DCOracle2 C Layer API

Release 1.1
April 17, 2002

Matthew T. Kromer

Zope Corporation
Lafayette Technology Center
513 Prince Edward Street
Fredericksburg, VA 22408
(888) 344-4332 (sales)
http://www.zope.com

DCOracle2 C Layer API Documentation

This document describes the various objects and interfaces to the dco2 C binding for Python to the Oracle 8 Call Interface (oci8). Under normal circumstances, no exposure to dco2 is required for users of DCOracle2. **This reference is provided for completeness, but does not form an official application programming interface -- as such, it may be changed at any time, without warning.**

Usage

```
import dco2

ServerContext = dco2.connect(userid, password, database)
tracebacklist = dco2.Traceback()
oracledate = dco2.OracleDate(ticks)

ServerContext.commit()
ServerContext.prepare()
ServerContext.rollback()
description = ServerContext.describe(schemaname)
bindingarray = ServerContext.BindingArray(elements, size, type)

Cursor = ServerContext.cursor()

Cursor.prepare(statement)
Cursor.bindbypos(pos, value)
Cursor.bindbyname(name, value)
Cursor.execute()
description = Cursor.describe()
ResultSet = Cursor.fetch(n)
Cursor.setPrefetch(records[, size])
Cursor.rowcount()

value = ResultSet[i][j].value()
string = str(ResultSet[i][j])

length = LobLocator.length()
data = LobLocator.read([bytes, [offset, [csid, [csfrm]]]])
count = LobLocator.write(string, [offset, [csid, [csfrm]]])
LobLocator.trim(length)

BindingArray.setStatic({0|1})
BindingArray.type([type])
BindingArray.width()
```

Objects

```
ServerContext
Cursor
BindingArray
ResultSet
ResultSetItem
OracleDate
LobLocator
Error
Warning
InterfaceError
DatabaseError[1]
InternalError
OperationalError
ProgrammingError
IntegrityError
DataError
NotSupportedError
```

[1]The DatabaseError object is the error returned when an OCI error is raised, regardless of the meaning of the OCI error.

dco2.connect(*userid*, *password*, *database*)

The Connect method connects to the database with the specified credentials; it returns a `ServerContext` object. When the `ServerContext` object goes out of scope (is deallocated) the session is terminated. If *database* is "", the default database is used. If *userid* and *password* are "", then `OCI_CRED_EXT` external credentials are used for authentication as opposed to `OCI_DEFAULT` authentication.

dco2.Traceback()

The Traceback method on the module will return the current traceback table as a list of lists. The first element for each row is the timestamp, the second element is the trace code, and the remaining elements are trace data.

dco2.OracleDate(*ticks*)

Constructs an OracleDate object based on the timestamp in `ticks`, which should be given in GMT. OracleDate objects are used to pass DATE fields in and out of Oracle when binding positionally. When referred to in a string context, the OracleDate returns a string representation as "yyyy-mm-dd hh:mm:ss" and in an integer context, returns the timestamp as seconds since the epoch.

ServerContext.commit([2])

The commit method commits the data in the current transaction. If the optional parameter 2 is specified, the commit is performed using `OCI_TRANS_TWOPHASE` vs. the default of `OCI_DEFAULT`. Note that in a globally managed transaction a two phase commit can introduce an indeterminate state.

ServerContext.prepare()

The prepare method prepares the data to be committed in the current transaction.

ServerContext.rollback()

The rollback method discards any changes in the current transaction.

SeverContext.describe(*schemaname*)

This method gets back a set of nested dictionaries describing the named schema object, returned from the OCIDescribeAny call.

*ServerContext.*BindingArray(*elements, size, type*)

This returns a BindingArray object of at least *elements* length, each element being *size* bytes long, with the specified Oracle *type*. BindingArrays may be used with cursor binding operations for operations which require writing back into the bound parameters (stored procedure calls).

ServerContext.cursor()

The cursor method generates a cursor which is a combination of Oracle statement handles, statement definitions, binding data, and result information. A cursor must be allocated for any SQL work to be performed.

Cursor.prepare(*statement*)

The prepare method parses a SQL statement and prepares it for execution. No server round trip is involved in preparation.

Cursor.bindbypos(*pos, object,* [*type*]*)*

The bindbypos method binds an object to an input position in a statement. The *pos* parameter starts with 1, for the first positional parameter. For example:

```
c.prepare("select * from emp where empno = :1")
c.bindbypos(1, empno)
```

See the section on ‘‘Type Conversion Rules’’ below for a detailed description of how an object is bound.

Cursor.bindbypos will throw a `ValueError` if it cannot adequately bind *object*.

Note that a ResultSet or ResultSet Item is not supported by bindbypos, meaning the output of a SELECT cannot be immediately reused!

Cursor.bindbyname(*name*, *value*)

The functional eqivalent of *Cursor*.bindbypos, this binds a variable to a statement by name, rather than by position. All relelvant *Cursor*.bindbypos comments apply.
Note that the name used should be ":name", with a leading colon. The dco2 module does not automatically insert the colon.

***Cursor*.execute()**

The execute method of the cursor will cause the statement handle bound by the prepare method to be executed. After an execution, the statement result definitions are obtained and made available for the *Cursor*.describe() method.

***Cursor*.describe()**

The describe method of the cursor will return a list of 7-tuples `(name, type, size, size, precision, scale, nullok)` for each column in the result of the statement. The Python DB API 2.0 specification states that the first size is the display_size and the second is the internal_size. The Oracle maximum external data representation size field is returned for both values.

***Cursor*.fetch([*n*])**

The fetch method of the cursor will fetch the next *n* results from the results of the statement The default is to fetch one result. The return value of *Cursor*.fetch() is a list of ResultSet objects. Each ResultSet object acts as a sequence. An individual value of a ResultSet is a ResultSetItem.

Note: *Cursor*.fetch returns a list of columns, not a list of rows. The DCOracle2 fetch methods will reorder the columns into rows.

***Cursor.*setPrefetch(*rows* [, *size*])**

Sets the prefetching on the cursor. Size is the size in bytes to allocate. Oracle will prefetch up to *rows* results or until a buffer or *size* is filled when executing queries.

***Cursor.*rowcount()**

Gets the row count of the last operation. For non-select operations, this is the number of rows affected by the operation. For select operations, the value is updated as rows are fetched, but does not indicate the total number of records available to be returned.

***ResultSetItem*.value()**

Generates an object representing the value of the result set Item. If there is no type translator to convert the Oracle type to a Python object, a `TypeError` will be raised. In this instance, the application can refer to str(*ResultSetItem*) to get the data in unaltered format.

Note: Currently, Numbers are retrieved as type SQL_STR (strings) not SQL_NUM. Numeric translation from the string format happens in dco2's output conversion functions. Numeric conversion is based on the precision (length) and scale (number of decimal places). If the number has a scale, it is not an integer, and so is returned as a Python float. If it has no scale, and the precision is less than 10, the result is returned as a Python int. Integers with a precision of 10 or higher are returned as Python longs. It is possible, however, to use the str() representation function on the ResultSetItem object to retrieve the original string however.

***LobLocator*.length()**

Returns the length of the LOB referenced by *LobLocator*.

***LobLocator*.read([*size*, [*offset*, [*csid*, [*csfrm*]]]])**

Returns the data from the *LobLocator* for *size* bytes, which defaults to the end of the LOB. Optionally, the starting LOB *offset* may be provided, as may be the character set ID *csid* and character set form *csfrm*. Both *csid* and *csfrm* default and can be safely omitted unless the default character set Oracle was installed with is not appropriate.

***LobLocator*.write(*string*, [*offset*, [*csid*, [*csfrm*]]])**

Writes *string* to the selected LOB, optionally starting *offset* bytes into the LOB. The character set parameters *csid* and *csfrm* default to the Oracle installation defaults and can be safely omitted.

Note: Empty LOBs can be inserted by using the EMPTY_BLOB() or EMPTY_CLOB() Oracle functions. LOB objects must be selected for updating before they can be written, for example:

```
import DCOracle2

db = DCOracle2.connect('scott/tiger')
c = db.cursor()
c.execute('insert into blobtest values (:1, :2, EMPTY_BLOB())', 'Shane Hathaway', 4)
c.execute('select * from blobtest where id=4 for update')
r = c.fetchone()
lob = r[2]
lob.write(shanepic)
db.commit()
```

***LobLocator*.trim(*length*)**

Trims the LOB to the specified length, which must be less than or equal to the current length.

***BindingArray*.setStatic({0|1})**

Sets the binding type of the binding array, 1 for static binding, 0 for dynamic binding. The internal function Cursor_bind in dco2.c uses this to set up the binding values.

***BindingArray*.type([*type*])**

Gets/sets the Oracle type associated with the BindingArray.

***BindingArray*.width()**

Gets the *size* field of the BindingArray (the width of each element.)

BUILDING and INSTALLATION

Generally, building the DCO2 module is as simple as setting up the Oracle build environment and typing `make`. Here's an example:

```
bane(1)$ . oraenv
ORACLE_SID = [ORAC] ? <enter>
bane(2)$ make
(cd src; \
./testora)
Checking ORACLE_HOME...passed.
Checking for Oracle8i...not found, Oracle 8.0 assumed.
(cd src; \
cp -p Setup.in Setup )
(cd src; \
make -f Makefile.pre.in boot PYTHON=python || ./setuperrors; \
make dummy || ./setuperrors)
make[1]: Entering directory ‘/stripe0/home/matt/wrk/dc/DCO2/src’
rm -f *.o *~
rm -f *.a tags TAGS config.c Makefile.pre python sedscript
rm -f *.so *.sl so_locations
VERSION=‘python -c "import sys; print sys.version[:3]"‘; \
installdir=‘python -c "import sys; print sys.prefix"‘; \
exec_installdir=‘python -c "import sys; print sys.exec_prefix"‘; \
make -f ./Makefile.pre.in VPATH=. srcdir=. \
        VERSION=$VERSION \
        installdir=$installdir \
        exec_installdir=$exec_installdir \
        Makefile
make[2]: Entering directory ‘/stripe0/home/matt/wrk/dc/DCO2/src’
sed -n \
 -e ’1s/.*/1i\\/p’ \
 -e ’2s%.*%# Generated automatically from Makefile.pre.in by sedscript.%p’ \
 -e ’/^VERSION=/s/^VERSION=[    ]*\(.*\)/s%@VERSION[@]%\1%/p’ \
 -e ’/^CC=/s/^CC=[      ]*\(.*\)/s%@CC[@]%\1%/p’ \
 -e ’/^CCC=/s/^CCC=[    ]*\(.*\)/s%#@SET_CCC[@]%CCC=\1%/p’ \
 -e ’/^LINKCC=/s/^LINKCC=[      ]*\(.*\)/s%@LINKCC[@]%\1%/p’ \
 -e ’/^OPT=/s/^OPT=[    ]*\(.*\)/s%@OPT[@]%\1%/p’ \
 -e ’/^LDFLAGS=/s/^LDFLAGS=[    ]*\(.*\)/s%@LDFLAGS[@]%\1%/p’ \
 -e ’/^DEFS=/s/^DEFS=[  ]*\(.*\)/s%@DEFS[@]%\1%/p’ \
 -e ’/^LIBS=/s/^LIBS=[  ]*\(.*\)/s%@LIBS[@]%\1%/p’ \
 -e ’/^LIBM=/s/^LIBM=[  ]*\(.*\)/s%@LIBM[@]%\1%/p’ \
 -e ’/^LIBC=/s/^LIBC=[  ]*\(.*\)/s%@LIBC[@]%\1%/p’ \
 -e ’/^RANLIB=/s/^RANLIB=[      ]*\(.*\)/s%@RANLIB[@]%\1%/p’ \
 -e ’/^MACHDEP=/s/^MACHDEP=[    ]*\(.*\)/s%@MACHDEP[@]%\1%/p’ \
 -e ’/^SO=/s/^SO=[      ]*\(.*\)/s%@SO[@]%\1%/p’ \
 -e ’/^LDSHARED=/s/^LDSHARED=[  ]*\(.*\)/s%@LDSHARED[@]%\1%/p’ \
 -e ’/^CCSHARED=/s/^CCSHARED=[  ]*\(.*\)/s%@CCSHARED[@]%\1%/p’ \
 -e ’/^LINKFORSHARED=/s/^LINKFORSHARED=[        ]*\(.*\)/s%@LINKFORSHARED[@]%\1%/p’ \
 -e ’/^prefix=/s/^prefix=\(.*\)/s%^prefix=.*%prefix=\1%/p’ \
 -e ’/^exec_prefix=/s/^exec_prefix=\(.*\)/s%^exec_prefix=.*%exec_prefix=\1%/p’ \
 /usr/lib/python1.5/config/Makefile >sedscript
echo "/^#@SET_CCC@/d" >>sedscript
echo "/^installdir=/s%=.*%=     /usr%" >>sedscript
echo "/^exec_installdir=/s%=.*%=/usr%" >>sedscript
echo "/^srcdir=/s%=.*%=         .%" >>sedscript
echo "/^VPATH=/s%=.*%=          .%" >>sedscript
echo "/^LINKPATH=/s%=.*%=       %" >>sedscript
echo "/^BASELIB=/s%=.*%=        %" >>sedscript
echo "/^BASESETUP=/s%=.*%=      %" >>sedscript
sed -f sedscript ./Makefile.pre.in >Makefile.pre
/usr/lib/python1.5/config/makesetup \
         -m Makefile.pre -c /usr/lib/python1.5/config/config.c.in Setup -n
/usr/lib/python1.5/config/Setup
make -f Makefile do-it-again
make[3]: Entering directory ‘/stripe0/home/matt/wrk/dc/DCO2/src’
/usr/lib/python1.5/config/makesetup \
         -m Makefile.pre -c /usr/lib/python1.5/config/config.c.in Setup -n
/usr/lib/python1.5/config/Setup
make[3]: Leaving directory ‘/stripe0/home/matt/wrk/dc/DCO2/src’
make[2]: Leaving directory ‘/stripe0/home/matt/wrk/dc/DCO2/src’
make[1]: Leaving directory ‘/stripe0/home/matt/wrk/dc/DCO2/src’
make[1]: Entering directory ‘/stripe0/home/matt/wrk/dc/DCO2/src’
make[1]: ‘dummy’ is up to date.
make[1]: Leaving directory ‘/stripe0/home/matt/wrk/dc/DCO2/src’
( cd src; \
make || ./builderrors)
make[1]: Entering directory ‘/stripe0/home/matt/wrk/dc/DCO2/src’
```

```
gcc -fPIC  -I/stripe1/oracle/8.0.5/rdbms/demo -I/stripe1/oracle/8.0.5/network/public -
I/stripe1/oracle/8.0.5/plsql/public -I/stripe1/oracle/8.0.5/rdbms/public -g -O2 -
I/usr/include/python1.5 -I/usr/include/python1.5 -DHAVE_CONFIG_H -c ./dco2.c
gcc -shared  dco2.o  -L/stripe1/oracle/8.0.5/lib/ -lclntsh -lcommon -lcore4 -lnlsrtl3 -Wl,-
rpath,/stripe1/oracle/8.0.5/lib -o dco2.so
make[1]: Leaving directory '/stripe0/home/matt/wrk/dc/DCO2/src'
cp src/dco2.so DCOracle2
```

That's it! The makefile will try to detect as much as it can and will properly copy the built dco2.so module to the DCOracle2 directory.

A number of problems can happen during building:

Oracle environment not set up correctly
If the environment variable `ORACLE_HOME` is not set, then the compilation will fail to find the proper Oracle include and library files.

Oracle development files not installed
Oracle installs the C headers to build OCI programs in `$ORACLE_HOME/rdbms/demo`, `$ORACLE_HOME/network/public`, and `$ORACLE_HOME/plsql/public`. If these directories do not have the proper include files, the build will not succeed. Oracle typically only installs these files when a complete server installation is performed.

Python development Makefile not available
If the make fails stating that `/usr/lib/python1.5/config/Makefile` (or similar) cannot be built, it is symptomatic of the problem that the Python development is not installed. On a RedHat Linux system this would be the *python-devel* package.

Python is not linked against libpthread (Linux)
Oracle on Linux is linked against libpthread. Some recent linux distributions may include versions of Python that are not linked against libpthread. When dco2 is loaded, it will load the Oracle shared libraries which are incompatible with the libpth libraries of the existing Python binary. This will cause random lockups to occur to the process. The only solution is to acquire a Python that is not linked against libpth, but is instead linked against libptthread. This may be accomplished by either obtaining a different Python binary package, or rebuilding Python from source.

DIAGNOSIS

Trace Table Debugging

DCO2 keeps an internal trace table, controlled by four environment variables:

DCO2TRACELOG	(trace file name)
DCO2TRACEDUMP	(dump file name)
DCO2TRACESIZE	(trace table size)
DCO2TRACEFLAGS	(trace filter)

The variable DCO2TRACELOG, if set, names a file to record all trace events (as they occur). The variable DCO2TRACEDUMP names a file to dump the trace table to if an Oracle error occurs (but this is not the only reason dco2 will raise an error).

DCOracle2 will sometimes trap errors within a try/except, so this will cause a trace dump to take place if DCO2TRACEDUMP is set. Each new Oracle error will append the trace file to the dump file.

The variable DCO2TRACESIZE specifies the number of entries in the trace table (the default is 512). Each entry in the trace table is 10 words (40 bytes) long, so the default trace table consumes 20K of storage. Setting the size of the trace table to 0 will also set the DCO2TRACEFLAGS to 0. DCO2TRACEFLAGS is the binary OR (sum) of the following values:

1	Entry
2	Exit
4	Error
8	(unused)
16	Program
32	Oracle
64	Info
128	Verbose

The typical codes used are:

17	Function Entry
18	Function Exit
33	Oracle call
34	Oracle return
36	Oracle error
65	Info arguments
66	Info result

161	Oracle handle call
162	Oracle handle return

Up to 7 arguments are also recorded in the trace table, with the first argument being a function name.

Debugging with gdb

If dco2 causes a core dump (typically a segment fault) it is useful to obtain a traceback of what was occurring at the time of the crash. The gnu debugger `gdb` should be run against python to obtain the internal trace table. The following is an example:

```
% gdb python
...
(gdb) run badprog.py
...
<segmentation fault>
(gdb) where
(gdb) call Traceprint()                (prints trace table to stdout)
(gdb) call Tracedump()                 (saves trace table to "dco2.tdm")
```

The output of the trace table should be included in any support message. Also include the output from the `where` step of gdb, as this will pinpoint where in the code that the error occurred.

Problem Reporting

When submitting a problem report to Zope Corporation about DCOracle2, it is useful to include a trace dump when incorrect results are being returned. When reporting problems, please be sure and identify the following:

- Operating System
- Distribution (for Linux)
- Python Version
- Oracle Version
- DCO2TRACEDUMP generated file (where appropriate)
- Python tracebacks (where appropriate)

Type Conversion Rules

The type conversion for dco2 uses the following rules (they are not necessarily the correct rules, but this the current implementation):

Input types are converted in the internal function bindObject and obey the following rules:

1. If the object is None, bind the input as NULL.
2. If the object is a BindingArray, set the column for array input/output.
3. If the object is a Cursor, bind the column as an SQL Result Set (SQLT_RSET).
4. If the object is an OracleDate, bind the column as a a Date (SQLT_DAT).
5. If the object is a String, bind the column as a string (SQLT_STR).
6. If the object is a Long, bind the column as an integer (SQLT_INT).
7. If the object is an Integer, bind the column as an integer (SQLT_INT).
8. If the object is a Float, bind the column as a float (SQLT_FLT).

However, the DCOracle2 Python layer will pass type conversion overrides to dco2 if the object is a tuple (value, type). The input type override is either a string constant or a numeric value representing the SQL data type, e.g. 'SQLT_STR' for string. Unsupported column types can be coerced from string data in this fashion.

Output types are converted by lookup in a type table. This table is:

Type Name	**Descripton**	**Coercion**	**Internal Type**	**Result Object**
SQLT_CHR	Char		Yes	String
SQLT_NUM	Number	SQLT_STR	Yes	Automatic[2]
SQLT_INT	Integer		No	Integer
SQLT_FLT	Float		No	Float
SQLT_STR	String		No	String
SQLT_VNU	Length Prefixed Number		No	--
SQLT_PDN	Packed Decimal Number		No	--
SQLT_LNG	Long		Yes	String
SQLT_VCS	Variable Character		No	--
SQLT_NON	Null/Empty PCC desc		No	--
SQLT_RID	Row ID		Yes	--
SQLT_DAT	Date		Yes	OracleDate
SQLT_VBI	Binary VCS format		No	--
SQLT_BIN	Binary data		Yes	String
SQLT_LBI	Long binary data		Yes	String
SQLT_UIN	Unsigned Integer		No	--
SQLT_SLS	Display sign leading		No	--
SQLT_LVC	Longer longs (char)		No	--
SQLT_LVB	Longer longs (binary)		No	--
SQLT_AFC	ANSI Fixed Char		Yes	String
SQLT_AVC	ANSI Var Char		No	--
SQLT_CUR	Cursor		No	--
SQLT_RDD	Rowid Descriptor	SQLT_RID	No	--
SQLT_LAB	Label		No	--
SQLT_OSL	OS Label		No	--

[2]Automatic format means conversion to Float if the scale is greater than zero or the precision and scale are zero; conversion to Integer if the scale is less than 10, or conversion to Long otherwise.

SQLT_NTY	Named Type		Yes	--
SQLT_REF	Reference Type		Yes	--
SQLT_CLOB	CLOB		Yes	LobLocator
SQLT_BLOB	BLOB		Yes	LobLocator
SQLT_BFILE	Binary File LOB		No	--
SQLT_CFILE	Character File LOB		No	--
SQLT_RSET	Result set		No	--
SQLT_NCO	Named collection		No	--
SQLT_VST	OCIString type		No	--
SQLT_ODT	OCIDate type		No	--
SQLT_DATE	ANSI date		No	--
SQLT_TIME	Time		No	--
SQLT_TIME_TZ	Time with zone		No	--
SQLT_TIMESTAMP	Timestamp		No	--
SQLT_TIMESTAMP_TZ	Timestamp with zone		No	--

The result object column, if ‘‘--’’ means that there is no type converter for this type, and that only the raw interface from dco2’s ResultSet object can get at the value. When an output type is coerced, it is requested from Oracle as the type specified instead of the original format.

Implementation Notes

The C source dco2.c should be self-contained, requiring only system, Python, and Oracle include and library files.

Every normal entry and exit to functions in the code utilizes the TRACE macro to record a trace table entry. TRACE macro calls also surround Oracle calls, attempting to capture enough parameter data to provide a meaningful basis for diagnosis of errors. When the address of data is recorded, the address is very useful for inspection within gdb.

Most complex about operation is how type binding occurs; the two most responsible functions are ‘‘bindObject’’ and ‘‘Cursor_bind.’’ The bindObject function is responsible for inspecting the object to determine the type, and set up the data for the bind, without actually performing the bind. Cursor_bind will use this data to call OCIBindByName, OCIBindByPos, OCIBindDynamic, and OCIBindArrayOfStruct, as required. BindingArray operation is filled with nuances and side-effects; largely these involve whether or not the parameters maxarr_len and curelep get set in the OCIBind calls. The two values should only be set when an array bind to PL/SQL is desired. In practice, the code path makes the assumption that a BindingArray with size > 1 (not # of elements USED > 1) is destined for an array bind and then changing its mind when it becomes evident that the statement is used differently (i.e. it is messy).

Type coercion for input binds is done by seeing if the object is of type InstanceType; if so, it is checked for the attributes ‘‘value’’ and ‘‘type.’’ The bound value and type are set accordingly. This coercion is checked by bindObject and BindingArrayObject_ass_item.

For non-coerced types, a request type override is looked up in the type table. The override will specify more convenient external datatypes (largely for the convenience of the type conversion routines.) In this way, for instance, numeric type SQLT_NUM is requested as character SQLT_STR.

The convertOut functions mapped in the type table are called by ResultSetItem_value or BindingArrayObject_item functions to create a Python object. This result object is cached so the underlying data field may be cleared and reused. The ResultSet in particular works this way effectively ‘‘paging’’ though requests. DCOracle2 will never skip values in the ResultSet, which is inefficient if the Python application is going to throw them away (although not as inefficient as obtaining them from Oracle in the first place.)

Raw results from ResultSetItems are available through the str mechanism, which bypasses the output type conversion. Applications can use this to support additional data types.