Erlang ODBC application

version 0.9

Typeset in LaTeX from SGML source using the DOCBUILDER 3.2.2 Document System.

Contents

Chapter 1

ODBC User's Guide

The Erlang *ODBC* application is intended to be used from Erlang and provide an interface to relation databases.

1.1 Introduction

1.1.1 Introduction

This manual describes the *Erlang ODBC Application*. ODBC (Open DataBase Connectivity) was designed to provide an interface to relation databases and has become popular and is generally accepted as standard. The Erlang ODBC application provide a interface for Erlang to relational databases.

The Erlang ODBC application is a 'pure ODBC 3.0 application', and it is defined in Reference 1. The application is not compatible with ODBC 2.x, or earlier, drivers.

The Erlang ODBC application supports a subset of the Core level functionality specified in the ODBC standard.

The Erlang ODBC application consist of Erlang code and C code. The Erlang code is pre-compiled but the code written in C code must be compiled by the user. C-code consist of an import library and header files, which must be included when compiling. How this is done is described further in the next section.

1.1.2 Prerequisites

Readers of this manual are assumed to be familiar with the Erlang programming language in general.

1.2 How to compile ODBC on Windows

To be able to compile the Erlang ODBC application on Windows the following are required:

1. Erlang/OTP
2. Visual C++ version 5.0 or higher
3. ODBC drivers for your database

1.2.1 C-compiler on Windows

The C-part of Erlang ODBC application on Windows should be compiled with Visual C++ version 5.0 or higher. Visual C++ require certain environment variables to be set properly, i.e., point to the subdirectories of your Visual C++ installation:

- *PATH* - the `\bin` directory
- *LIB* - the `\lib` dircetory.
- *INCLUDE* - the `\include` dircetory.

When you install Visual C++, the batch file VCVARS32.BAT is created, which contains commands for modifying the PATH, LIB and INCLUDE environment variables. If these variables have not been set properly, run VCVARS32.BAT, located in the \bin subdirectory, before you compile at the command prompt.

1.2.2 Configuring the Erlang ODBC application

After installing Erlang, the Erlang ODBC application source code is located in the `<OTPROOT>\lib\odbc-<odbcversion>\src` subdirectory. <OTPROOT> is normally the path `"C:\Program Files\erl<erlang version>"`. In this subdirectory you find a Makefile, in which there is a path to the ODBC import library and paths to the header files used by the Erlang ODBC application. These paths must be set properly.

1.2.3 Configuring Makefile

As mention before Erlang ODBC application requires the ODBC import library (ODBC32.LIB) and header files. The Visual C++ \lib subdirectory contains the import library and \include the header files. Hence, depending on where Visual C++ is installed, set the variables in the Makefile to:

- `ODBCLIBS = "C:\Program Files\Microsoft Visual Studio\VC98\lib\Odbc32.lib"`
- `ODBCINCLUDE = "C:\Program Files\Microsoft Visual Studio\VC98\include"`

The Erlang ODBC application uses the Erl_interface header files and lib file. The variable EIROOT defines the path to the Erl_Interface application, e.g. `$(OTPROOT)\lib\erl_interface-3.2.3`. EIROOT must point to the version of Erl_Interface you intend to use, e.g., you might have to change the subdirectory `erl_interface-3.2.3` to represent a later version.

1.2.4 Compile ODBC

1. Start command prompt (The DOS windows).
2. Change to the \bin subdirectory of your Visual C++ installation.
3. Run VCVARS32.BAT by typing VCVARS32.
4. Change to `<OTPROOT>\lib\odbc-<odbcversion>\src` subdirectory.
5. Compile ODBC by typing `nmake`.

1.3 How to compile ODBC on Unix

To be able to compile the Erlang ODBC application on Unix the following are required:

1. Erlang/OTP
2. GCC (GNU Compiler Collection) C-Compiler version 2.7.2 or higher
3. ODBC drivers for your database

1.3.1 C-compiler on Unix

The C-part of Erlang ODBC application on Unix should be compiled with GCC C-compiler version 2.7.2 or higher. The GCC C-compiler must be found in the `PATH` environment variable. This is easily verified by running the Unix command `which gcc`.

1.3.2 Configuring the Erlang ODBC application

After installing Erlang, the Erlang ODBC application source code is located in the `<OTPROOT>/lib/odbc-<odbcversion>/src` subdirectory. <OTPROOT> can, for example, be the path `/usr/local/erlang`. In this subdirectory you find a Makefile, in which there is a path to the ODBC import library and the header files. These paths must be set properly.

1.3.3 Configuring Makefile

As mentioned earlier, Erlang ODBC application depends on the ODBC import library and header files, which are a part of the ODBC driver for your database.

To be able to access the import library you must set the variables ODBCLDFLAGS, ODBCLIBS and ODBCINCLUDE properly in the Makefile. ODBCLDFLAGS defines the path of ODBC import library, ODBCLIBS defines the name of ODBC import library and ODBCINCLUDE defines the path to the ODBC header files.
The Erlang ODBC application also uses header files from the Erl_Interface. Hence, the variable EIROOT, which defines the path to the Erl_Interface application, e.g., `$(OTPROOT)/lib/erl_interface-3.2.3`, must be point to the version of Erl_Interface you intend to use. To be able to use a later version, EIROOT must be updated.

1.3.4 Compile ODBC

1. Change to `<OTPROOT>/lib/odbc-<odbcversion>/src` subdirectory.
2. Compile ODBC by typing `gmake`.

1.4 Overview

1.4.1 Interfaces

The interface of the Erlang *ODBC* application divided into four parts:

- Functions for starting and stopping ODBC servers.
- The *Basic API* which are easy_to-use functions for database access with some control. A basic API function has a corresponding SQL function defined by the ODBC standard.
- The*Utility API* which supplies a few easy-to-use functions for database access with little control over details. A utility API function is a collection of some SQL functions.
- *Depricated functions* are only with for compatibility reason. Some of them do nothing and some are replaced by a function from Basic or Utility API.

An ODBC serving process can be compared to an Erlang port. It allows you to make function calls, from Erlang, to an ODBC Driver (or in reality a Driver Manager, which talks to a Driver). The Driver communicates with the database, which it is built for. The serving process also plays the role of a server and will be referred to as the server in this text.

When you start ODBC you get a new server, which handles requests by passing them on to an ODBC Driver. Requests are handled sequentially and the server cannot be connected to more than one database at any time.

> **Note:**
> An ODBC server can only be started on a named node with a cookie (start Erlang with one of the -name <nodename> or -sname <nodename> options and possibly with the -setcookie <cookie> option).

The server links to the process that starts it (the client). Should the client terminate, the server terminates too. The server is dedicated to the client process just like an Erlang port, but there are two important differences: an ODBC server accepts requests from any process (ports accept messages from the connected process only), and it does not send messages spontaneously (or deliver them from the ODBC Driver) – the ODBC server is passive.

Utility API

Using the Utility API is easy. Follow the steps below:

1. Start the server by calling `start_link/[2, 3]` .
2. Connect to the database with one of `erl_connect/[2, 3, 4, 5]`
3. Submit SQL statements to the database by calling `erl_executeStmt/[2, 3]`.
4. Disconnect by calling `erl_disconnect/[1, 2]`. At this point it is possible to connect to another database by calling `erl_connect/[2, 3, 4, 5]` again.
5. The server process dies when `stop/[1, 2]` is called.

Basic API

To be able to make full use of the Basic API it is necessary to be familiar with the ODBC standard. Here is a typical way to use it for a SELECT statement though:

1. Connect to the database: `sqlConnect/[4, 5]`.
2. Execute an SQL statement: `sqlExecDirect/[2, 3]`
3. Check if the statement generated a result set (or table), which SELECT statements do: `sqlNumResultCols/[1, 2]` returns a value greater than zero.
4. Describe the returned column of the table: `sql_describe_col/[4, 5]`.
5. Create reference for the returned column: `columnRef/0`.
6. Bind the reference to the column: `sqlBindCol/[3, 4]`.
7. Repeat steps 4 through 6 for each desired column (not necessarily all columns in the table).
8. Get the data for the columns: `sqlFetch/[1, 2]`.
9. Get the data for a column: `getData/[2, 3]`.
10. Repeat step 9 for all selected columns.
11. Repeat steps 8 through 10 until all rows in the table have been retrieved.
12. Close the cursor on the statement: `sqlCloseCursor/[1, 2]`.
13. Start over with step 2 to execute a new statement.
14. Disconnect from the database: `sqlDisConnect/[1, 2]`.

You may use the Basic API and the Utility API intermittently, but remember that certain operations performed in the Basic API (like setting different attributes defined by the ODBC standard) may affect the behaviour of the Utility API.

Choice of API

When to use which API? The Utility API can be used when none of the following is true:

- It is necessary to set an attribute. This can only be done through the Basic API. You may however use the Basic API just for setting attributes, and the Utility API for all other tasks.
- The table resulting from a SELECT statement is very big. In this case using the Utility API would consume a huge amount of memory, since the whole table is returned in one chunk. You can get around this problem by using the Basic API by retrieving fewer rows at a time.

1.5 Erlang ODBC application Examples

1.5.1 Introduction

In this and the following chapter two examples of the usage of the `ODBC` interface will be introduced.

The examples contain some basic operations, creating tables, writing and reading data, and dropping tables. They do the same things using the two different interface parts: the Utility API and the Basic API. The SQL used in the example is supported by Oracle8 and Intersolv's DataDirect ODBC Oracle8 Driver, but it may not work with other databases. The strings used to connect to the database depend on how your ODBC Driver is configured.

Utility API

We have a database with the user myself. We will now create a new table, insert data into it, select the data, and finally delete the table. The Erlang ODBC application has been started on a named node with some cookie, e.g `erl -sname odbc1`.

```
%% ''The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved via the world wide web at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
%% AB. All Rights Reserved.''
%%
%%     $Id$
%%
-module(utility).

-export([start/0]).


% This string depends on how your ODBC Driver is configured.
% You need to fill in your own value.
%-define(ConnectStr, "DSN=Oracle8;UID=myself;PWD=secret").


%% Note that the SQL syntax is database and ODBC Driver dependent.

start() ->
    % Start a new ODBC server. The application must already be started.
    {ok, Pid1} = odbc:start_link([], []),

    % Connect to the database.
    ok  =
        odbc:erl_connect(Pid1, ?ConnectStr, infinity),

    % Create a table
    % By default, all transactions are automatically committed.
    CreateStmt = "CREATE TABLE testtable (ID number(3), DATA char(100))",
    SqlRet1 =
        odbc:erl_executeStmt(Pid1, CreateStmt, infinity),
    io:format("executeStmt Create a table returns ~p~n",[SqlRet1]),

    % Insert data into the table
    InsertStmt = "INSERT INTO testtable VALUES(1, '1a2b3c4d5e')",
    SqlRet2 =
```

```
        odbc:erl_executeStmt(Pid1, InsertStmt, infinity),
    io:format("executeStmt Insert into table returns ~p~n",[SqlRet2]),

    % Select all rows
    % By default, all transactions are automatically committed.
    SelectStmt = "SELECT * FROM testtable",
    {selected, Columnnames, Rows} =
        odbc:erl_executeStmt(Pid1, SelectStmt, infinity),
    io:format("execute_stmt Select statement returns ~p~n",
              [{selected, Columnnames, Rows}]),

    % Delete the table
    DropStmt = "DROP TABLE testtable",
   {updated, NAffectedRows3} =
        odbc:erl_executeStmt(Pid1, DropStmt, infinity),
    io:format("Delete: Number of affected rows: ~p~n", [NAffectedRows3]),

    % Disconnect.
    odbc:erl_disconnect(Pid1, infinity),

    % Stop the server.
    odbc:stop(Pid1, infinity).
```

Basic API

We have a database with the user myself. We will now create a new table, insert data into it, select the data, and finally delete the table. The Erlang ODBC application has been started on a named node with some cookie, e.g. `erl -sname odbc1`.

```
%% ''The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved via the world wide web at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
%% AB. All Rights Reserved.''
%%
%%     $Id$
%%
-module(basic).

-export([start/0]).


% Contains macros defined by the ODBC standard.
```

```
-include("/clearcase/otp/libraries/odbc/include/odbc.hrl").


% These strings depend on how your ODBC Driver is configured.
% You need to fill in your own values.
%-define(DSN, "Oracle8").
%-define(UID, "myself").
%-define(PWD, "secret").

%% Note that the SQL syntax is database and ODBC Driver dependent.
%% Error handling is not covered by the example.

start() ->
    % Start a new ODBC server. The application must already be started.
    {ok, _Pid} = odbc:start_link({local, odbc1}, [], []),



    % Connect to the database (also loads the Driver).
    ?SQL_SUCCESS =
      odbc:sqlConnect(odbc1, ?DSN, ?UID, ?PWD, infinity),

    % Create a new table.
    CreateStmt =
        "CREATE TABLE TAB1 (ID number(3), DATA char(10))",
    ?SQL_SUCCESS = odbc:sqlExecDirect(odbc1, CreateStmt, infinity),

    % Print how many rows were affected by the statement.
    {?SQL_SUCCESS, NAffectedRows1} =
        odbc:sqlRowCount(odbc1, infinity),
    io:format("Create: Number of affected rows: ~p~n", [NAffectedRows1]),


    % Insert a new row.
    InsertStmt1 = "INSERT INTO TAB1 VALUES (1, 'a1a2a3a4a5')",
    ?SQL_SUCCESS = odbc:sqlExecDirect(odbc1,InsertStmt1, infinity),

    % Print how many rows were affected by the statement.
    {?SQL_SUCCESS, NAffectedRows2} =
      odbc:sqlRowCount(odbc1, infinity),
    io:format("Insert: Number of affected rows: ~p~n", [NAffectedRows2]),

    % Select all columns from all rows.
    SelectStmt = "SELECT * FROM TAB1",
    ?SQL_SUCCESS = odbc:sqlExecDirect(odbc1, SelectStmt, infinity),

    % Print how many columns there are in the table resulting from the
    % statement.
    {?SQL_SUCCESS, NSelectedCols} =
      odbc:sqlNumResultCols(odbc1, infinity),
    io:format("Select: Number of columns: ~p~n", [NSelectedCols]),

    % Describe the column(s) of the resulting table.
```

```
{?SQL_SUCCESS, ColName1, Nullable1} =
  odbc:sqlDescribeCol(odbc1, 1, infinity),
{?SQL_SUCCESS, ColName2, Nullable2} =
  odbc:sqlDescribeCol(odbc1, 2, infinity),

% Create references for columns
Buf1 = odbc:columnRef(),
Buf2 = odbc:columnRef(),

% Bind the refererces to the columns.
?SQL_SUCCESS = odbc:sqlBindColumn(odbc1, 1, Buf1, infinity),
?SQL_SUCCESS = odbc:sqlBindColumn(odbc1, 2, Buf2, infinity),

% Fetch the first row of selected rows.
?SQL_SUCCESS = odbc:sqlFetch(odbc1, infinity),

% Read the value from the buffer(s).
{ok, ColValue1} =
  odbc:readData(odbc1, Buf1, infinity),
io:format("Select: Column name: ~p, Data: ~p~n", [ColName1, ColValue1]),
{ok, ColValue2} =
  odbc:readData(odbc1, Buf2, infinity),
io:format("Select: Column name: ~p, Data: ~p~n", [ColName2, ColValue2]),

% Check that there are no more rows to fetch.
?SQL_NO_DATA = odbc:sqlFetch(odbc1, infinity),

% Close the cursor on the statement.
?SQL_SUCCESS = odbc:sqlCloseCursor(odbc1, infinity),

% Delete the table.
DropStmt = "DROP TABLE TAB1",
?SQL_SUCCESS = odbc:sqlExecDirect(odbc1, DropStmt, infinity),

% Print how many rows were affected by the statement.
{?SQL_SUCCESS, NAffectedRows3} =
    odbc:sqlRowCount(odbc1, infinity),
io:format("Delete: Number of affected rows: ~p~n", [NAffectedRows3]),

% Disconnect from the database.
?SQL_SUCCESS = odbc:sqlDisConnect(odbc1, infinity),

% Stop the server.
ok = odbc:stop(odbc1).
```

1.6 Erlang ODBC application Release Notes

This document describes the changes made to the Erlang ODBC application. The intention of this document is to list all incompatibilities as well as all enhancements and bug-fixes for each and every release. Each release of Erlang ODBC application constitutes one section in this document and the title is the version number.
Any comments regarding the ODBC application would be appreciated.

1.6.1 Erlang ODBC application 0.9.1

Improvements and new features

The most importend improvements in Erlang ODBC application are: Communication with database is more efficient and therefore faster. The information is encoded/decoded with functions in Erl_interface. When memory are needed, the application automatic allocate memory, and when not needed, automatic deallocate the memory.
The Erlang ODBC application has new functions, which replace the old ones. The old functions can still be used, but will be cancelled in a future. The functions are described in the Reference Manual.
Erlang ODBC application version 0.9.1.
The functionen sqlSetConnectAttr and sqlEndTran has been added. The error messages has been improved with information from the SQL function:SQLGetDiagRec.

Fixed Bugs and malfunctions

-

Incompatibilities

Erlang ODBC application from version 0.9.1 needs Erl_interface version 3.2.9 or higher.

Known bugs and problems

-

1.6.2 Erlang ODBC application 0.8.2

Improvements and new features

-

Fixed Bugs and malfunctions

Version 0.8.2 is the source code of ODBC in the directory <OTPROOT>/lib/odbc-<odbcversion>/src.

Incompatibilities

-

Known bugs and problems

-

1.6.3 Erlang ODBC application 0.8.1

Improvements and new features

- Functions for starting and stopping ODBC servers.
- The *Basic API* which is almost identical to that defined by the ODBC standard.
- The *Utility API* which supplies a few easy-to-use functions for database access with little control over details.

Fixed Bugs and malfunctions

-

Incompatibilities

-

Known bugs and problems

-

ODBC Reference Manual

Short Summaries

- Erlang Module odbc [page 18] – Erlang ODBC application

odbc

The following functions are exported:

- `start_link(Args, Options) ->`
  [page 18] Start a new ODBC server process.
- `start_link(ServerName, Args, Options) -> Result`
  [page 18] Start a new ODBC server process.
- `stop(Server) ->`
  [page 19] Stop the ODBC server process
- `stop(Server, Timeout) -> ok`
  [page 19] Stop the ODBC server process
- `sqlBindColumn(Server, ColNum, Ref) ->`
  [page 19] Assign a reference to a column in a result set
- `sqlBindColumn(Server, ColNum, Ref, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 19] Assign a reference to a column in a result set
- `sqlCloseCursor(Server) ->`
  [page 20] Close a cursor that has been opened on a statement and discards pending results
- `sqlCloseCursor(Server, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 20] Close a cursor that has been opened on a statement and discards pending results
- `sqlConnect(Server, DSN, UID, Auth) ->`
  [page 20] Establishes a connection to a database
- `sqlConnect(Server, DSN, UID, Auth, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 20] Establishes a connection to a database
- `sqlDescribeCol(Server, ColNum) ->`
  [page 21] Return the result descriptor
- `sqlDescribeCol(Server, ColNum, Timeout) -> {Result, ColName, Nullable} | {error, ErrMsg, ErrCode}`
  [page 21] Return the result descriptor

- `sqlDisConnect(Server) ->`
  [page 21] Close the connection associated with the Server
- `sqlDisConnect(Server, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 21] Close the connection associated with the Server
- `sqlEndTran(Server, ComplType) ->`
  [page 22] Request a commit or rollback operation for all active operations on all statement handles associated with a connection
- `sqlEndTran(Server, ComplType, Timeout) -> Result | {error, ErrorMsg, errCode}`
  [page 22] Request a commit or rollback operation for all active operations on all statement handles associated with a connection
- `sqlExecDirect(Server, Stmt) ->`
  [page 22] Execute a statement
- `sqlExecDirect(Server, Stmt, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 22] Execute a statement
- `sqlFetch(Server) ->`
  [page 23] Fetch a row of data from a result set
- `sqlFetch(Server, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 23] Fetch a row of data from a result set
- `sqlNumResultCols(Server) ->`
  [page 23] Return the number of columns in a result set
- `sqlNumResultCols(Server, Timeout) -> {Result, ColCount} | {error, ErrMsg, ErrCode}`
  [page 23] Return the number of columns in a result set
- `sqlRowCount(Server) ->`
  [page 24] Returns the number of rows affected by an UPDATE, INSERT, or DELETE statement
- `sqlRowCount(Server, Timeout) -> {Result, RowCount} | {error, ErrMsg, ErrCode}`
  [page 24] Returns the number of rows affected by an UPDATE, INSERT, or DELETE statement
- `sqlSetConnectAttr(Server, Attr, Value) ->`
  [page 24] set Connection Attribute
- `sqlSetConnectAttr(Server, Attr, Value, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 24] set Connection Attribute
- `readData(Server, Ref) ->`
  [page 25] Get contents of the associated column.
- `readData(Server, Ref, Timeout) -> {ok, Value}`
  [page 25] Get contents of the associated column.
- `columnRef() -> {ok, Ref}`
  [page 25] Return a reference.
- `erl_connect(Server, ConnectStr) ->`
  [page 26] Open a connection to a database
- `erl_connect(Server, ConnectStr, Timeout) ->`
  [page 26] Open a connection to a database

- sql_bind_col(Server, RefStmtHandle, ColNum, RefBuf) ->
  [page 28] Deprecated function
- sql_bind_col(Server, RefStmtHandle, ColNum, RefBuf, Timeout) ->
  Result
  [page 28] Deprecated function
- sql_close_cursor(Server, RefStmtHandle) ->
  [page 28] Deprecated function
- sql_close_cursor(Server, RefStmtHandle, Timeout) -> Result
  [page 28] Deprecated function
- sql_connect(Server, RefConnHandle, DSN, UID, Auth) ->
  [page 28] Deprecated function
- sql_connect(Server, RefConnHandle, DSN, UID, Auth, Timeout) ->
  Result
  [page 28] Deprecated function
- sql_describe_col(Server, RefStmtHandle, ColNum, BufLenColName) ->
  [page 29] Deprecated function
- sql_describe_col(Server, RefStmtHandle, ColNum, BufLenColName,
  Timeout) -> {Result,{ColName, LenColName}, SqlType, ColSize,
  DecDigs, Nullable}
  [page 29] Deprecated function
- sql_disconnect(Server, RefConnHandle) ->
  [page 29] Deprecated function
- sql_disconnect(Server, RefConnHandle, Timeout) -> Result
  [page 29] Deprecated function
- sql_driver_connect(Server, RefConnHandle, InConnStr,
  BufLenOutConnStr, DrvCompletion) ->
  [page 29] Deprecated function
- sql_driver_connect(Server, RefConnHandle, InConnStr,
  BufLenOutConnStr, DrvCompletion, Timeout) -> {Result, {OutConnStr,
  LenOutConnStr}}
  [page 29] Deprecated function
- sql_end_tran(Server, HandleType, RefHandle, ComplType) ->
  [page 29] Deprecated Function
- sql_end_tran(Server, HandleType, RefHandle, ComplType, Timeout) ->
  Result
  [page 29] Deprecated Function
- sql_exec_direct(Server, RefStmtHandle, Stmt) ->
  [page 29] Deprecated function
- sql_exec_direct(Server, RefStmtHandle, Stmt, Timeout) -> Result
  [page 29] Deprecated function
- sql_fetch(Server, RefStmtHandle) ->
  [page 29] Deprecated function
- sql_fetch(Server, RefStmtHandle, Timeout) -> Result
  [page 29] Deprecated function
- sql_free_handle(Server, HandleType, RefHandle) ->
  [page 29] Deprecated functoin
- sql_free_handle(Server, HandleType, RefHandle, Timeout) -> Result
  [page 29] Deprecated functoin

- `sql_get_connect_attr(Server, RefConnHandle, Attr, BufType) ->`
  [page 29] Deprecated function
- `sql_get_connect_attr(Server, RefConnHandle, Attr, BufType, Timeout) -> {Result, Value}`
  [page 29] Deprecated function
- `sql_get_diag_rec(Server, HandleType, RefHandle, RecNum, BufLenErrMsg) ->`
  [page 29] Depcrecated function
- `sql_get_diag_rec(Server, HandleType, RefHandle, RecNum, BufLenErrMsg, Timeout) -> {Result, SqlState, NativeErr, {ErrMsg, LenErrMsg}}`
  [page 29] Depcrecated function
- `sql_num_result_cols(Server, RefStmtHandle) ->`
  [page 30] Depreciated function
- `sql_num_result_cols(Server, RefStmtHandle, Timeout) -> {Result, ColCount}`
  [page 30] Depreciated function
- `sql_row_count(Server, RefStmtHandle) ->`
  [page 30] Deprecated function
- `sql_row_count(Server, RefStmtHandle, Timeout) -> {Result, RowCount}`
  [page 30] Deprecated function
- `sql_set_connect_attr(Server, RefConnHandle, Attr, Value, BufType) ->`
  [page 30] Deprecated function
- `sql_set_connect_attr(Server, RefConnHandle, Attr, Value, BufType, Timeout) -> Result`
  [page 30] Deprecated function
- `sql_set_env_attr(Server, RefEnvHandle, Attr, Value, BufType) ->`
  [page 30] Deprecated function
- `sql_set_env_attr(Server, RefEnvHandle, Attr, Value, BufType, Timeout) -> Result`
  [page 30] Deprecated function
- `alloc_buffer(Server, BufCType, Size) ->`
  [page 30] Deprecated function
- `alloc_buffer(Server, BufCType, Size, Timeout) -> {ok, RefBuf}`
  [page 30] Deprecated function
- `dealloc_buffer(Server, RefBuf) ->`
  [page 30] Deprecated function
- `dealloc_buffer(Server, RefBuf, Timeout) -> ok`
  [page 30] Deprecated function
- `read_buffer(Server, RefBuf) ->`
  [page 30] Deprecated function
- `read_buffer(Server, RefBuf, Timeout) -> {ok, {Value, LenInd}}`
  [page 30] Deprecated function

odbc

Erlang Module

As mention before is the interface of the Erlang *ODBC* application is divided into four parts:

- Start and Stop
  Starts and stops a gen_server process.
- Basic API
  Gives access to basic ODBC functions.
- Utility API
  Consists of functions that are easier to use than the Basic API. These functions are on a higher level, do more of the job, but allow less control to the application programmer.
- Depricated functions
  are only with for compatibility reason. Some of them do nothing and some are replaced by a function from Basic or Utility API.

All functions described are synchronous. The interface supports all ODBC defined SQL data types except binaries. They are all mapped on Erlang strings. The type `string()` is a `list()` of integers representing ASCII codes. The type `boolean()` is either the macro `?SQL_TRUE` or the macro `?SQL_FALSE`. The default Timeout for all functions is 5000 ms, unless otherwise stated. Most Erlang ODBC functions does have common arguments.

Start and Stop

Exports

```
start_link(Args, Options) ->
start_link(ServerName, Args, Options) -> Result
```

Types:

- Args = []
- Options = [Opt]
- Opt = This are options which are used by the gen_server module.
  For information see the module documentation `gen_server` and `sys`.
- ServerName = {local, atom()} | {global, atom()}
  When supplied, causes the server to be registered locally or globally. If the server is started without a name it can only be called using the returned pid.
- Result = {ok, pid()} | {error, Reason}
  The pid of the server or an error tuple.

- Reason = {already_started, pid()} | timeout | {no_c_node, Info}
  The server was already started, a timeout has expired, or the C node could not be started (the program may not have been found or may not have been executable e.g.).
- Info = string()
  More information.

Starts a new ODBC server process, registers it with the supervisor, and links it to the calling process. Opens a port to a new C node on the local host, using the same cookie as is used by the node of the calling process. Links to the process on the C node.

> **Note:**
> There is no default timeout value. Not using the timeout option is equivalent to having an infinite timeout value.
> An expired timeout is reported as an error here, not an exception.
> The debug options are described in the `sys` module documentation.

```
stop(Server) ->
stop(Server, Timeout) -> ok
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Timeout = integer() | infinity
  Max time (ms) for serving the request.

Stops the ODBC server process as soon as all already submitted requests have been processed. The C node is also stopped.

Basic API

To use the Basic API it is necessary to gain a comprehensive understanding of ODBC by studying [1].
Erlang ODBC application Basic API function allocate and deallocate memory automatic and therefore have the ODBC function which allocate or deallocate memory been excluded.

Exports

```
sqlBindColumn(Server, ColNum, Ref) ->
sqlBindColumn(Server, ColNum, Ref, Timeout) -> Result | {error, ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.

- ColNum = integer()
  Column number from left to right starting at 1.
- Ref = term()
  A reference.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO
- ErrMsg = string()
  Error message.
- ErrCode = ?SQL_INVALID_HANDLE | ?SQL_ERROR

Assigns a reference to the column with the number ColNum.

Differences from the ODBC Function:
The parameters `Server` and `Timeout` have been added. The input parameters `TargetType`, `TargetValuePtr`, `BufferLength`, and `StrLen_or_IndPtr` of the ODBC function have been replaced with the `Ref` parameter.

```
sqlCloseCursor(Server) ->
sqlCloseCursor(Server, Timeout) -> Result | {error, ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO
- ErrMsg = string()
  Error message.
- ErrCode = ?SQL_INVALID_HANDLE | ?SQL_ERROR

Closes a cursor that has been opened on a statement and discards pending results. See SQLCloseCursor in [1].

Differences from the ODBC Function:

The parameters `Server` and `Timeout` have been added.

```
sqlConnect(Server, DSN, UID, Auth) ->
sqlConnect(Server, DSN, UID, Auth, Timeout) -> Result | {error, ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- DSN = string()
  The name of the database.
- UID = string()
  The user ID
- Auth = string()
  The user's password for the database.

- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO
- ErrMsg -> string()
  Error message.
- ErrCode -> ?SQL_INVALID_HANDLE | ?SQL_ERROR

Establishes a connection to a driver and a data source. See SQLConnect in [1].
Differences from the ODBC Function:

Connection pooling is not supported. The parameters `Server` and `Timeout` have been added. The input parameters `NameLength1`, `NameLength2`, and `NameLength3` of the ODBC function have been excluded.

```
sqlDescribeCol(Server, ColNum) ->
sqlDescribeCol(Server, ColNum, Timeout) -> {Result, ColName, Nullable} | {error,
            ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- ColNum = integer()
  The column number from left to right, starting at 1.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO
- ColName = string()
  The column name.
- Nullable = ?SQL_NO_NULLS | ?SQL_NULLABLE | ?SQL_NULLABLE_UNKNOWN
  Indicates whether the column allows null values or not.
- ErrMsg -> string()
  Error message.
- ErrCode -> ?SQL_INVALID_HANDLE | ?SQL_ERROR

Returns the result descriptor – column name, and nullability for one column in the result set. See SQLDescribeCol in [1].
Differences from the ODBC Function:

The function does not support retrieval of bookmark column data. The parameters `Server` and `Timeout` have been added. The output parameters `ColumnName` and `NullablePtr` of the ODBC function have been changed into the returned values `ColName` and `Nullable`. The output parameters `BufferLength`, `NameLengthPtr`, `DataTypePtr`, `ColumnSizePtr`, and `DecimalDigitsPtr` of the ODBC function hsve been excluded.

```
sqlDisConnect(Server) ->
sqlDisConnect(Server, Timeout) -> Result | {error, ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO
- ErrMsg -> string()
  Error message.
- ErrCode -> ?SQL_INVALID_HANDLE | ?SQL_ERROR

Closes the connection associated with a specific server. See SQLDisconnect in [1].

Differences from the ODBC Function:

Connection pooling is not supported. The parameters `Server` and `Timeout` have been added.

```
sqlEndTran(Server, ComplType) ->
sqlEndTran(Server, ComplType, Timeout) -> Result | {error, ErrorMsg, errCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- ComplType = ?SQL_COMMIT | ?SQL_ROLLBACK
  Commit operation or rollback operation.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO
- ErrMsg -> string()
  Error message.
- ErrCode -> ?SQL_INVALID_HANDLE | ?SQL_ERROR

Requests a commit or rollback operation for all active operations on all statement handles associated with a connection. See SQLEndTran in [1].

> **Note:**
> Rollback of transactions may be unsupported by core level drivers.

Differences from the ODBC Function:

The parameter HandleType and Handle of the ODBC function has been excluded. The parameters `Server` and `Timeout` have been added.

```
sqlExecDirect(Server, Stmt) ->
sqlExecDirect(Server, Stmt, Timeout) -> Result | {error, ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Stmt = string()
  An SQL statement.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_NEED_DATA | ?SQL_NO_DATA
- ErrMsg -> string()
  Error message.
- ErrCode -> ?SQL_INVALID_HANDLE | ?SQL_ERROR

Executes a statement. See SQLExecDirect in [1].

Differences from the ODBC Function:

`?SQL_NO_DATA` is returned only in connection with positioned updates, which are not supported. The parameters `Server` and `Timeout` have been added. The input parameter StatementHandle and TextLength of the ODBC function has been excluded.

```
sqlFetch(Server) ->
sqlFetch(Server, Timeout) -> Result | {error, ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_NO_DATA
- ErrMsg -> string()
  Error message.
- ErrCode -> ?SQL_INVALID_HANDLE | ?SQL_ERROR

Fetches a row of data from a result set. The driver returns data for all columns that were bound to storage locations with `sqlBindCol/[3, 4]`. See SQLFetch in [1].

Differences from the ODBC Function:

The parameter StatementHandle of the ODBC function has been excluded. The parameters `Server` and `Timeout` have been added.

```
sqlNumResultCols(Server) ->
sqlNumResultCols(Server, Timeout) -> {Result, ColCount} | {error, ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.

- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO
- ColCount = integer()
  The number of columns in the result set.
- ErrMsg -> string()
  Error message.
- ErrCode -> ?SQL_INVALID_HANDLE | ?SQL_ERROR

Returns the number of columns in a result set. See SQLNumResultCols in [1].

Differences from the ODBC Function:

The parameter StatementHandle of the ODBC function has been excluded. The parameters `Server` and `Timeout` have been added. The output parameter `ColumnCountPtr` of the ODBC function has been changed into the returned value `ColCount`.

```
sqlRowCount(Server) ->
sqlRowCount(Server, Timeout) -> {Result, RowCount} | {error, ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO
  Result macro.
- RowCount = integer()
  The number of affected rows. If the number of affected rows is not available -1 is returned. For exceptions, see SQLRowCount in [1].
- ErrMsg -> string()
  Error message.
- ErrCode -> ?SQL_INVALID_HANDLE | ?SQL_ERROR

Returns the number of rows affected by an UPDATE, INSERT, or DELETE statement. See SQLRowCount in [1].

Differences from the ODBC Function:

The parameter StatementHandle has been excluded from ODBC function. The parameters `Server` and `Timeout` have been added. The output parameter `RowCountPtr` of the ODBC function has been changed into the returned value `RowCount`.

```
sqlSetConnectAttr(Server, Attr, Value) ->
sqlSetConnectAttr(Server, Attr, Value, Timeout) -> Result | {error, ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Attr = integer()
  One of the attributes described further down are supported. The attributes defined by ODBC are supplied through macros, but driver-specific attributes are not.

- Value = string() | integer()
  The new attribute value.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO
- ErrMsg -> string()
  Error message.
- ErrCode -> ?SQL_INVALID_HANDLE | ?SQL_ERROR

Sets attributes that govern aspects of connections. The following attributes, and their possible values, are supported (through macros):
?SQL_ATTR_AUTOCOMMIT
?SQL_ATTR_TRACE
?SQL_ATTR_TRACEFILE
These attributes can only be set after a connection. More information can be found under SQLSetConnectAttr in [1]. Driver-specific attributes are not supported through macros, but can be retrieved, if they are of character or signed/unsigned long integer types.

Differences from the ODBC Function:

Only character and signed/unsigned long integer attribute types are supported. The parameters `Server` and `Timeout` have been added. The input parameter `ConnectionHandle` and `StringLength` of the ODBC function has been excluded.

```
readData(Server, Ref) ->
readData(Server, Ref, Timeout) -> {ok, Value}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Ref
  A reference to the column.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Value = string()
  Contents of the column associated with `Ref`.

Returns the contents of a deferred data buffer and its associated length/indicator buffer. Used in connection with `sqlFetch/[1, 2]`.

```
columnRef() -> {ok, Ref}
```

Types:

- Ref
  A reference.

Returns a reference. The reference is assigned to a column in the function `sqlBindColumn/[3,4]`.

Utility API

Erlang ODBC application Utility API has a few easy-to-use function. The reported errors are tuples with arity 3.

Exports

```
erl_connect(Server, ConnectStr) ->
erl_connect(Server, ConnectStr, Timeout) ->
erl_connect(Server, DSN, UID, PWD) ->
erl_connect(Server, DSN, UID, PWD, Timeout) -> ok, | {error, ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- ConnectStr = string()
  Connection string. For syntax see SQLDriverConnect in [1].
- DSN = string()
  Name of the database.
- UID = string()
  User ID.
- PWD = string()
  Password.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- ErrMsg = string()
  Error message.
- ErrCode = ?SQL_INVALID | ?SQL_ERROR

Opens a connection to a database. There can be only one open connections to a database and per server. `connect/[2, 3]` is used when the information that can be supplied through `connect/[4, 5]` does not suffice.

> **Note:**
> The syntax to be used for `ConnectStr` is described under SQLDriverConnect in [1]. The `ConnectStr` must be complete.

```
erl_disconnect(Server) ->
erl_disconnect(Server, Timeout) -> ok | {error, ErrMsg, ErrCode}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.

- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- ErrMsg = string()
  Error message.
- ErrCode = ?SQL_INVALID_HANDLE | ?SQL_ERROR

Closes the connection to a database.

```
erl_executeStmt(Server, Stmt) ->
erl_executeStmt(Server, Stmt, Timeout) -> {updated, NRows} | {selected, [ColName],
              [Row]} | {error, ErrMsg}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Stmt = string()
  SQL statement to execute.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- NRows = integer()
  The number of updated rows for UPDATE, INSERT, or DELETE statements, or -1 if the number is not available. For other statement types the value is driver defined, see [1].
- ColName = string()
  The name of a column in the resulting table.
- Row = [Value]
  One row of the resulting table.
- Value = string() | null
  One value in a row.
- ErrMsg = string()
  Error message.

Executes a single SQL statement. All changes to the data source are, by default, automatically committed if successful.

> **Note:**
> {`updated, 0`} or {`updated, -1`} is returned when a statement that does not select or update any rows is successfully executed.
>
> The `ColNames` are ordered the same way as the `Values` in the `Rows` (the first `ColName` is associated with the first `Value` of each `Row` etc.). The `Rows` have no defined order since they represent a set.

Depricated functions

Depricated functions are only with for compatibility reason. Some of them do nothing and some are replaced by a function from above.

Exports

```
init_env(Server) ->
init_env(Server, Timeout) -> {ok, void}
```

This function is deprecated and does nothing. It is included only for compatibility reason.

```
connect(Server, RefEnvHandle, ConnectStr) ->
connect(Server, RefEnvHandle, ConnectStr, Timeout) ->
connect(Server, RefEnvHandle, DSN, UID, PWD) ->
connect(Server, RefEnvHandle, DSN, UID, PWD, Timeout) -> {ok, RefConnHandle} |
            {error, {Fcn, [Reason]}}
```

This function is deprecated and replaced by the function *erl_Connect/[2, 3, 4, 5]*.

```
execute_stmt(Server, RefConnHandle, Stmt) ->
execute_stmt(Server, RefConnHandle, Stmt, Timeout) -> {updated, NRows} | {selected,
            [ColName], [Row]} {error, {Fcn, [Reason]}}
```

This function is deprecated and replaced by the function *erl_executeStmt/[2, 3]*.

```
disconnect(Server, RefConnHandle) ->
disconnect(Server, RefConnHandle, Timeout) -> ok | {error, {Fcn, [Reason]}}
```

This function is deprecated and replaced by the function *erl_disconnect/[1, 2]*.

```
terminate_env(Server, RefEnvHandle) ->
terminate_env(Server, RefEnvHandle, Timeout) -> ok
```

This function is deprecated and does nothing. It is included only for compatibility reason.

```
 sql_alloc_handle(Server, HandleType, RefInputHandle) ->
sql_alloc_handle(Server, HandleType, RefInputHandle, Timeout) -> {0, void}
```

This function is deprecated and does nothing. It is included only for compatibility reason.

```
sql_bind_col(Server, RefStmtHandle, ColNum, RefBuf) ->
sql_bind_col(Server, RefStmtHandle, ColNum, RefBuf, Timeout) -> Result
```

This function is deprecated and replaced by the function *sqlBindColumn/[3,4]*.

```
sql_close_cursor(Server, RefStmtHandle) ->
sql_close_cursor(Server, RefStmtHandle, Timeout) -> Result
```

This function is deprecated and replaced by the function *sqlCloseCursor/[1, 2]*.

```
sql_connect(Server, RefConnHandle, DSN, UID, Auth) ->
sql_connect(Server, RefConnHandle, DSN, UID, Auth, Timeout) -> Result
```

This function is deprecated and replaced by the function *sqlConnect/[4, 5]*.

```
sql_describe_col(Server, RefStmtHandle, ColNum, BufLenColName) ->
sql_describe_col(Server, RefStmtHandle, ColNum, BufLenColName, Timeout) ->
            {Result,{ColName, LenColName}, SqlType, ColSize, DecDigs, Nullable}
```

This function is deprecated and replaced by the function *sqlDescribeCol/[2, 3]*.

```
sql_disconnect(Server, RefConnHandle) ->
sql_disconnect(Server, RefConnHandle, Timeout) -> Result
```

This function is deprecated and replaced by the function *sqlDisConnect/[1, 2]*.

```
sql_driver_connect(Server, RefConnHandle, InConnStr, BufLenOutConnStr, DrvCompletion)
            ->
sql_driver_connect(Server, RefConnHandle, InConnStr, BufLenOutConnStr, DrvCompletion,
            Timeout) -> {Result, {OutConnStr, LenOutConnStr}}
```

This function is deprecated and does nothing. It is included only for compatibility reason.

```
sql_end_tran(Server, HandleType, RefHandle, ComplType) ->
sql_end_tran(Server, HandleType, RefHandle, ComplType, Timeout) -> Result
```

This function is deprecated and replaced by the function *sqlEndTran/[2, 3]*.

```
sql_exec_direct(Server, RefStmtHandle, Stmt) ->
sql_exec_direct(Server, RefStmtHandle, Stmt, Timeout) -> Result
```

This function is deprecated and replaced by the function *sqlExecDirect/[1, 2]*.

```
sql_fetch(Server, RefStmtHandle) ->
sql_fetch(Server, RefStmtHandle, Timeout) -> Result
```

This function is deprecated and replaced by the function *sqlFetch/[1, 2]*.

```
sql_free_handle(Server, HandleType, RefHandle) ->
sql_free_handle(Server, HandleType, RefHandle, Timeout) -> Result
```

This function is deprecated and does nothing. It is included only for compatibility reason.

```
sql_get_connect_attr(Server, RefConnHandle, Attr, BufType) ->
sql_get_connect_attr(Server, RefConnHandle, Attr, BufType, Timeout) -> {Result, Value}
```

This function is deprecated and does nothing. It is included only for compatibility reason.

```
sql_get_diag_rec(Server, HandleType, RefHandle, RecNum, BufLenErrMsg) ->
sql_get_diag_rec(Server, HandleType, RefHandle, RecNum, BufLenErrMsg, Timeout) ->
            {Result, SqlState, NativeErr, {ErrMsg, LenErrMsg}}
```

This function is deprecated and does nothing. It is included only for compatibility reason.

```
sql_num_result_cols(Server, RefStmtHandle) ->
sql_num_result_cols(Server, RefStmtHandle, Timeout) -> {Result, ColCount}
```

This function is deprecated and replaced by the function *sqlNumResultCol/[1, 2]*.

```
sql_row_count(Server, RefStmtHandle) ->
sql_row_count(Server, RefStmtHandle, Timeout) -> {Result, RowCount}
```

This function is deprecated and replaced by the function *sqlRowCount/[1, 2]*.

```
sql_set_connect_attr(Server, RefConnHandle, Attr, Value, BufType) ->
sql_set_connect_attr(Server, RefConnHandle, Attr, Value, BufType, Timeout) -> Result
```

This function is deprecated and replaced by the function *sqlSetConnectAttr/[2, 3]*.

```
sql_set_env_attr(Server, RefEnvHandle, Attr, Value, BufType) ->
sql_set_env_attr(Server, RefEnvHandle, Attr, Value, BufType, Timeout) -> Result
```

This function is deprecated and does nothing. It is included only for compatibility reason.

```
alloc_buffer(Server, BufCType, Size) ->
alloc_buffer(Server, BufCType, Size, Timeout) -> {ok, RefBuf}
```

This function is deprecated and replaced by the function *columnRef/0*.

```
dealloc_buffer(Server, RefBuf) ->
dealloc_buffer(Server, RefBuf, Timeout) -> ok
```

This function is deprecated and does nothing. It is included only for compatibility reason.

```
read_buffer(Server, RefBuf) ->
read_buffer(Server, RefBuf, Timeout) -> {ok, {Value, LenInd}}
```

This function is deprecated and replaced by the function *readData/[2,3]*.

Error Messages and Exceptions

Errors caused by inability to contact the C node, allocate memory, or otherwise call ODBC functions cause exceptions. Exceptions are common to all functions. Errors caused by ODBC not being able to execute calls are reported through returned errors. These exceptions terminate the client only.

- {'EXIT', {badarg, F, ArgNo, Info}}
  The argument is of wrong type or out of range.
- {'EXIT', GenServerSpecificInfo}
  Error detected by `gen_server`.

These cause the ODBC server, and the C node, to terminate as well:

- {'EXIT', {timeout, Info}}
  Timeout expired.
- {'EXIT', {stopped, Reason}}
  The ODBC server died.

References

[1]: Microsoft ODBC 3.0, Programmer's Reference and SDK Guide

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.