor and delete pending results.

In manual-commit mode, each time an SQL statement that can be contained within a transaction is submitted to the O_2 data source an O_2 transaction is started only if no transaction is already open. All statements associated to a connection share the same transaction space. In order to commit or rollback a transaction, the application must call **SQLTransact** with the appropriate parameter.

Executing a **SELECT** statement will imply, for the O_2 data source, processing and opening a scan on the result, and, for the O_2 ODBC client, opening a cursor. An **SQLFreeStmt** with **SQL_CLOSE** option value will, for the O_2 data source, close the scan and delete pending results, and, for the O_2 ODBC client, close the cursor and delete pending results.

When **SQLTransact** is called, with the only valid option **SQL_CB_DELETE**, it commits or rollbacks all the previously submitted requests within the transaction. For the O_2 data source, all opened scans are closed, all pending results and all access plans (i.e. O_2 handles) are deleted. For the O_2 ODBC client, cursors and pending results are deleted for all the associated statements.

5.3.9 Terminating a connection

- **SQLDisconnect**
- **SQLFreeConnect**
- **SQLFreeEnv**

5.3.10 General information

- **SQLError**
- **SQLGetFunctions**

The argument **fFunction** is **SQL_API_ALL_FUNCTIONS** or a defined value identifying the ODBC function of interest. The output argument **pfExists** is an array of 100 elements or a single **UWORD**. Values are set to **TRUE** if the function is supported, **FALSE** otherwise.

SQLGetFunctions will return **FALSE** for the following level 2 functions only: **SQLBrowseConnect**, **SQLParamOptions**, **SQLSetPos**, **SQLSetScrollOptions**, **SQLDescribeParam**, **SQLMoreResults**, **SQLProcedureColumns**, **SQLColumnPrivilIges** and **SQLTablePrivileges**.

5.4 O_2 ODBC Tools

A number of tools is available for O_2 ODBC developers. These are programs that should be found in the **bin/<platform>** subdirectory of the O_2 installation directory.

5.4.1 o2sql_export

As described in Section 5.3, an O_2 data source corresponds to an O_2 base on which a view has been activated. To be able to connect to a data source, an O_2 base must exist and a view on the schema of that base must have been previously created.

The **o2sql_export** tool is the view creation tool. Its features and complete usage are described in Section 4.3.1 and Section 5.4.1 respectively.

5.4.2 o2sql_query

The **o2sql_query** is an auxilairy tool used for view schema management. It can be very useful for virtual schema designers as it allows quick inspection of virtual schemas and databases. In particular, it can be used to delete view schemas from the SQL catalog and to retrieve the contents of a configuration file used to derive a given view into a file. This file can be thus modified and the view re-generated.

Its features and complete usage are described in Section 4.3.2 and command “o2sql_query”, page 7-115.

5.4.3 o2odbc_server

An O_2 ODBC servers process O_2 ODBC client requests.

When started, **o2odbc_server** establishes a connection with an O_2 OpenDispatcher (**o2open_dispatcher**) which must already be running and establishes also a connection with a named O_2 database system through an **o2server**, which must also already be running.

An O_2 ODBC server loads view information from the SQL catalog stored in an O_2 system for a given data source so as to be able to perform query translations. It also updates the SQL catalog whenever schema update commands (table, view and index creation, modification and destruction) are executed on the data source. Finally, it performs all the ODBC specific activity (data conversions, cursor management, etc) involved in the processing of clients requests.

An O_2 ODBC server can run in two modes, namely the auto-commit and manual modes. In the auto-commit mode, an implicit commit is performed after the execution of each SQL statement. In manual, mode, commits and/ or rollbacks must be explicitly performed by the application through the ODBC API function **SQLTransact**.

The complete usage of the **o2odbc_server** program is given in command "o2odbc_server", page 7-109.

5.4.4 o2odbc_dump_base

The whole contents of an O_2 data source, i.e. of the virtual relational database corresponding to the application of a view on an O_2 base, can be logically dumped into an ASCII file with the **o2odbc_dump_base** program. The logical dump of a virtual database consists of all table creation and row insertion SQL commands. The generated ASCII file can be given as input to a program that sends each command to execution on a given database. This allows the contents of a dumped database to be loaded elsewhere.

In particular, the generated output file can be given as input to the **o2sql_query** tool to duplicate the contents of a virtual database into another base. This allows an O_2 base (or the part of an O_2 base that is exported as a virtual database) to be materialized as a relational database.

The complete usage of the **o2odbc_dump_base** program is given in command "o2odbc_dump_base", page 7-108.

5.4.5 o2open_dispatcher

An O_2 OpenDispatcher registers all O_2 ODBC servers running on a LAN and is queried to get the address of a server able to answer to an O_2 ODBC client requests.

A server is chosen by the dispatcher according to a heuristics and based on connection options set by the client. A score is computed for each server running and the server with the best score is returned to the client.

The complete usage of the **o2open_dispatcher** program, including more details on the heuristics used by the dispatcher to choose a server for a given client, is given in command "o2open_dispatcher", page 7-111.

5 O2 ODBC

6 Programming an O_2ODBC Server

Programmers can use the **o2_Odbc** class to build their own O_2 ODBC servers.

This chapter describes how to integrate your C++ application with an O_2ODBC server so as to be able to access instances of the imported C++ classes stored in an O_2 database as relational data and, in particular, to execute C++ functions as stored procedures through the ODBC interface.

We assume the reader is familiar with the ODMG C++ Binding and with the concept of stored procedure in SQL and ODBC.

To implement your own O_2 ODBC server you build an ODMG C++ application using the following:

- user classes
- ODMG C++ libraries
- O_2 ODBC libraries

The following sections detail the different steps involved in the construction of an O_2 ODBC server.

6 Programming an O2ODBC Server

6.1 Defining the O_2 ODBC Server main function

You must build an O_2 ODBC server executable from a main function and application files. The main function uses the o2_Odbc class.

The general structure of a **main** function used in the construction of an O_2 ODBC server is the following:

- Creates an **o2_Odbc** class object.
- Sets the the server options and parameters.
- Initializes the O_2ODBC server (**begin**).
- Starts the server loop (**loop**).
- Finishes (**end**).

An example of a **main** function is given below. .

```
int main(int argc, char** argv)
{
        short  error=0;
        o2_Odbc *o2odbcServer = new o2_Odbc();
        o2odbcServer->set_sysdir(getenv("O2HOME"));
        o2odbcServer->set_conffile(".o2rc");
        o2odbcServer->set_confvar("O2OPTIONS");
        o2odbcServer->set_enroll(enroll_func);
        o2odbcServer->set_check(check_func);
        error = o2odbcServer->begin(argc, argv);
        if (error) {
                return(1);
        }
        error = o2odbcServer->init();
        if (error) {
                return(1);
        }
        o2odbcServer->loop();
        o2odbcServer->end();
        delete o2odbcServer;
        return (0);
}
```

Once the server is started, it connects to an O_2 server first (function begin) and then to an O_2 OpenDispatcher (function init). It then waits for requests sent by O_2 ODBC clients (function `loop`). Functions `begin`, `init`, `loop` and `end` of class `o2_Odbc` are defined in the o2odbc_svr library.

For a given application, a specific configuration can be defined in the `main` function by applying the appropriate `set` functions to the instance of `o2_Odbc`.

The full referential information on the `o2_Odbc` class is given in Section 7.1.

6.2 Compiling your own O_2 ODBC server

An O_2 ODBC server is built as an ODMG C++ application with the help of the o2makegen tool. A configuration file is used to build the makefile used in the construction of an O_2 ODBC server. The example below illustrates such a configuration file..

```
O2Home= $O2HOME
O2System= $O2SYSTEM
O2Server= $O2SERVER
O2Schema= o2odbc_cpp

+UseOql
+UseConfirmClasses

ImpFiles= Person.hxx SQLStoredProcedureHandler.hxx
[Person.hxx]ImpClasses= Person
[SQLStoredProcedureHandler.hxx]ImpClasses=
SQLStoredProcedureHandler
+[SQLStoredProcedureHandler.hxx]
     [SQLStoredProcedureHandler]
ImpAllPublicMemberFunc
ImpSet= Person
ProgramLibDir= $O2HOME/lib
ProgramLib= o2odbc_svr sql oql o2cppruntime o2runtime
o2api o2util
o2store o2common

Sources= Person.cc SQLStoredProcedureHandler.cc main.cc
ProgramObjs= main.o Person.o
SQLStoredProcedureHandler.o $O2HOME/obj/o2odbc_load.o
ProgramName= my_o2odbc_server
```

In the example aboce, `Person` is an application class defined in file `Person.hxx`. Its member functions are defined in file **`Person.cc.`** Class `SQLStoredProcedureHandler` groups all C++ functions that are declared as stored procedures in the configuration file used by the **`o2sql_export`** tool to create the relational view of the O_2 schema. The implementation of such functions is provided in file **`SQLStoredProcedureHandler.cc`**. The **`main.cc`** file contains the definition of the `main` function, as illustrated above.

The executable **`my_o2odbc_server`** (clause **`ProgramName`**) is generated by importing the **`Perso`**n and **`SQLStoredProcedureHandler`** classes into O_2 , compiling the corresponding source files and linking the respective object files with the libraries declared in the **`ProgramLib`** clause.

.

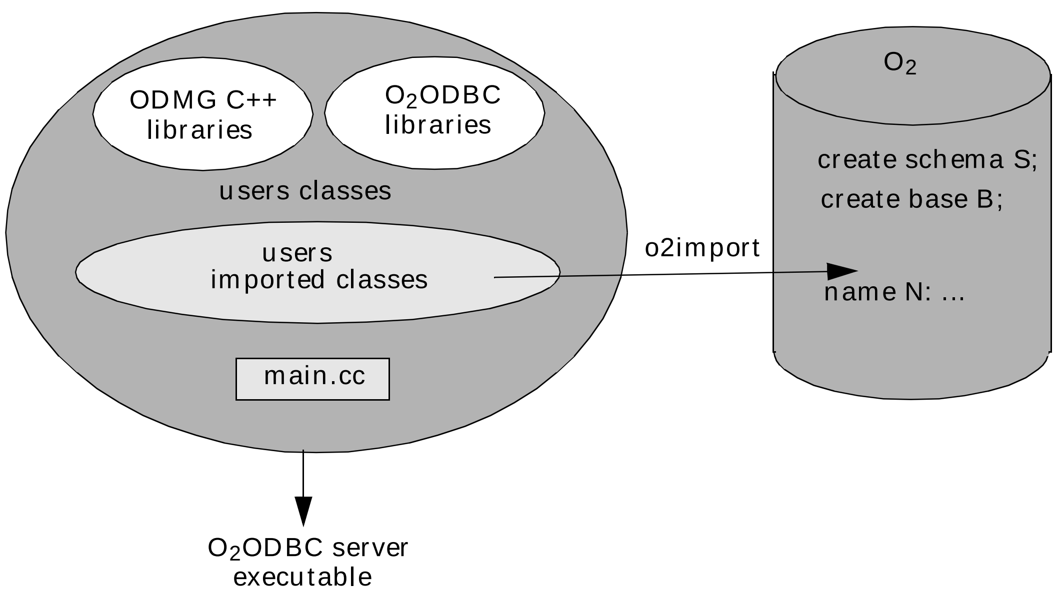


Figure 4.2: Components of an O_2ODBC application

For more details, refer to the ODMG C++ and **`o2makegen`** user and reference manuals.

6.3 Running your own O_2 ODBC server

Given a C++ application, the following steps should be performed to run an O_2 ODBC server that can access instances of C++ application classes through SQL and launch C++ functions as stored procedures:

- Initialize an O_2 system and run the O_2 server.

  This is achieved through the appropriate o2dba_init and o2server programs. For more information refer to the O_2 System Administration Manuals.

- Create an O_2 schema.

  This can be achieved through the appropriate administration tools (e.g. o2dsa). Refer to the O_2 System Administration Manuals.

- Import the C++ classes into O_2 .

  After schema creation, you must import the classes and member functions of your application. This is achieved through the appropriate tools (e.g. `o2makegen`). Refer to the ODMG C++ Binding Reference Manual and User's Guide.

- Import the class `SQLStoredProcedureHandler`.

  This class is the entry point allowing C++ functions to be called as stored procedures through the SQL interface.

- Create persistent roots.

  Persistent roots must be defined to store instance of the C++ application classes. Such roots can be used as an entry point in the database by the SQL engine, if they are declared as extents in the configuration file used by the `o2sql_export` tool.

- Create the O_2 ODBC server.

  As detailed above, the creation of an O_2 ODBC server involves the definition of a main function that, together with the application files, is used to build an executable that is linked to the appropriate libraries.

- Populate the database.

  An application must load data in the database before querying it. The database can be populated by the C++ application or through the SQL interface, with the appropriate row insertion SQL command or by calling user defined C++ functions declared as stored procedures.

- Run the server.

6

Programming an O2ODBC Server

A user-defined O_2 ODBC server works as a standard O_2 ODBC server, but as it is linked to some user-defined classes, it is able to run C++ functions defined in class `SQLStoredProcedureHandler` and declared in a view generation configuration file as stored procedures.

7 O_2 ODBC Reference

This chapter details the **o2_0dbc** class and all O_2 ODBC commands. It is divided into the following sections:

- The o2_odbc Class.

  This class is used by an application to start an O_2ODBC server and begin the server loop.

- The O2 ODBC Commands.

  This section provides the O_2ODBC system commands.

7 O2 ODBC Reference

7.1 The o2_odbc Class

This section presents the **o2_Odbc** class

and describes the following member functions:

- **banner**
- **begin**
- **end**
- **enroll**
- **enroll_path**
- **get_option**
- **init**
- **set...**
- **usage**

.

```
class o2_Odbc {
public:
  enum OptionType {
          NoValue, OptionalValue, MandatoryValue };
  enum OptionMode{
          Append=0,
      // string value are appended to old ones
          Replace=1,
      // string value replace old one
          Add=2
      // string value are added in the list found
  };
  o2_Odbc();
  ~o2_Odbc();

  static void interruptFunc(int signal);
  int begin(int argc, register char *argv[]);
int begin(int argc, register char *argv[],
      const char *sysdir, const char
      *systemname, const char *servername,
      const char *dispatchername, int verbose);
  int  begin(int argc, register char *argv[],
      const char *sysdir, const char *systemname,
      const char *servername, const char
      *dispatchername, const char *conf_file,
      const char *conf_var, void (*enroll_func)(),
      void (*check_func)(), int verbose);
  int  begin(int argc, register char *argv[],
      const char *sysdir, const char *systemname,
      const char *servername, const char
      *dispatchername, const char *conf_file,
      const char *conf_var, void (*enroll_func)(),
      void (*check_func)(), const char *swapdir,
      char * const *libpath, char * const *libname,
      int commitfrequency, const char * commitmode,
      const char *application, int verbose);
```

.

```
int init();
 int end();
 int loop();

 void set_systemname(const char *systemname);
 void set_servername(const char *servername);
 void set_sysdir(const char *sysdir);
 void set_swapdir(const char *swapdir);
 void set_dispatchername(const char *dispatchername);
void set_commitFrequency(const char *commitfrequency);
 void set_commitFrequency(int commitfrequency);
 void set_commitMode(const char *commitmode);
 void set_verbose(int verbose);
 void set_libpath(char * const *libpath);
 void set_libname(char * const *libname);
 void set_application(const char *appli);
 void set_conffile(const char *conf_file);
 void set_confvar(const char *conf_var);
 void set_enroll(void (*enroll_function)());
 void set_check(void (*check_function)());
 void set_default_env();
 static void default_enroll_func();
 static void default_check_func();
 static int usage();
 static int banner();
 static int enroll(const char * const name,
                  const char * const confname,
                  const char * const optname,
                  char *dflt,
                  const OptionType t,
                  const char * const desc,
                  const OptionMode mode=Replace);
```

.

```
 static int enroll(const char * const name,
                   const char * const confname,
                   const char * const optname,
                   long dflt,
                   const OptionType t,
                   const char * const desc,
                   const OptionMode mode=Replace);

  static int enroll(const char * const name,
                   const char * const confname,
                   const char * const optname,
                   char dflt,
                   const OptionType t,
                   const char * const desc,
                   const OptionMode mode=Replace);

  static int enroll(const char * const name,
                   const char * const confname,
                   const char * const optname,
                   double dflt,
                   const OptionType t,
                   const char * const desc,
                   const OptionMode mode=Replace);

  static int enroll_path(const char *path);
  static int get_option(const char *name,
                       char *&value,
                       int ind = -1);
  static int get_option(const char *name,
                       long &value,
                       int ind = -1);
  static int get_option(const char *name,
                       double &value,
                       int ind = -1);
  static int get_option(const char *name,
                       char &value,
                       int ind = -1);
};
```

7 O2 ODBC Reference

banner

Summary Displays the version number of O_2.

Syntax **`static int o2_odbc::banner();`**

Arguments None.

Description Displays the version number of O_2 on the standard output.

Returns 0 if successful.
-1 if there was an internal error.

begin

Summary Starts up a connection to an O_2 database.

Syntax

```
int begin (    int argc, register char * argv[]);

int begin (    int argc, register char * argv[],
               const char *systemname,
               const char *servername,
               const char *sysdir,
               int verbose);

int begin (    int argc, register char * argv[],
               const char *conf_file,
               const char *conf_var,
               void (*enroll_func) (),
               void (*check_func) (),
               const char *systemname,
               const char *servername,
               const char *sysdir,
               int verbose);

int begin (    int argc, register char * argv[],
               const char *conf_file,
               const char *conf_var,
               void (*enroll_func) (),
               void (*check_func) (),
               const char *systemname,
               const char *servername,
               const char *sysdir,
               const char *swapdir,
               const * const *libpath,
               const * const *libname,
               int verbose);
```

Arguments

argc Number of arguments of the C++ executable.

argv List of arguments of the C++ executable.

systemname Name of database system. This information is mandatory. It can be given as a parameter or by calling **o2_odbc::set_systemname** before beginning the session.

It can also be set by **o2_odbc::set_default_env**, in which case it is found in the parameter **-system** of your executable, in the **O2OPTIONS** environment variable (see the **conf_var** argument), or in the O_2 option file **.o2rc** (see the **conf_file** argument). See the *O_2 System Administration Guide* for further details.

servername Name of machine on which the O_2 server is running. It can be given as a parameter or by calling **o2_odbc::set_servername** before beginning the session.

It can also be set by **o2_odbc::set_default_env**, in which case it is found in the parameter **-server** of your executable, in the **O2OPTIONS** environment variable (see the **conf_var** argument), or in the O_2 option file **.o2rc** (see the **conf_file** argument). See the *O_2 Administration Guide* for further details.

sysdir Path to the directory where O_2 is installed. This information is mandatory. It not given, the value found in the environment variable O2HOME is used.

swapdir Path to a directory where a swap file can be created if O_2 needs it. It can be NULL, in which case the swap directory in the O_2 directory is used (See the *O_2 System Administration Guide*).

libpath A NULL-terminated array of character strings, where each string gives a directory path. O_2 searches these directories for libraries named in **libname** if dynamic linking is needed. It may be NULL.

libname A NULL-terminated array of character strings, each specifying a library name to use when linking and loading functions dynamically. It may be NULL.

conf_file Name of the file where the O_2 options manager can find the value for the enrolled options (see the **enroll** and **enroll_path** functions). If 0, **conf_file** takes the default value **.o2rc**.

conf_var Name of the environment variable where the O_2 options manager can find the value for the enrolled options (see the **enroll** and **enroll_path** functions). If 0, **conf_file** takes the default value **O2OPTIONS**.

enroll_func Pointer to a C function of type **static void (*func) ()**. This function must contain code for registering options.

check_func Pointer to a C function of type **static void (*func) ()**. This function must contain code for retrieving and verifying option values.

verbose An integer specifying the session as a verbose session.

Description

Starts up the connection to the database after analyzing the options.

This member function allows you to use the same powerful option mechanism that is used by all the tools of the O_2 environment. This option mechanism is explained in detail in the *O_2 System Administration Guide*.

The O_2 options mechanism allows you to define options from the following sources:

- Configuration file.
- Environment variables.
- Command line.

For a given option, a value retrieved from a configuration file can be overloaded by a value defined as an environment variable, which in turn can be overloaded by a value defined at the command line.

Using the O_2 options mechanism has the following advantages:

- Simple management of runtime options.
- A coherent set of options for all O_2 applications and tools.

Using the O_2 options mechanism is not mandatory. The most simple way to use the O_2 options mechanism is to use the member function o2_odbc::set_default_env before calling o2_odbc::begin.

```
void o2_odbc::set_default_env()
```

This function allows your C++ program to use the standard O_2 configuration file (**`.o2rc`**), the standard O_2 environment variable options (**`O2OPTIONS`**), and the standard O_2 command options:

-system, which defines the O_2 system name,
-server, which defines the name of the O_2 server host,
-help, which displays the help text for the program, and
-verbose, which enables the verbose mode.

Customizing the options

You can add your own options. For example, you can retrieve O_2C parameters using new options. To do this, you must use the **`o2_odbc::enroll`** and **`o2_odbc::get_option`** member functions.

The **`o2_odbc::enroll`** function allows you to register the options, and the **`o2_odbc::get_option`** function allows you to retrieve the value of the options.

You must write the following two functions:

- A register function that contains a call to **o2_odbc::enroll** functions, which register each of your options.
- A check function that contains a call to **o2_odbc::get_option** functions, which retrieve the value of the registered options.

These two functions can be registered using the **o2_odbc:begin** member function (**enroll_function** and **check_function** parameters) or explicitly, before calling **o2_odbc::begin**, using the following member functions:

```
void o2_odbc::set_enroll(void (*enroll_function) ())
void o2_odbc::set_check(void (*check_function) ())
```

The options for the system name and the server name are mandatory. These two options are registered by the following code, which you must add to your register function:

```
session->enroll( "system_name", "system", "system"
                 (char *)NULL,
                 MandatoryValue,
                 "o2 system name to connect to",
                 Replace);

session->enroll( "system_name", "server", "server"
                 (char *)NULL,
                 MandatoryValue,
                 "machine on which o2 server is running",
                 Replace);
```

After registering these mandatory options, you can register your own options.

Returns 0 if the connection was carried out successfully. If not, an error code is given.

end

Summary Ends an O_2 session.

Syntax **`int o2_odbc::end();`**

Arguments None.

Description Ends an O_2 session and the connection to the O_2 server. A commit is carried out automatically.

Returns Zero if successful, a non-zero value otherwise.

7 O2 ODBC Reference

enroll

Summary Registers an option to be recognized by the O_2 options manager.

Syntax

```
static int o2_odbc::enroll (const char * const name,
                            const char * const confname,
                            const char * const optname,
                            char *dflt,
                            const OptionType t,
                            const char * const desc,
                            const OptionMode mode=Replace);
```

```
static int o2_odbc::enroll (const char * const name,
                            const char * const confname,
                            const char * const optname,
                            long dflt,
                            const OptionType t,
                            const char * const desc,
                            const OptionMode mode=Replace);
```

```
static int o2_odbc::enroll (const char * const name,
                            const char * const confname,
                            const char * const optname,
                            char dflt,
                            const OptionType t,
                            const char * const desc,
                            const OptionMode mode=Replace);
```

```
static int o2_odbc::enroll (const char * const name,
                            const char * const confname,
                            const char * const optname,
                            double dflt,
                            const OptionType t,
                            const char * const desc,
                            const OptionMode mode=Replace);
```

Arguments

name A string that indicates the name of the option. This name is used for retrieving the value of the option.

confname A string that indicates under which name the value of this option can be given in a configuration file.

optname A string that indicates under which name the value of this option can be given in the environment variable or at the command line.

dflt The default value of the option. This value is retrieved if the end user does not give a value is given to the option.

t A value taken from the OptionType enumeration:

NoValue.	The option represents a boolean value. If there are values there will be an error during parsing of the options.
OptionalValue	The option can have an associated value.
MandatoryValue	The option represents a value. If this value is not indicated there will be an error during parsing of the options.

desc A string describing the option. This string is displayed when the **usage** function is called or when a parsing error is detected.

mode A value taken from the OptionMode enumeration.

Add	Each time a value for the option is found, the new value is added to the array of values. Values can be retrieved by the **get** function using the index argument.
Replace	Each time a value for the option is found the old value is replaced with a new value. Only one value can be retrieved.
Append	Each time a value for the option is found, this value is appended to the current value. Only one value can be retrieved.

Description

These member function allow you to register new options on the O_2 options manager.

These function are registered by the **begin** member function.

Each of these functions allow you to enroll one option. There is one function for each type of option.

Returns

1 if successful.
0 if the option could not be enrolled.
-1 if there was an internal error in the option manager.

enroll_path

Summary Allows you to register hierarchical options.

Syntax **`static int o2_odbc::enroll_path (const char * path);`**

Description This member function allows you to register hierarchical options. Hierarchical options are described as a path, i.e., an ordered list of options such as:

`system.base.loadname`

The hierarchical options only work in a configuration file such as **`.o2rc`**.

Returns 0 if successful.
-1 if there was an internal error.

get_option

Summary Retrieves the value of an option.

Syntax

```
static int o2_odbc::get_option (   const char *name,
                                    char *&value,
                                    int ind = -1);

static int o2_odbc::get_option (   const char *name,
                                    long &value,
                                    int ind = -1);

static int o2_odbc::get_option (   const char *name,
                                    double &value,
                                    int ind = -1);

static int o2_odbc::get_option (   const char *name,
                                    char &value,
                                    int ind = -1);
```

Arguments

name A string that indicates the internal name of the option as defined in the corresponding **o2_odbc::enroll** member function.

value This argument points to the returned value.

ind An index that is used if the user enters an option several times. If you have registered the option with the replace or append mode, you should set this argument to -1. If the index is -1, the last value entered by the end-user is returned. If the index is >= 0, the index-th value is returned. If the index is too large, the returned value is NULL.

Description This member function allows you to retrieve the value of the registered options. This function should only be called for options that are registered.

This function is intended to be used in the check function, which can be registered by the **o2_odbc::begin** member function.

Returns 0 if successful.
-1 if the option cannot be retrieved (i.e., the option is not registered).

7

O2 ODBC Reference

init

Summary Starts up a connection to an **o2open_dispatcher**.

Syntax **`int o2_odbc::init();`**

Arguments None.

Description This function connects the server to the dispatcher.

Returns Zero if the operation was successful. Else a non-zero value.

set...

Summary Sets the various session parameters.

Syntax

```
void o2_odbc::set_default_env();

void o2_odbc::set_enroll();

void o2_odbc::set_libname(char **);

void o2_odbc::set_libpath(char **);

void o2_odbc::set_servername(char *);

void o2_odbc::set_swapdir(char *);

void o2_odbc::set_sysdir(char *);

void o2_odbc::set_systemname(char *);
```

Description Explicitly set various session parameters before beginning the session with **o2_odbc::begin(argc, argv, mode);**.

set_default_env(); allows your C++ program to use the standard O_2 configuration file (**.o2rc**), the standard O_2 environment variable options (**O2OPTIONS**), and the standard O_2 command options:
-system, which defines the O_2 system name,
-server, which defines the name of the O_2 server host,
-help, which displays the help text for the program, and
-verbose, which enables the verbose mode.

Returns Nothing.

Note

Refer to **o2_odbc::begin()** for additional information.

usage

Summary Displays a description of the registered options.

Syntax **`static void o2_odbc::usage ();`**

Description This member function displays a usage description of the registered options. All valid options are displayed with the contents of the **`desc`** argument of the **`o2_odbc::enroll`** member function.

Returns 0 if successful.
-1 if there was an internal error.

7.2 The O_2 ODBC Commands

The commands outlined in this section should be found in the bin/<platform> subdirectory of the O_2 installation directory. These commands are:

- **o2odbc_dump_base**
- **o2odbc_server**
- **o2open_dispatcher**
- **o2sql_export**
- **o2sql_query**

o2odbc_dump_base

Summary Generates a logical dump of an O_2 data source in a given ASCII file.

Syntax **o2odbc_dump_base**

connection_string

output_file

Mandatory arguments

- **connection_string**

  This argument must be defined as specified in Section 5.2.3. It is used by the **o2odbc_dump_base** program to connect to a given O_2ODBC data source.

- **output_file** This is the name of the file where the SQL commands are dumped into.

Optional arguments

None.

Description The logical dump of a virtual database consists of all table creation and row insertion SQL commands. Commands are terminated by ";".

The generated ASCII file can be given as input to a program that sends each command to execution on a given database. This allows the contents of a dumped database to be loaded elsewhere.

A dispatcher and a server must be running, as the **o2odbc_dump_base** tool is an O_2ODBC client. In addition, the ODBC server used by the **o2odbc_dump_base** tool must be running in manual mode.

Environment variables

None.

Files An output file is generated. If a file with the same name already exists, it is overwritten.

See also **o2odbc_server**, **o2open_dispatcher**

o2odbc_server

Summary Starts an O_2ODBC server.

Syntax **o2odbc_server**

```
[-system       system_name]
[-server       server_host]
[-dispatcher   dispatcher_host]
[-commit_mode  commit_mode]
[-verbose]
```

Mandatory arguments

None.

Optional arguments

Default arguments (like **-system** or **-server** arguments) are accepted according the general option mechanism of O_2(see the *System Administration Reference Manual*).

- **-system system_name**

  Specifies the O_2 system name.

- **-server server_host**

  Specifies the O_2 server host name. This must be the name of a machine on the network.

- **-dispatcher dispatcher_host**

  Specifies the O_2OpenAccess dispatcher host. This must be the name of a machine on the network.

- **-commit_mode commit_mode**

  Specifies the commit mode on which the server will run. Possible values are **auto** (for auto-commit mode) and **manual** (for manual mode). If not specified, the auto-commit mode is set by default.

- **-verbose**

  Displays additional information about the **o2odbc_server** activity, i.e. sets the verbose mode on.

7

O2 ODBC Reference

Description

This command starts a new O_2ODBC server on a machine. An O_2ODBC server processes O_2ODBC client requests.

When started, **o2odbc_server** establishes a connection with an O_2OpenDispatcher (**o2open_dispatcher**) which must already be running and establishes also a connection with a named O_2 database system through an **o2server**, which must also already be running.

Environment variables

- **O2HOME**

Specifies the installation directory of O_2. This variable is mandatory.

Files

The file **/etc/services** (Unix) or **$WINDIR\system32\drivers\etc\services** (Windows NT) contains the dispatcher host name and port number.

See also

o2open_dispatcher, **o2server**

o2open_dispatcher

Summary Starts an O_2OpenAccess dispatcher.

Syntax **o2open_dispatcher**

[-verbose]

Mandatory arguments

None.

Optional arguments

- **-verbose**

Displays additional information about the **o2open_dispatcher** activity, i.e. sets the verbose mode on.

Description

This command starts a new O_2OpenAccess dispatcher on a machine. An O_2OpenDispatcher registers all O_2ODBC servers running on a LAN and is queried to get the address of a server able to answer to an O_2ODBC client requests.

A server is chosen by the dispatcher according to a heuristics and based on connection options set by the client. A score is computed for each server running and the server with the best score is returned to the client.

The following elements enter in the computation of the score:

- a server is running on the same host as the client
- a server is already connected to the database to which the client wants to connect
- the current load of each server (the number of connected clients)
- the value of the **SQL_AUTOCOMMIT** connection option (specified by the client with the **SQLSetConnectOption** or the default value **SQL_AUTOCOMMIT_ON**)

Environment variables

None.

Files

The file **/etc/services** (Unix) or **$WINDIR\system32\drivers\etc\services** (Windows NT) contains the dispatcher host name and port number.

See also

o2odbc_server, **o2server**, **o2odbc_dump_base**

o2sql_export

Summary

View schema generation program.

Syntax

```
o2sql_export
       [-system system_name]
       [-server server_host]
        -schema schema_name
        -view   view_name
       [-config config_file]
       [-output output_file]
       [-verbose]
```

Mandatory arguments

- **-schema schema_name**

  This is the name of a schema for which the view **view_name** is to be derived.

- **-view view_name**

  This is the name of the view to be derived for schema **schema_name**. Many different views can be derived for the same schema.

Optional arguments

Default arguments (like **-system** or **-server** arguments) are accepted according the general option mechanism of O_2(see the *System Administration Reference Manual*).

- **-system system_name**

  Specifies the O_2 system name.

- **-server server_host**

Specifies the O_2 server host name. This must be the name of a machine on the network.

- **-config config_file**

  Specifies a configuration to be used when exporting the O_2 schema as a relational schema.

- **-output output_file**

  If an output file is specified through argument **-output** then the generated view schema definition is dumped into this file.

- **-verbose**

  Displays additional information about the **o2sql_export** activity, i.e. sets the verbose mode on.

Description

The **o2sql_export** tool allows view schemas to be created and modified. When started, it establishes a connection with a named O_2 database system through an **o2server**, which must already be running.

Environment variables

- **O2HOME**

  Specifies the installation directory of O_2. This variable is mandatory.

Files

An output file is generated if the option **-output** is specified. If a file with the same name already exists, it is overwritten.

An input configuration file is used if the option **-config** is specified. If the file cannot be opened, an error is reported and the program aborts.

See also

o2server, **o2sql_query**, **o2odbc_server**

o2sql_query

Summary SQL interactive shell.

Syntax **o2sql_query**

```
[-system system_name]
[-server server_host]
 -base   base_name
 -view   view_name
[-output output_file]
[-auto_commit]
[-verbose]
```

Mandatory arguments

None.

Optional arguments

Default arguments (like **-system** or **-server** arguments) are accepted according the general option mechanism of O_2(see the *System Administration Reference Manual*).

- **-system system_name**

  Specifies the O_2 system name.

- **-server server_host**

  Specifies the O_2 server host name. This must be the name of a machine on the network.

- **-base base_name**

  The name of a base on which the view **view_name** is to be activated.

- **-view view_name**

  The name of the view to be activated on base **base_name**. The view must have been previously derived with the **o2sql_export** tool for the schema from which the base **base_name** is an instance.

- **-output output_file**

If an output file is specified through argument **-output** then the generated view schema definition is dumped into this file.

- **-auto_commit**

  Specifies that the shell must run in auto-commit mode, i.e. a commit will be automatically performed after the execution of each SQL statement. The default mode is the manual mode, whereby commits and/ or rollbacks must be explicitly executed with the appropriate shell commands (**commit work** and **rollback work**).

- **-verbose**

  Displays additional information about the **o2sql_query** activity, i.e. sets the verbose mode on.

Description

The **o2sql_query** tool allows view schemas to be activated on a given database. Once a view is activated on an O_2 base, SQL commands can be executed on the resulting virtual database. The view schema can also be inspected through specific shell commands (see Section [Ref: o2sqlquery]) for more details).

When started, **o2sql_query** establishes a connection with a named O_2 database system through an **o2server**, which must already be running.

Environment variables

- **O2HOME**

  Specifies the installation directory of O_2. This variable is mandatory.

Files An output file is generated if the option **-output** is specified. If a file with the same name already exists, it is overwritten.

See also **o2server**, **o2sql_export**

A Syntax for View Customization

The syntax for view customization through a configuration file is given below in EBNF format. Reserved words are "**quoted**" and non terminal symbols are given *in italics*. The symbol | represents a choice (a disjunction), brackets ({ and }) represent zero or many occurrences and square brackets ([and]) represent zero or one occurrence.

The non-terminal *query_expression* corresponds to a valid quoted OQL expression, whereas *schema_name*, *class_name*, *proc_name*, *table_name*, *collection_name*, *column_name*, *method_name* and *attribute_name* correspond to valid O_2 identifiers.

The non-terminal *proc_description* corresponds to a quoted string and is intended to allow a short text describing the semantics of the procedure to be attached to the procedure declaration in the SQL catalog.

A Syntax for View Customization

schema ::= "`view schema`" *schema_name* "`from`" *schema_name* "`;`"
[*hide_command*] [*proc_command*] [*export_list*]
hide_command ::= "`hide`" *class_name_list* "*;*"
proc_command ::= "`stored procedure`" *proc_list*"*;*"
export_list ::= *export_command* { "`;`"*export_command* }
export_command ::= *export_class_command*
| *export_collection_command*
export_class_command ::= "`export class`" *class_name* ["`as`" *table_name*]
["`define key`" *attribute_name*]"`;`"
["`hide`" *attribute_name_list*]"`;`"
["`redefine`" *virtual_attribute_list*]"`;`"
["`method`" *method_name_list*]"`;`"
["`extent`" *query_expression*]"`;`"
["`with`" *data_update_clause_list*]"`;`"
"end"
export_collection_command ::= "`export collection`" *collection_name* "`in class`" *class_name*
["`as`" *table_name*]
["`redefine`" *virtual_attribute_list*]
"end"
class_name_list ::= *class_name* { "`,`" *class_name* }
proc_list ::= *proc* { "`,`" *proc* }
proc ::= *proc_lang proc_name* [*proc_description*]
proc_lang ::= "`C++:`" | "`O2C:`" \mid
virtual_attribute_list ::= *virtual_attribute* { "`,`" *virtual_attribute* }
virtual_attribute ::= *path* "`as`" *column_name*
attribute_name_list ::= *attribute_name* { "`,`" *attribute_name* }
method_name_list ::= *virtual_method* { "`,`" *virtual_method* }
virtual_method ::= *method_name* "`as`" *column_name*
data_update_clause_list ::= *data_update_clause* { "`,`" *data_update_clause* }
data_update_clause ::= "`insert`"
| "`update`"
| "`delete`"
collection_name ::= *class_name* { "`.`" *path* }
path ::= *attribute_name* { "`.`" *attribute_name* }

B SQLGETINFO Return Values

We list below the C language **`#define`**'s for the **`fInfoType`** argument and the corresponding values returned by the ODBC API function **`SQLGetInfo`**.

B SQLGETINFO Return Values

fInfoType	Returned Value
SQL_ACTIVE_CONNECTIONS	64
SQL_ACTIVE_STATEMENTS	64
SQL_DATA_SOURCE_NAME	a long pointer to DSN
SQL_DRIVER_HDBC	Handled by the driver manager
SQL_DRIVER_HENV	Handled by the driver manager
SQL_DRIVER_HSTMT	Handled by the driver manager
SQL_DRIVER_NAME	a long pointer to "O2 Technology Driver"
SQL_DRIVER_VER	a long pointer to "02.01.0000"
SQL_FETCH_DIRECTION	SQL_FD_FETCH_NEXT
SQL_ODBC_API_CONFORMANE	SQL_OAC_LEVEL1
SQL_ODBC_VER	a long pointer to "02.10"
SQL_ROW_UPDATES	a long pointer to "N"
SQL_ODBC_SAG_CLI_CONFORMANCE	SQL_OSCC_COMPLIANT
SQL_SERVER_NAME	a long pointer to ""
SQL_SEARCH_PATTERN_ESCAPE	a long pointer to ""
SQL_ODBC_SQL_CONFORMANCE	SQL_OSC_CORE
SQL_DBMS_NAME	a long pointer to "O2 Technology"
SQL_DBMS_VER	a long pointer to "05.00.0000"
SQL_ACCESSIBLE_TABLES	a long pointer to "Y"
SQL_ACCESSIBLE_PROCEDURES	a long pointer to "Y"
SQL_PROCEDURES	a long pointer to "Y"
SQL_CONCAT_NULL_BEHAVIOR	0
SQL_CURSOR_COMMIT_BEHAVIOR	SQL_CB_DELETE
SQL_CURSOR_ROLLBACK_BEHAVIOR	SQL_CB_DELETE
SQL_DATA_SOURCE_READ_ONLY	a long pointer to "N"
SQL_DEFAULT_TXN_ISOLATION	SQL_TXN_SERIALIZABLE
SQL_EXPRESSIONS_IN_ORDERBY	a long pointer to "N"
SQL_IDENTIFIER_CASE	SQL_IC_SENSITIVE
SQL_IDENTIFIER_QUOTE_CHAR	a long pointer to "\"{"}
SQL_MAX_COLUMN_NAME_LEN	0
SQL_MAX_CURSOR_NAME_LEN	18
SQL_MAX_OWNER_NAME_LEN	0
SQL_MAX_PROCEDURE_NAME_LEN	0
SQL_MAX_QUALIFIER_NAME_LEN	0
SQL_MAX_TABLE_NAME_LEN	0
SQL_MULT_RESULT_SETS	a long pointer to "N"
SQL_MULTIPLE_ACTIVE_TXN	a long pointer to "Y"
SQL_OUTER_JOINS	a long pointer to "N"
SQL_OWNER_TERM	a long pointer to ""
SQL_PROCEDURE_TERM	a long pointer to "stored procedure"
SQL_QUALIFIER_NAME_SEPARATOR	a long pointer to ""

fInfoType	Returned Value
SQL_QUALIFIER_TERM	a long pointer to "database"
SQL_SCROLL_CONCURRENCY	SQL_SCCO_READ_ONLY
SQL_SCROLL_OPTIONS	SQL_SO_FORWARD_ONLY
SQL_TABLE_TERM	a long pointer to "O2 name"
SQL_TXN_CAPABLE	SQL_TC_ALL
SQL_USER_NAME	a long pointer to ""
SQL_CONVERT_FUNCTIONS	0
SQL_NUMERIC_FUNCTIONS	SQL_FN_NUM_ABS | SQL_FN_NUM_MOD
SQL_STRING_FUNCTIONS	SQL_FN_STR_CONCAT | SQL_FN_STR_LENGTH
SQL_SYSTEM_FUNCTIONS	0
SQL_TIMEDATE_FUNCTIONS	0
SQL_CONVERT_BIGINT	0
SQL_CONVERT_BINARY	0
SQL_CONVERT_BIT	0
SQL_CONVERT_CHAR	0
SQL_CONVERT_DATE	0
SQL_CONVERT_DECIMAL	0
SQL_CONVERT_DOUBLE	0
SQL_CONVERT_FLOAT	0
SQL_CONVERT_INTEGER	0
SQL_CONVERT_LONGVARCHAR	0
SQL_CONVERT_NUMERIC	0
SQL_CONVERT_REAL	0
SQL_CONVERT_SMALLINT	0
SQL_CONVERT_TIME	0
SQL_CONVERT_TIMESTAMP	0
SQL_CONVERT_TINYINT	0
SQL_CONVERT_VARBINARY	0
SQL_CONVERT_VARCHAR	0
SQL_CONVERT_LONGVARBINARY	0
SQL_TXN_ISOLATION_OPTION	SQL_TXN_SERIALIZABLE
SQL_ODBC_SQL_OPT_IEF	a long pointer to "N"

B SQLGETINFO Return Values