Erlang ODBC application

version 1.0

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

Contents

Chapter 1

Erlang ODBC User's Guide

The *Erlang ODBC Application* provides an interface for accessing relational SQL-databases from Erlang.

1.1 Introduction

1.1.1 Purpose

The purpose of the Erlang ODBC application is to provide the programmer with an ODBC interface that has a Erlang/OTP touch and feel. So that the programmer may concentrate on solving his/her actual problem instead of struggling with pointers and memory allocation which is not very relevant for Erlang. This user guide will give you some information about technical issues and provide some examples of how to use the Erlang ODBC interface.

1.1.2 Pre-requisites

It is assumed that the reader is familiar with the Erlang programming language, concepts of OTP and has a basic understanding of relational databases and SQL.

1.1.3 About ODBC

Open Database Connectivity (ODBC) is a Microsoft standard for accessing relational databases that has become widely used. The ODBC standard provides a c-level application programming interface (API) for database access. It uses Structured Query Language (SQL) as its database access language.

1.1.4 About the Erlang ODBC application

Provides an Erlang interface to communicate with relational SQL-databases. It is built on top of Microsofts ODBC interface and therefore requires that you have an ODBC driver to the database that you want to connect to. The Erlang ODBC application is designed using the version 3.0 of the ODBC-standard, however using the option `{scrollable_cursors, off}` for a connection has been known to make it work for at least some 2.X drivers.

1.2 Getting started

1.2.1 Setting things up

As the Erlang ODBC application is dependent on third party products there are a few administrative things that needs to be done before you can get things up and running.

- The first thing you need to do, is to make sure you have an ODBC driver installed for the database that you want to access. Both the client machine where you plan to run your erlang node and the server machine running the database needs the the ODBC driver. (In some cases the client and the server may be the same machine).
- Secondly you might need to set environment variables and paths to appropriate values. This may differ a lot between different os's, databases and ODBC drivers. This is a configuration problem related to the third party product and hence we can not give you a standard solution in this guide.
- The Erlang ODBC application consists of both `Erlang` and `C` code. The `C` code is delivered as a precompiled executable for windows and solaris. If you for any reason need to recompile the `C` code or if you are running some other os and want to compile the code there is a include makefile to help you do so.

Note:
The Erlang ODBC application should run on all Unix dialects including Linux, Windows 2000, Windows XP and NT. But currently it is only tested for Solaris, Windows 2000, Windows XP and NT.

1.2.2 Compiling on Windows

On windows compilers are often distributed in some development environment such as `Visual C++`, that is what we use to compile the `C` code for windows.

If you need to compile the `C` code open a command prompt. Assume that Erlang/OTP is installed at "c:\Program Files\erl<erlang-version>". Change to the directory "c:\Program Files\erl<erlang-version>\lib\odbc-<odbc-version>\c_src" directory. Here you will find a Makefile. There are three variables in this makefile that you may want to override.

- ODBCLIBS - Path to the ODBC library.
- ODBCINCLUDE - Path to the ODBC header files.
- EIROOT - Path to the root directory of the Erlang application erl_interface.

An example of how the make command might look:

```
nmake EIROOT="..\..\..\lib\erl_interface-3.3.0"
```

1.2.3 Compiling on Unix

The prefered compiler is gcc version 2.7.2 or higher. Assume that the Erlang/OTP is installed in /usr/local/erlang then the `C` code is located in the /usr/local/erlang/lib/odbc-<odbc-version>/c_src directory. In the `C` code directory you will find a Makefile. There are three variables in this make file that you may want to override.

- ODBCROOT - Path to the root directory of the ODBC installation.
- ODBCLIBS - Path to the ODBC library.
- EIROOT - Root directory of the Erlang applicationerl_interface.

An example of how the make command might look:

```
gmake EIROOT="../../../lib/erl_interface-3.3.0"
```

1.2.4 Using the Erlang API

The following dialog within the Erlang shell illustrates the functionality of the Erlang ODBC interface. The table used in the example does not have any relevance to anything that exist in reality, it is just a simple example. The example was created using `sqlserver 7.0 with servicepack 1` as database and the ODBC driver for `sqlserver` with version `2000.80.194.00`.

```
1 > application:start(odbc).
     ok
```

Connect to the database

```
2 > {ok, Ref} = odbc:connect("DSN=sql-server;UID=aladin;PWD=sesame", []).
     {ok,<0.342.0>}
```

Create a table

```
3 > odbc:sql_query(Ref, "CREATE TABLE EMPLOYEE (NR integer,
     FIRSTNAME  char varying(20), LASTNAME  char varying(20), GENDER char(1),
     PRIMARY KEY(NR))").
     {updated,undefined}
```

Insert some data

```
4 > odbc:sql_query(Ref, "INSERT INTO EMPLOYEE VALUES(1, 'Jane', 'Doe', 'F')").
     {updated,1}

5 >odbc:sql_query(Ref, "INSERT INTO EMPLOYEE VALUES(2, 'John', 'Doe', 'M')").
     {updated,1}

6 > odbc:sql_query(Ref, "INSERT INTO EMPLOYEE VALUES(3, 'Monica', 'Geller', 'F')").
     {updated,1}

7 > odbc:sql_query(Ref, "INSERT INTO EMPLOYEE VALUES(4, 'Ross', 'Geller', 'M')").
     {updated,1}
```

```
8 > odbc:sql_query(Ref, "INSERT INTO EMPLOYEE VALUES(5, 'Rachel', 'Green', 'F')").
     {updated,1}

9 > odbc:sql_query(Ref, "INSERT INTO EMPLOYEE VALUES(6, 'Piper', 'Halliwell', 'F')").
     {updated,1}

10 > odbc:sql_query(Ref, "INSERT INTO EMPLOYEE VALUES(7, 'Prue', 'Halliwell', 'F')").
     {updated,1}

11 > odbc:sql_query(Ref, "INSERT INTO EMPLOYEE VALUES(8, 'Louise', 'Lane', 'F')").
     {updated,1}
```

Fetch all data in the table employee

```
12> odbc:sql_query(Ref, "SELECT * FROM EMPLOYEE").
   {selected,["NR","FIRSTNAME","LASTNAME","
         [[1,"Jane","Doe","F"],
          [2,"John","Doe","M"],
          [3,"Monica","Geller","F"],
          [4,"Ross","Geller","M"],
          [5,"Rachel","Green","F"],
          [6,"Piper","Halliwell","F"],
          [7,"Prue","Halliwell","F"],
          [8,"Louise","Lane","F"]]}
```

Associate a result set containg the whole table EMPLOYEE to the connection. The number of rows in the result set is returned.

```
13 > odbc:select_count(Ref, "SELECT * FROM EMPLOYEE").
     {ok,8}
```

Fetch certain parts of the result set.

```
14 >
     odbc:first(Ref).
     {selected,["NR","FIRSTNAME","LASTNAME","GENDER"],[[1,"Jane","Doe","F"]]}

15 > odbc:next(Ref).
     {selected,["NR","FIRSTNAME","LASTNAME","GENDER"],[[2,"John","Doe","M"]]}

16 > odbc:last(Ref).
     {selected,["NR","FIRSTNAME","LASTNAME","GENDER"],[[8,"Louise","Lane","F"]]}

17 > odbc:prev(Ref).
     {selected,["NR","FIRSTNAME","LASTNAME","GENDER"],[[7,"Prue","Halliwell","F"]]}
```

Fetch the fields FIRSTNAME and NR for all female employees

```
18 > odbc:sql_query(Ref, "SELECT FIRSTNAME, NR FROM EMPLOYEE WHERE GENDER = 'F'").
    {selected,["FIRSTNAME","NR"],
         [["Jane",1],
          ["Monica",3],
          ["Rachel",5],
          ["Piper",6],
          ["Prue",7],
          ["Louise",8]]}
```

Fetch the fields FIRSTNAME and NR for all female employees and sort them on the field FIRSTNAME .

```
19 > odbc:sql_query(Ref, "SELECT FIRSTNAME, NR FROM EMPLOYEE WHERE GENDER = 'F'
     ORDER BY FIRSTNAME").
   {selected,["FIRSTNAME","NR"],
         [["Jane",1],
          ["Louise",8],
          ["Monica",3],
          ["Piper",6],
          ["Prue",7],
          ["Rachel",5]
```

Associate a result set that contains the fields FIRSTNAME and NR for all female employees to the connection. The number of rows in the result set is returned.

```
20 > odbc:select_count(Ref, "SELECT FIRSTNAME, NR FROM EMPLOYEE WHERE GENDER = 'F'").
     {ok,6}
```

Fetch certain parts of the result set.

```
21 > odbc:select(Ref, {relative, 2}, 3).
     {selected,["FIRSTNAME","NR"],[["Monica",3],["Rachel",5],["Piper",6]]}

22 > odbc:select(Ref, next, 2).
     {selected,["FIRSTNAME","NR"],[["Prue",7],["Louise",8]]}

23 > odbc:select(Ref, {absolute, 1}, 2).
     {selected,["FIRSTNAME","NR"],[["Jane",1],["Monica",3]]}

24 > odbc:select(Ref, next, 2).
     {selected,["FIRSTNAME","NR"],[["Rachel",5],["Piper",6]]}

25 > odbc:select(Ref, {absolute, 1}, 4).
     {selected,["FIRSTNAME","NR"],
               [["Jane",1],["Monica",3],["Rachel",5],["Piper",6]]}
```

Delete the table EMPLOYEE

```
26 > odbc:sql_query(Ref, "DROP TABLE EMPLOYEE").
     {updated,undefined}
```

Shout down the connection.

```
27 > odbc:disconnect(Ref).
     ok
```

1.3 Databases

1.3.1 Databases

If you need to access a relational database such as `sqlserver`, `mysql`, `postgress`, `oracle`, `cybase` etc. from your erlang application using the Erlang ODBC interface is the way to go about it.

The Erlang ODBC application should work for any relational database that has an ODBC driver. But currently it is only tested for `sqlserver` and `oracle`.

1.3.2 Database independence

The Erlang ODBC interface is in principal database independent, e.i. an erlang program using the interface could be run without changes towards different databases. But as SQL is used it is alas possible to write database dependent programs. Even though SQL is an ANSI-standard meant to be database independent, different databases have proprietary extensions to SQL defining their own data types. If you keep to the ANSI data types you will minimize the problem. But unfortunately there is no guarantee that all databases actually treats the ANSI data types equivalently. For instance an installation of `Oracle Enterprise release 8.0.5.0.0 for unix` will accept that you create a table with the ANSI data type `integer`, but internally it will use an oracle data type. This alas is not transparent, so when retrieving the data it will have a different data type, in this case it will result in that the erlang user will get the string `"1"` instead of the value `1`.

Another obstacle is that some drivers do not support scrollable cursors which has the effect that the only way to traverse the result set is sequentially, with next, from the first row to the last, and once you pass a row you can not go back. This means that some functions in the interface will not work together with certain drivers. A similar problem is that not all drivers support “row count” for select queries, hence resulting in that the function `select_count/[3,4]` will return {`ok, undefined`} instead of {`ok, NrRows`} where `NrRows` is the number of rows in the result set.

1.3.3 Data types

The following is a list of the ANSI data types. For details turn to the ANSI standard documentation. Usage of other data types is of course possible, but you should be aware that this makes your application dependent on the database you are using at the moment.

- CHARACTER (size), CHAR (size)
- NUMERIC (precision, scale), DECIMAL (precision, scale), DEC (precision, scale) precision - total number of digits, scale - total number of decimal places
- INTEGER, INT, SMALLINT
- FLOAT (precision)
- REAL
- DOUBLE PRECISION
- CHARACTER VARYING(size), CHAR VARYING(size)

When inputing data the values will always be in string format as they are part of an SQL-query. Example:

```
odbc:sql_query(Ref, "INSERT INTO TEST VALUES(1, 2, 3)").
```

Note:
Note that when the value of the data to input is a string, it has to be quoted with '. Example:

```
odbc:sql_query(Ref, "INSERT INTO EMPLOYEE VALUES(1, 'Jane', 'Doe', 'F')").
```

When selecting data from a table, all data types are returned from the database to the ODBC driver as an ODBC data type. The tables below shows the mapping between those data types and what is returned by the Erlang API.

ODBC Data Type	Erlang Data Type
SQL_CHAR	String
SQL_NUMERIC	Float
SQL_DECIMAL	String
SQL_INTEGER	Integer
SQL_SMALLINT	Integer
SQL_FLOAT	Float
SQL_REAL	Float
SQL_DOUBLE	Float
SQL_VARCHAR	String

Table 1.1: Mapping of ODBC data types to the Erlang data types returned to the Erlang application.

ODBC Data Type	Erlang Data Type
SQL_TYPE_DATE	String
SQL_TYPE_TIME	String
SQL_TYPE_TIMESTAMP	String
SQL_LONGVARCHAR	String
SQL_BINARY	String
SQL_VARBINARY	String
SQL_LONGVARBINARY	String
SQL_BIGINT	String
SQL_TINYINT	Integer
SQL_BIT	Boolean

Table 1.2: Mapping of extended ODBC data types to the Erlang data types returned to the Erlang application.

1.4 Error handling

1.4.1 Strategy

On a conceptual level starting a database connection using the Erlang ODBC API is a basic client server application. The client process uses the API to start and communicate with the server process that manages the connection. The strategy of the Erlang ODBC application is that programming faults in the application itself will cause the connection process to terminate abnormally.(When a process terminates abnormally its supervisor will log relevant error reports.) Calls to API functions during or after termination of the connection process, will return {`error, connection_closed`}. Contextual errors on the other hand will not terminate the connection it will only return {`error, Reason`} to the client, where `Reason` may be any erlang term.

Clients

The connection is associated with the process that created it and can only be accessed through it. The reason for this is to preserve the semantics of result sets and transactions when select_count/[2,3] is called or auto_commit is turned off. Attempts to use the connection from another process will fail. This will not effect the connection. On the other hand, if the client process dies the connection will be terminated.

Timeouts

All request made by the client to the connection are synchronous. If the timeout is used and expires the client process will exit with reason timeout. Proably the right thing to do is let the client die and perhaps be restarted by its supervisor. But if the client chooses to catch this timeout, it is a good idea to wait a little while before trying again. If there are too many consecutive timeouts that are caught the connection process will conclude that there is something radically wrong and terminate the connection.

Gaurds

All API-functions are guarded and if you pass an argument of the wrong type a runtime error will occur. All input parameters to internal functions are trusted to be correct. It is a good programming practise to only distrust input from truly external sources. You are not supposed to catch these errors, it will only make the code very messy and much more complex, which introduces more bugs and in the worst case also covers up the actual faults. Put your effort on testing instead, you should trust your own input.

1.4.2 The whole picture

As the Erlang ODBC application relies on third party products and communicates with a database that proably runs on an other computer in the network there are plenty of things that might go wrong. To fully understand the things that might happen it facilitate to know the design of the Erlang ODBC application, hence here follows a short description of the current design.

Note:
Please note that design is something, that not necessarily will, but might change in future releases. While the semantics of the API will not change as it is independent of the implementation.

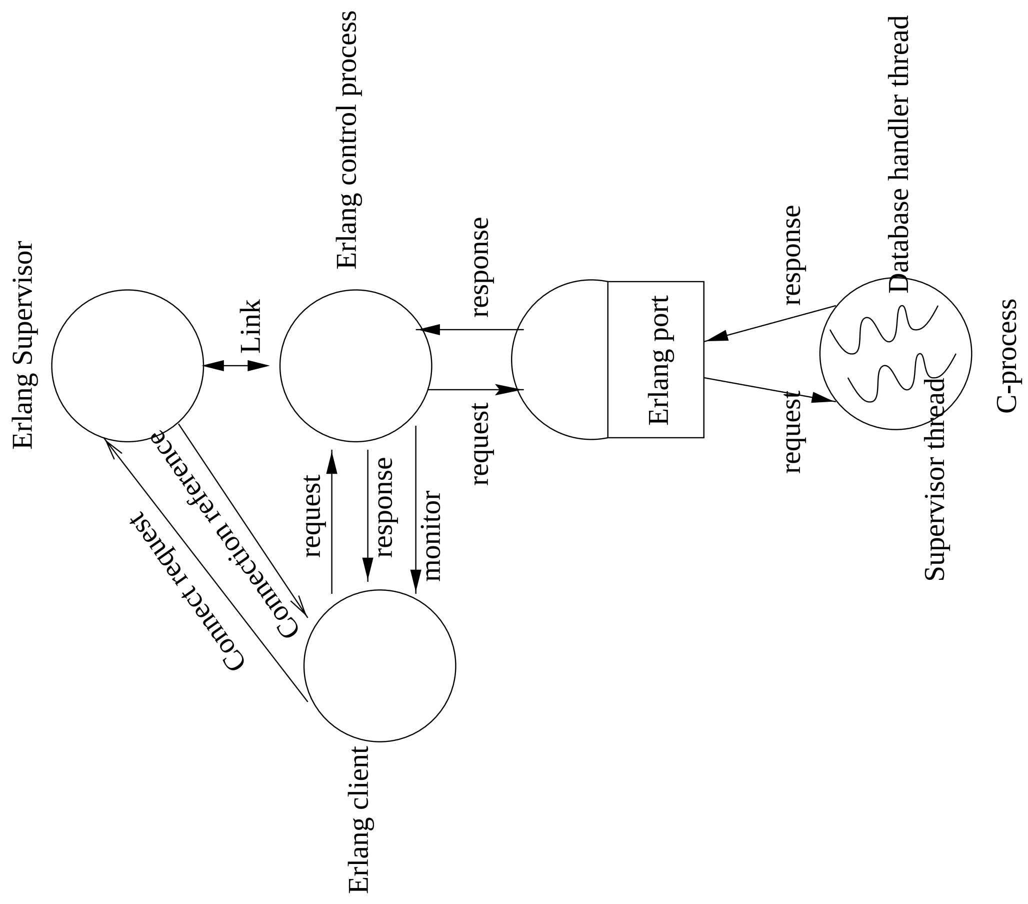

Figure 1.1: Architecture of the Erlang odbc application

When you do application:start(odbc) the only thing that happens is that a supervisor process is started. For each call to the API function connect/2 a process is spawned and added as a child to the Erlang ODBC supervisor. The supervisors only tasks are to provide error-log reports, if a child process should die abnormally, and the possibility to do a code change. Only the client process has the knowledge to decide if this connection managing process should be restarted.

The erlang connection process spawned by connect/2, will open a port to a c-process that handles the communication with the database through Microsoft's ODBC API. The C-process consists of two threads, the supervisor thread and the database handler thread. The supervisor thread checks for new messages on the erlang port while the database handler thread talks to the database. This has the effect that the process will detect if erlang closes the port when the database handler thread seems to hang on some database call. In this case the c-process will exit. If the c-process crashes/exits it will bring the erlang-process down too and vice versa i.e. the connection is terminated.

Note:
The function connect/2 will start the odbc application if that is not already done. In this case a supervisor information log will be produced stating that the odbc application was started as a temporary application. It is really the responsibility of the application that uses the API too make sure it is started in the desired way.

Error types

The types of errors that may occur can be divide into the following categories.

- Configuration problems - Everything from that the database was not set up right to that the c-program that should be run through the erlang port was not compiled for your platform.
- Errors discovered by the ODBC driver - If calls to the ODBC-driver fails due to circumstances that can not be controlled by the Erlang ODBC application programmer, an error string will be dug up from the driver. This string will be the `Reason` in the `{error, Reason}` return value. How good this error message is will of course be driver dependent. Examples of such circumstances are trying to insert the same key twice, invalid SQL-queries and that the database has gone off line.
- Connection termination - If a connection is terminated in an abnormal way, or if you try to use a connection that you have already terminated in a normal way by calling disconnect/1, the return value will be `{error, connection_closed}`. A connection could end abnormally because of an programming error in the Erlang ODBC application, but also if the ODBC driver crashes.
- Contextual errors - If API functions are used in the wrong context, the `Reason` in the error tuple will be a descriptive atom. For instance if you try to call the function `last/[1,2]` without first calling `select_count/[2,3]` to associate a result set with the connection. If the ODBC-driver does not support some functions, or if you disabled some functionality for a connection and then try to use it.

Erlang ODBC Reference Manual

Short Summaries

- Erlang Module **odbc** [page 17] – Erlang ODBC application
- Erlang Module **odbc (deprecated)** [page 25] – Deprecated version of the Erlang ODBC application

odbc

The following functions are exported:

- `commit(ConnectionReference, CommitMode) ->`
  [page 18] Commits or rollbacks a transaction.
- `commit(ConnectionReference, CommitMode, TimeOut) -> ok | {error, Reason}`
  [page 18] Commits or rollbacks a transaction.
- `connect(ConnectStr, Options) -> {ok, ConnectionReference} | {error, Reason}`
  [page 18] Opens a connection to the database.
- `disconnect(ConnectionReference) -> ok | {error, Reason}`
  [page 19] Closes a connection to a database.
- `first(ConnectionReference) ->`
  [page 19] Returns the first row of the result set and positions a cursor at this row.
- `first(ConnectionReference, Timeout) -> {selected, ColNames, Rows} | {error, Reason}`
  [page 19] Returns the first row of the result set and positions a cursor at this row.
- `last(ConnectionReference) ->`
  [page 20] Returns the last row of the result set and positions a cursor at this row.
- `last(ConnectionReference, TimeOut) -> {selected, ColNames, Rows} | {error, Reason}`
  [page 20] Returns the last row of the result set and positions a cursor at this row.
- `next(ConnectionReference) ->`
  [page 20] Returns the next row of the result set relative the current cursor position and positions the cursor at this row.
- `next(ConnectionReference, TimeOut) -> {selected, ColNames, Rows} | {error, Reason}`
  [page 20] Returns the next row of the result set relative the current cursor position and positions the cursor at this row.

- `prev(ConnectionReference) ->`
  [page 20] Returns the previous row of the result set relative the current cursor position and positions the cursor at this row.
- `prev(ConnectionReference, TimeOut) -> {selected, ColNames, Rows} | {error, Reason}`
  [page 20] Returns the previous row of the result set relative the current cursor position and positions the cursor at this row.
- `sql_query(ConnectionReference, SQLQuery) ->`
  [page 20] Executes a SQL query. If it is a SELECT query the result set is returned, on the format `{selected, ColNames, Rows}`. For other query types the tuple `{updated, NRows}` is returned.
- `sql_query(ConnectionReference, SQLQuery, TimeOut) -> {updated, NRows} | {selected, ColNames, Rows} | {error, Reason}`
  [page 20] Executes a SQL query. If it is a SELECT query the result set is returned, on the format `{selected, ColNames, Rows}`. For other query types the tuple `{updated, NRows}` is returned.
- `select_count(ConnectionReference, SelectQuery) ->`
  [page 21] Executes a SQL SELECT query and associates the result set with the connection. A cursor is positioned before the first row in the result set and the tuple `{ok, NrRows}` is returned.
- `select_count(ConnectionReference, SelectQuery, TimeOut) -> {ok, NrRows} | {error, Reason}`
  [page 21] Executes a SQL SELECT query and associates the result set with the connection. A cursor is positioned before the first row in the result set and the tuple `{ok, NrRows}` is returned.
- `select(ConnectionReference, Position, N) ->`
  [page 21] Selects `N` consecutive rows of the result set.
- `select(ConnectionReference, Position, N, TimeOut) -> {selected, ColNames, Rows} | {error, Reason}`
  [page 21] Selects `N` consecutive rows of the result set.
- `start_link(Args, Options) ->`
  [page 22] Depricated function
- `start_link(ServerName, Args, Options) -> Result`
  [page 22] Depricated function
- `stop(Server) ->`
  [page 22] Depricated function
- `stop(Server, Timeout) -> ok`
  [page 22] Depricated function
- `sqlConnect(Server, DSN, UID, Auth) ->`
  [page 22] Depricated function
- `sqlConnect(Server, DSN, UID, Auth, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 22] Depricated function
- `erl_connect(Server, ConnectStr) ->`
  [page 22] Depricated function
- `erl_connect(Server, ConnectStr, Timeout) ->`
  [page 22] Depricated function
- `erl_connect(Server, DSN, UID, PWD) ->`
  [page 22] Depricated function

- `erl_connect(Server, DSN, UID, PWD, Timeout) -> ok, | {error, ErrMsg, ErrCode}`
  [page 22] Depricated function
- `sqlDisConnect(Server) ->`
  [page 22] Depricated function
- `sqlDisConnect(Server, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 22] Depricated function
- `erl_disconnect(Server) ->`
  [page 22] Depricated function
- `erl_disconnect(Server, Timeout) -> ok | {error, ErrMsg, ErrCode}`
  [page 23] Depricated function
- `sqlSetConnectAttr(Server, Attr, Value) ->`
  [page 23] Depricated function
- `sqlSetConnectAttr(Server, Attr, Value, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 23] Depricated function
- `erl_executeStmt(Server, Stmt) ->`
  [page 23] Depricated function
- `erl_executeStmt(Server, Stmt, Timeout) -> {updated, NRows} | {selected, ColNames, Rows} | {error, ErrFunc, ErrMsg}`
  [page 23] Depricated function
- `sqlEndTran(Server, ComplType) ->`
  [page 23] Depricated function
- `sqlEndTran(Server, ComplType, Timeout) -> Result | {error, ErrorMsg, errCode}`
  [page 23] Depricated function
- `sqlRowCount(Server) ->`
  [page 23] Depricated function
- `sqlRowCount(Server, Timeout) -> {Result, RowCount} | {error, ErrMsg, ErrCode}`
  [page 23] Depricated function
- `sqlDescribeCol(Server, ColNum) ->`
  [page 23] Depricated function
- `sqlDescribeCol(Server, ColNum, Timeout) -> {Result, ColName, Nullable} | {error, ErrMsg, ErrCode}`
  [page 23] Depricated function
- `sqlNumResultCols(Server) ->`
  [page 23] Depricated function
- `sqlNumResultCols(Server, Timeout) -> {Result, ColCount} | {error, ErrMsg, ErrCode}`
  [page 23] Depricated function
- `sqlCloseHandle(Server) ->`
  [page 23] Depricated function
- `sqlCloseHandle(Server, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 23] Depricated function
- `sqlExecDirect(Server, Stmt) ->`
  [page 23] Depricated functions

- `sqlExecDirect(Server, Stmt, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 23] Depricated functions
- `columnRef() ->{ok, Ref}`
  [page 23] Depricated functions
- `sqlBindColumn(Server, ColNum, Ref) ->`
  [page 23] Depricated functions
- `sqlBindColumn(Server, ColNum, Ref, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 23] Depricated functions
- `sqlFetch(Server) ->`
  [page 23] Depricated functions
- `sqlFetch(Server, Timeout) ->`
  [page 23] Depricated functions
- `readData(Server, Ref) ->`
  [page 23] Depricated functions
- `readData(Server, Ref, Timeout) -> {ok, Value}`
  [page 23] Depricated functions

odbc (deprecated)

The following functions are exported:

- `start_link(Args, Options) ->`
  [page 26] Start a new ODBC server process.
- `start_link(ServerName, Args, Options) -> Result`
  [page 26] Start a new ODBC server process.
- `stop(Server) ->`
  [page 26] Stop the ODBC server process
- `stop(Server, Timeout) -> ok`
  [page 26] Stop the ODBC server process
- `sqlBindColumn(Server, ColNum, Ref) ->`
  [page 27] Assign a reference to a column in a result set
- `sqlBindColumn(Server, ColNum, Ref, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 27] Assign a reference to a column in a result set
- `sqlCloseCursor(Server) ->`
  [page 27] Close a cursor that has been opened on a statement and discards pending results
- `sqlCloseCursor(Server, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 27] Close a cursor that has been opened on a statement and discards pending results
- `sqlConnect(Server, DSN, UID, Auth) ->`
  [page 28] Establishes a connection to a database
- `sqlConnect(Server, DSN, UID, Auth, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 28] Establishes a connection to a database

- `sqlDescribeCol(Server, ColNum) ->`
  [page 28] Return the result descriptor
- `sqlDescribeCol(Server, ColNum, Timeout) -> {Result, ColName, Nullable} | {error, ErrMsg, ErrCode}`
  [page 28] Return the result descriptor
- `sqlDisConnect(Server) ->`
  [page 29] Close the connection associated with the Server
- `sqlDisConnect(Server, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 29] Close the connection associated with the Server
- `sqlEndTran(Server, ComplType) ->`
  [page 29] 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 29] Request a commit or rollback operation for all active operations on all statement handles associated with a connection
- `sqlExecDirect(Server, Stmt) ->`
  [page 30] Execute a statement
- `sqlExecDirect(Server, Stmt, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 30] Execute a statement
- `sqlFetch(Server) ->`
  [page 30] Fetch a row of data from a result set
- `sqlFetch(Server, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 30] Fetch a row of data from a result set
- `sqlNumResultCols(Server) ->`
  [page 31] Return the number of columns in a result set
- `sqlNumResultCols(Server, Timeout) -> {Result, ColCount} | {error, ErrMsg, ErrCode}`
  [page 31] Return the number of columns in a result set
- `sqlRowCount(Server) ->`
  [page 31] Returns the number of rows affected by an UPDATE, INSERT, or DELETE statement
- `sqlRowCount(Server, Timeout) -> {Result, RowCount} | {error, ErrMsg, ErrCode}`
  [page 31] Returns the number of rows affected by an UPDATE, INSERT, or DELETE statement
- `sqlSetConnectAttr(Server, Attr, Value) ->`
  [page 32] set Connection Attribute
- `sqlSetConnectAttr(Server, Attr, Value, Timeout) -> Result | {error, ErrMsg, ErrCode}`
  [page 32] set Connection Attribute
- `readData(Server, Ref) ->`
  [page 32] Get contents of the associated column.
- `readData(Server, Ref, Timeout) -> {ok, Value}`
  [page 32] Get contents of the associated column.
- `columnRef() -> {ok, Ref}`
  [page 33] Return a reference.

- `erl_connect(Server, ConnectStr) ->`
  [page 33] Open a connection to a database
- `erl_connect(Server, ConnectStr, Timeout) ->`
  [page 33] Open a connection to a database
- `erl_connect(Server, DSN, UID, PWD) ->`
  [page 33] Open a connection to a database
- `erl_connect(Server, DSN, UID, PWD, Timeout) -> ok, | {error, ErrMsg, ErrCode}`
  [page 33] Open a connection to a database
- `erl_disconnect(Server) ->`
  [page 34] Close the connection to a database
- `erl_disconnect(Server, Timeout) -> ok | {error, ErrMsg, ErrCode}`
  [page 34] Close the connection to a database
- `erl_executeStmt(Server, Stmt) ->`
  [page 34] Execute a single SQL statement
- `erl_executeStmt(Server, Stmt, Timeout) -> {updated, NRows} | {selected, [ColName], [Row]} | {error, ErrMsg}`
  [page 34] Execute a single SQL statement

odbc

Erlang Module

This application provides an Erlang interface to communicate with relational SQL-databases. It is built on top of Microsofts ODBC interface and therefore requires that you have an ODBC driver to the database that you want to connect to.

Note:
The functions `first/[1,2]`, `last/[1,2]`, `next/[1,2]`, `prev[1,2]` and `select/[3,4]` assumes there is a result set associated with the connection to work on. Calling the function `select_count/[2,3]` associates such a result set with the connection, calling the function `sql_query/[2,3]` will remove the association. Calling select_count again will remove the current result set association and create a new one. Alas some drivers only support sequential traversal of the result set, e.i. they do not support what in the ODBC world is known as scrollable cursors. This will have the effect that functions such as `first/[1,2]`, `last/[1,2]`, `prev[1,2]`, etc may return {`error, driver_does_not_support_function`}

COMMON DATA TYPES

Here follows type definitions that are used by more than one function in the ODBC API.

Note:
The type `TimeOut` has the default value `infinity`, so for instance: commit(ConnectionReference, CommitMode) is the same as commit(ConnectionReference, CommitMode, infinity). If the timeout expires the client will exit with the reason timeout.

```
ConnectionReference - as returned by connect/2

TimeOut = Milliseconds | infinity

Milliseconds = integer()

CommonReason = connection_closed | term() - some kind of explanation of what went wro

String = list of ASCII characters

ColName = String - Name of column in the result set

ColNames -  [ColName] - e.g. a list of the names of the
     selected columns in the result set.
```

```
Row = [Value] | {Value} - List or tuple of column values
     e.g. one row of the result set. (Configurable per connection.)
```

```
Value = null | term() - A column value.
```

```
Rows = [Row] - A list of rows from the result set.
```

ERROR HANDLING

The error handling strategy and possible errors sources are described in the Erlang ODBC User's Guide. [page 8]

Exports

```
commit(ConnectionReference, CommitMode) ->
commit(ConnectionReference, CommitMode, TimeOut) -> ok | {error, Reason}
```

Types:

- CommitMode = commit | rollback
- Reason = not_an_explicit_commit_connection | process_not_owner_of_odbc_connection | CommonReason
- See also common data types.

Commits or rollbacks a transaction. Needed on connections where automatic commit is turned off.

```
connect(ConnectStr, Options) -> {ok, ConnectionReference} | {error, Reason}
```

Types:

- ConnectionReference - should be used to acess the connection.
- ConnectStr
  An example of a connection string: `"DSN=sql-server;UID=alladin;PWD=sesame"` where DSN is your ODBC Data Source Name, UID is a database user id and PWD is the password for that user. These are usually the attributes required in the connection string, but some drivers have other driver specific attributes, for example `"DSN=Oracle8;DBQ=gandalf;UID=alladin;PWD=sesame"` where DBQ is your TNSNAMES.ORA entry name e.g. some Oracle specific configuration attribute.
- Options = [] | [Option]
  All options has default values.
- Option = {auto_commit, AutoCommitMode} | {timeout, Milliseconds} | {tuple_row, TupleMode} | {scrollable_cursors, UseSrollableCursors} | {trace_driver, TraceMode}
  The default timeout is infinity
- AutoCommitMode = on | off
  Default is on.
- TupleMode = on | off
  Default is off.
- UseSrollableCursors = on | off
  Default is on.

- TraceMode = on | off
  Default is off.
- Reason = port_program_executable_not_found | CommonReason
- See also common data types.

Opens a connection to the database. The connection is associated with the process that created it and can only be accessed through it. This funtion may spawn new processes to handle the connection. These processes will terminate if the process that created the connection dies or if you call disconnect/1.

If automatic commit mode is turned on, each query will be considered as an individual transaction and will be automaticly commited after it has been executed. If you want more than one query to be part of the same transaction the automatic commit mode should be turned off. Then you will have to call commit/3 explicitly to end a transaction.

As default a result set row is returned as a list of values but a connection can be configured to return a result set row as a tuple of values.

Scrollable cursors are nice but causes some overhead. For some connections speed might be more important than flexible data access and then you can disable scrollable cursor for a connection, limiting the API but gaining speed

If trace mode is turned on this tells the ODBC driver to write a trace log to the file SQL.LOG that is placed in the current directory of the erlang emulator. This information may be useful if you suspect there might be a bug in the erlang ODBC application, and it might be relevant for you to send this file to our support. Otherwise you will probably not have much use of this.

Note:
For more information about the `ConnectStr` see description of the function SQLDriverConnect in [1].

`disconnect(ConnectionReference) -> ok | {error, Reason}`

Types:

- Reason = process_not_owner_of_odbc_connection
- See common data types.

Closes a connection to a database. This will also terminate all processes that may have been spawned when the connection was opened. This call will always succeed. If the connection can not be disconnected gracefully it will be brutally killed. However you may receive an error message as result if you try to disconnect a connection started by another process.

`first(ConnectionReference) ->`
`first(ConnectionReference, Timeout) -> {selected, ColNames, Rows} | {error, Reason}`

Types:

- Reason = result_set_does_not_exist | driver_does_not_support_function | scrollable_cursors_disabled | process_not_owner_of_odbc_connection | CommonReason
- See common data types.

Returns the first row of the result set and positions a cursor at this row.

`last(ConnectionReference) ->`
`last(ConnectionReference, TimeOut) -> {selected, ColNames, Rows} | {error, Reason}`

Types:

- Reason = result_set_does_not_exist | driver_does_not_support_function | scrollable_cursors_disabled | process_not_owner_of_odbc_connection | CommonReason
- See common data types.

Returns the last row of the result set and positions a cursor at this row.

`next(ConnectionReference) ->`
`next(ConnectionReference, TimeOut) -> {selected, ColNames, Rows} | {error, Reason}`

Types:

- Reason = result_set_does_not_exist | process_not_owner_of_odbc_connection | CommonReason
- See common data types.

Returns the next row of the result set relative the current cursor position and positions the cursor at this row. If the cursor is positioned at the last row of the result set when this function is called the returned value will be `{selected, ColNames,[]}` e.i. the list of row values is empty indicating that there is no more data to fetch.

`prev(ConnectionReference) ->`
`prev(ConnectionReference, TimeOut) -> {selected, ColNames, Rows} | {error, Reason}`

Types:

- Reason = result_set_does_not_exist | driver_does_not_support_function | scrollable_cursors_disabled | process_not_owner_of_odbc_connection | CommonReason
- See common data types.

Returns the previous row of the result set relative the current cursor position and positions the cursor at this row.

`sql_query(ConnectionReference, SQLQuery) ->`
`sql_query(ConnectionReference, SQLQuery, TimeOut) -> {updated, NRows} | {selected, ColNames, Rows} | {error, Reason}`

Types:

- SQLQuery = String
  SQL query.
- NRows = integer()
  The number of affected rows for UPDATE, INSERT, or DELETE queries. For other query types the value is driver defined, and hence should be ignored.
- Reason = process_not_owner_of_odbc_connection | CommonReason
- See also common data types.

Executes a SQL query. If it is a SELECT query the result set is returned, on the format {`selected, ColNames, Rows`}. For other query types the tuple {`updated, NRows`} is returned.

Note:

Some drivers may not have the information of the number of affected rows available and then the return value may be {`updated, undefined`} .

The list of colum names is ordered in the same way as the list of values of a row, e.g. the first `ColName` is associated with the first `Value` in a `Row`.

`select_count(ConnectionReference, SelectQuery) ->`

`select_count(ConnectionReference, SelectQuery, TimeOut) -> {ok, NrRows} | {error, Reason}`

Types:

- SelectQuery = String
  SQL SELECT query.
- NrRows = integer() | undefined
  Number of row in the result set.
- Reason = process_not_owner_of_odbc_connection | CommonReason
- See also common data types.

Executes a SQL SELECT query and associates the result set with the connection. A cursor is positioned before the first row in the result set and the tuple {`ok, NrRows`} is returned.

Note:

Some drivers may not have the information of the number of rows in the result set, then `NrRows` will have the value `undefined`.

`select(ConnectionReference, Position, N) ->`

`select(ConnectionReference, Position, N, TimeOut) -> {selected, ColNames, Rows} | {error, Reason}`

Types:

- Position = next | {relative, Pos} | {absolute, Pos}
  Selection strategy, determines at which row in the result set to start the selection.
- Pos = integer()
  Should indicate a row number in the result set. When used together with the option `relative` it will be used as an offset from the current cursor position, when used together with the option `absolute` it will be interpreted as a row number.
- Reason = result_set_does_not_exist | driver_does_not_support_function | scrollable_cursors_disabled | process_not_owner_of_odbc_connection | CommonReason
- See also common data types.

Selects `N` consecutive rows of the result set. If `Position` is `next` it is semanticly equivalent of calling `next/[1,2]` `N` times. If `Position` is `{relative, Pos}`, `Pos` will be used as an offset from the current cursor position to determine the first selected row. If `Position` is `{absolute, Pos}`, `Pos` will be the number of the first row selected. After this function has returned the cursor is positioned at the last selected row. If there is less then `N` rows left of the result set the length of `Rows` will be less than `N`. If the first row to select happens to be beyond the last row of the result set, the returned value will be `{selected, ColNames,[]}` e.i. the list of row values is empty indicating that there is no more data to fetch.

DEPRECATED FUNCTIONS

The following functions are deprecated and will disappear in the next release. They are only kept to maintain a temporary backward compatibility to make the transition to the new interface less abrupt. However you should not mix the use of the old and the new interface. This may lead to an unexpected behavior. The deprecated interface also requires the application programmer to use the include file odbc.hrl, this whole file is now deprecated. Here follows a short description of the deprecated interface, for details see odbc (deprecated) documentation. [page 25]

Exports

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

Functionality provided by connect/2.

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

Functionality provided by disconnect/2.

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

Use connect/2 instead.

`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}`

Use connect/2 instead.

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

Use disconnect/1 instead.

`erl_disconnect(Server) ->`

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

Use disconnect/1 instead.

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

Functionality provided by connect/2.

`erl_executeStmt(Server, Stmt) ->`
`erl_executeStmt(Server, Stmt, Timeout) -> {updated, NRows} | {selected, ColNames, Rows} | {error, ErrFunc, ErrMsg}`

Use sql_query/[2,3] instead.

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

Use commmit/[2,3] instead.

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

Functionality provided by sql_query/[2,3] and select_count/[2,3]

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

Not needed.

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

Not needed.

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

Not needed.

`sqlExecDirect(Server, Stmt) ->`
`sqlExecDirect(Server, Stmt, Timeout) -> Result | {error, ErrMsg, ErrCode}`
`columnRef() ->{ok, Ref}`
`sqlBindColumn(Server, ColNum, Ref) ->`
`sqlBindColumn(Server, ColNum, Ref, Timeout) -> Result | {error, ErrMsg, ErrCode}`
`sqlFetch(Server) ->`
`sqlFetch(Server, Timeout) ->`
`readData(Server, Ref) ->`
`readData(Server, Ref, Timeout) -> {ok, Value}`

The semantical benefit of using the above functions can be achived much easier by using select_count/[2,3] followed by next/[1,2].

REFERENCES

[1]: Microsoft ODBC 3.0, Programmer's Reference and SDK Guide
See also http://msdn.microsoft.com/

Deprecated odbc

Erlang Module

The Erlang ODBC interface is divided into three parts:

- Start and Stop
  Start and stop the Erlang ODBC gen_server process.
- Basic API
  Consist of most 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.

General information

Erlang ODBC is an Erlang application. An Erlang application allows noramlly code change in a running Erlang system. Erlang ODBC application does not allow code upgrade in a running Erlang system.
Erlang ODBC functions are synchronous.
Erlang ODBC functions supports all ODBC defined SQL data types. SQL data types are mapped to nearest Erlang type, except binaries which are mapped into string(). The type `string()` is a `list()` of integers representing ASCII codes.
When a column in a row has no value then columns is called to be null or contain a null. The null is mapped to a Erlang atom null.
A SQL question in Erlang is a string. The string is decoded in the Erlang ODBC C-program as a string. If the SQL questions contains a null character "\0" then the decode functions think that the string ends there. Do not use null characters in the SQL question.
In some databases has data in string format been filled out with space until column length.
The default Timeout for all functions is 5000 ms, unless otherwise stated.

Error handle

Erlang ODBC application may fail for following reasons:
Bad arguments to an Erlang ODBC function.
Failure from ODBC-driver.
Bad arguments to an Erlang ODBC function returns the tuple {`'EXIT',Reason`}.
When an Erlang ODBC function receive faliure from ODBC driver, the functions returns the tuple {`error, ErrMsg, ErrCode`} .

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.

List of Figures

List of Tables

Index of Modules and Functions

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