


# Erlang ODBC

Copyright © 1999-2025 Ericsson AB. All Rights Reserved.
Erlang ODBC 2.13.3.1
December 10, 2025

**Copyright © 1999-2025 Ericsson AB. All Rights Reserved.**
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

**December 10, 2025**

# 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 Prerequisites

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 cannot 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, solaris and linux (SLES10) in the commercial build. In the open source distribution it is built the same way as all other application using configure and make. You may want to provide the the path to your ODBC libraries using --with-odbc=PATH.

**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 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 > odbc:start().
     ok
```

Connect to the database

```
2 > {ok, Ref} = odbc:connect("DSN=sql-server;UID=aladdin;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}
```

Check what data types the database assigned for the columns. Hopefully this is not a surprise, some times it can be! These are the data types that you should use if you want to do a parameterized query.

```
5 > odbc:describe_table(Ref, "EMPLOYEE").
     {ok, [{"NR", sql_integer},
           {"FIRSTNAME", {sql_varchar, 20}},
           {"LASTNAME", {sql_varchar, 20}}
           {"GENDER", {sql_char, 1}}]}
```

Use a parameterized query to insert many rows in one go.

```
6 > odbc:param_query(Ref,"INSERT INTO EMPLOYEE (NR, FIRSTNAME, "
                   "LASTNAME, GENDER) VALUES(?, ?, ?, ?)",
                    [{sql_integer,[2,3,4,5,6,7,8]},
                     {{sql_varchar, 20},
                              ["John", "Monica", "Ross", "Rachel",
                              "Piper", "Prue", "Louise"]},
                    {{sql_varchar, 20},
                              ["Doe","Geller","Geller", "Green",
                               "Halliwell", "Halliwell", "Lane"]},
                    {{sql_char, 1}, ["M","F","M","F","F","F","F"]}]).
     {updated, 7}
```

Fetch all data in the table employee

```
7> odbc:sql_query(Ref, "SELECT * FROM EMPLOYEE").
   {selected,["NR","FIRSTNAME","LASTNAME","GENDER"],
         [{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 containing the whole table EMPLOYEE to the connection. The number of rows in the result set is returned.

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

You can always traverse the result set sequential by using next

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

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

If your driver supports scrollable cursors you have a little more freedom, and can do things like this.

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

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

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

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

Fetch the fields FIRSTNAME and NR for all female employees

```
15 > 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 .

```
16 > 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.

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

A few more ways of retrieving parts of the result set when the driver supports scrollable cursors. Note that next will work even without support for scrollable cursors.

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

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

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

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

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

Select, using a parameterized query.

```
23 > odbc:param_query(Ref, "SELECT * FROM EMPLOYEE WHERE GENDER=?",
     [{{sql_char, 1}, ["M"]}]).
     {selected,["NR","FIRSTNAME","LASTNAME","GENDER"],
               [{2,"John", "Doe", "M"},{4,"Ross","Geller","M"}]}
```

Delete the table `EMPLOYEE`.

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

Shut down the connection.

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

Shut down the application.

```
26 > odbc:stop().
   =INFO REPORT==== 7-Jan-2004::17:00:59 ===
   application: odbc
   exited: stopped
   type: temporary

   ok
```

# 1.3 Databases

## 1.3.1 Databases

If you need to access a relational database such as `sqlserver`, `mysql`, `postgres`, `oracle`, `cybase` etc. from your erlang application using the Erlang ODBC interface is a good 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 regularly tested for `sqlserver` and `postgres`.

## 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 column with the ANSI data type `integer`, but when retrieving values from this column the driver reports that it is of type `SQL_DECIMAL(0, 38)` and not `SQL_INTEGER` as you may have expected.

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 cannot 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 inputting data using sql_query/[2,3] 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')").
```

You may also input data using param_query/[3,4] and then the input data will have the Erlang type corresponding to the ODBC type of the column.See ODBC to Erlang mapping

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(size) | String \| Binary (configurable) |
| SQL_WCHAR(size) | Unicode binary encoded as UTF16 little endian. |
| SQL_NUMERIC(p,s)<br>when (p >= 0 and p <= 9 and s == 0) | Integer |
| SQL_NUMERIC(p,s)<br>when (p >= 10 and p <= 15 and s == 0) or (s <= 15 and s > 0) | Float |
| SQL_NUMERIC(p,s)<br>when p >= 16 | String |
| SQL_DECIMAL(p,s)<br>when (p >= 0 and p <= 9 and s == 0) | Integer |
| SQL_DECIMAL(p,s)<br>when (p >= 10 and p <= 15 and s == 0) or (s <= 15 and s > 0) | Float |
| SQL_DECIMAL(p,s)<br>when p >= 16 | String |
| SQL_INTEGER | Integer |
| SQL_SMALLINT | Integer |
| SQL_FLOAT | Float |
| SQL_REAL | Float |
| SQL_DOUBLE | Float |
| SQL_VARCHAR(size) | String \| Binary (configurable) |
| SQL_WVARCHAR(size) | Unicode binary encoded as UTF16 little endian. |

Table 3.1: 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 | {{YY, MM, DD}, {HH, MM, SS}} |
| SQL_LONGVARCHAR | String \| Binary (configurable) |
| SQL_WLONGVARCHAR(size) | Unicode binary encoded as UTF16 little endian. |

| | |
|---|---|
| SQL_BINARY | String \| Binary (configurable) |
| SQL_VARBINARY | String \| Binary (configurable) |
| SQL_LONGVARBINARY | String \| Binary (configurable) |
| SQL_TINYINT | Integer |
| SQL_BIT | Boolean |

Table 3.2: Mapping of extended ODBC data types to the Erlang data types returned to the Erlang application.

**Note:**

To find out which data types will be returned for the columns in a table use the function describe_table/[2,3]

### 1.3.4 Batch handling

Grouping of SQL queries can be desirable in order to reduce network traffic. Another benefit can be that the data source sometimes can optimize execution of a batch of SQL queries.

Explicit batches an procedures described below will result in multiple results being returned from sql_query/[2,3]. while with parameterized queries only one result will be returned from param_query/[2,3].

#### Explicit batches

The most basic form of a batch is created by semicolons separated SQL queries, for example:

```
"SELECT * FROM FOO; SELECT * FROM BAR" or
"INSERT INTO FOO VALUES(1,'bar'); SELECT * FROM FOO"
```

#### Procedures

Different databases may also support creating of procedures that contains more than one SQL query. For example, the following SQLServer-specific statement creates a procedure that returns a result set containing information about employees that work at the department and a result set listing the customers of that department.

```
CREATE PROCEDURE DepartmentInfo (@DepartmentID INT) AS
SELECT * FROM Employee WHERE department = @DepartmentID
SELECT * FROM Customers WHERE department = @DepartmentID
```

#### Parameterized queries

To effectively perform a batch of similar queries, you can use parameterized queries. This means that you in your SQL query string will mark the places that usually would contain values with question marks and then provide lists of values for each parameter. For instance you can use this to insert multiple rows into the `EMPLOYEE` table while executing only a single SQL statement, for example code see "Using the Erlang API" section in the "Getting Started" chapter.

## 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. Probably 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.

### Guards

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 probably runs on another 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.

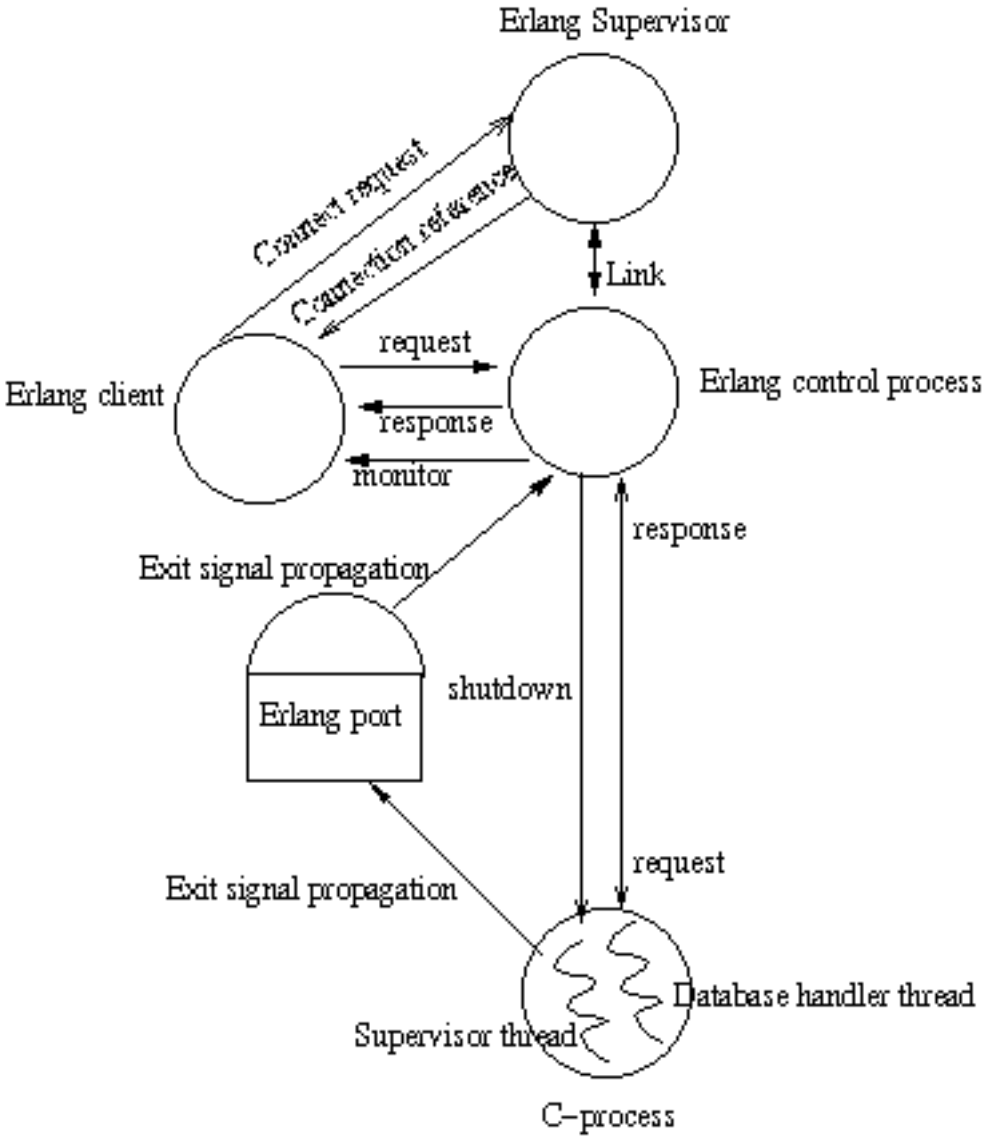


Figure 4.1: Architecture of the Erlang odbc application

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 erlang port will be kept open for exit signal propagation, if something goes wrong in the c-process and it exits we want know as mush as possible about the reason. The main communication with the c-process is done through sockets. The C-process consists of two threads, the supervisor thread and the database handler thread. The supervisor thread checks for shutdown messages on the supervisor socket and the database handler thread receives requests and sends answers on the database socket. If the database thread seems to hang on some database call, the erlang control process will send a shutdown message on the supervisor

socket, 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.

## 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 cannot 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.

# 2 Reference Manual

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

# 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 select_count again will remove the current result set association and create a new one. Calling a function which dose not operate on an associated result sets, such as `sql_query/[2,3]`, will remove the current result set association.

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 will 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(Ref, CommitMode) is the same as commit(Ref, CommitMode, infinity). If the timeout expires the client will exit with the reason timeout.

```
connection_reference() - as returned by connect/2
```

```
time_out() = milliseconds() | infinity
```

```
milliseconds() = integer() >= 0
```

```
common_reason() = connection_closed | extended_error() | term() - some kind of
explanation of what went wrong
```

```
extended_error() = {string(), integer(), Reason} - extended error type with ODBC
and native database error codes, as well as the base reason that would have been
returned had extended_errors not been enabled.
```

```
string() = list of ASCII characters
```

```
col_name() = string() - Name of column in the result set
```

```
col_names() - [col_name()] - e.g. a list of the names of the
          selected columns in the result set.
```

```
row() = {value()} - Tuple of column values e.g. one row of the
          result set.
```

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

```
rows() = [row()] - A list of rows from the result set.
```

```
result_tuple() =
     {updated, n_rows()} | {selected, col_names(), rows()}
```

```
n_rows() = 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.
```

```
odbc_data_type() = sql_integer | sql_smallint | sql_tinyint |
     {sql_decimal, precision(), scale()} |
     {sql_numeric, precision(), scale()} |
     {sql_char, size()} |
     {sql_wchar, size()} |
     {sql_varchar, size()} |
     {sql_wvarchar, size()}|
     {sql_float, precision()} |
     {sql_wlongvarchar, size()} |
     {sql_float, precision()} |
     sql_real | sql_double | sql_bit | atom()
```

```
precision() = integer()
```

```
scale() = integer()
```

```
size() = integer()
```

## ERROR HANDLING

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

## Exports

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

Types:

**`Ref = connection_reference()`**

**`CommitMode = commit | rollback`**

**`TimeOut = time_out()`**

**`Reason = not_an_explicit_commit_connection | process_not_owner_of_odbc_connection | common_reason()`**

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

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

Types:

**`ConnectStr = string()`**

An example of a connection string: `"DSN=sql-server;UID=aladdin;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=aladdin;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, on | off} | {timeout, milliseconds()} | {binary_strings, on | off} | {tuple_row, on | off} | {scrollable_cursors, on | off} | {trace_driver, on | off} | {extended_errors, on | off}`**

**`Ref = connection_reference() - should be used to access the connection.`**

**`Reason = port_program_executable_not_found | common_reason()`**

Opens a connection to the database. The connection is associated with the process that created it and can only be accessed through it. This function 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 automatically committed 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.

The default timeout is infinity

>If the option binary_strings is turned on all strings will be returned as binaries and strings inputed to param_query will be expected to be binaries. The user needs to ensure that the binary is in an encoding that the database expects. By default this option is turned off.

As default result sets are returned as a lists of tuples. The `TupleMode` option still exists to keep some degree of backwards compatibility. If the option is set to off, result sets will be returned as a lists of lists instead of a lists of tuples.

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.

**Note:**

Turning the scrollable_cursors option off is noted to make old odbc-drivers able to connect that will otherwhise fail.

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].

The `extended_errors` option enables extended ODBC error information when an operation fails. Rather than returning `{error, Reason}`, the failing function will reutrn `{error, {ODBCErrorCode, NativeErrorCode, Reason}}`. Note that this information is probably of little use when writing database-independent code, but can be of assistance in providing more sophisticated error handling when dealing with a known underlying database.

- `ODBCErrorCode` is the ODBC error string returned by the ODBC driver.
- `NativeErrorCode` is the numberic error code returned by the underlying database. The possible values and their meanings are dependent on the database being used.
- `Reason` is as per the `Reason` field when extended errors are not enabled.

**Note:**

The current implementation spawns a port program written in C that utilizes the actual ODBC driver. There is a default timeout of 5000 msec for this port programm to connect to the Erlang ODBC application. This timeout can be changed by setting an application specific environment variable 'port_timeout' with the number of milliseconds for the ODBC application. E.g.: [{odbc, [{port_timeout, 60000}]}] to set it to 60 seconds.

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

Types:

**Ref = connection_reference()**
**Reason = process_not_owner_of_odbc_connection | extended_error()**

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 cannot 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.

```
describe_table(Ref, Table) ->
describe_table(Ref, Table, Timeout) -> {ok, Description} | {error, Reason}
```

Types:

**Ref = connection_reference()**
**Table = string() - Name of databas table.**
**TimeOut = time_out()**
**Description = [{col_name(), odbc_data_type()}]**
**Reason = common_reason()**

Queries the database to find out the ODBC data types of the columns of the table `Table`.

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

Types:

**Ref = connection_reference()**
**TimeOut = time_out()**
**ColNames = col_names()**
**Rows = rows()**
**Reason = result_set_does_not_exist | driver_does_not_support_function | scrollable_cursors_disabled | process_not_owner_of_odbc_connection | common_reason()**

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

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

Types:

**Ref = connection_reference()**
**TimeOut = time_out()**
**ColNames = col_names()**

**Rows = rows()**

**Reason = result_set_does_not_exist | driver_does_not_support_function | scrollable_cursors_disabled | process_not_owner_of_odbc_connection | common_reason()**

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

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

Types:

**Ref = connection_reference()**

**TimeOut = time_out()**

**ColNames = col_names()**

**Rows = rows()**

**Reason = result_set_does_not_exist | process_not_owner_of_odbc_connection | common_reason()**

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.

```
param_query(Ref, SQLQuery, Params) ->
param_query(Ref, SQLQuery, Params, TimeOut) -> ResultTuple | {error, Reason}
```

Types:

**Ref = connection_reference()**

**SQLQuery = string() - a SQL query with parameter markers/place holders in form of question marks.**

**Params = [{odbc_data_type(), [value()]}] |[{odbc_data_type(), in_or_out(), [value()]}]**

**in_or_out = in | out | inout**

Defines IN, OUT, and IN OUT Parameter Modes for stored procedures.

**TimeOut = time_out()**

**Values = term() - Must be consistent with the Erlang data type that corresponds to the ODBC data type ODBCDataType**

Executes a parameterized SQL query. For an example see the "Using the Erlang API" in the Erlang ODBC User's Guide.

**Note:**

Use the function describe_table/[2,3] to find out which ODBC data type that is expected for each column of that table. If a column has a data type that is described with capital letters, alas it is not currently supported by the param_query function. Too know which Erlang data type corresponds to an ODBC data type see the Erlang to ODBC data type mapping in the User's Guide.

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

Types:

**`Ref = connection_reference()`**

**`TimeOut = time_out()`**

**`ColNames = col_names()`**

**`Rows = rows()`**

**`Reason = result_set_does_not_exist | driver_does_not_support_function | scrollable_cursors_disabled | process_not_owner_of_odbc_connection | common_reason()`**

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

```
start() ->
start(Type) -> ok | {error, Reason}
```

Types:

**`Type = permanent | transient | temporary`**

Starts the odbc application. Default type is temporary. See application(3)

```
stop() -> ok
```

Stops the odbc application. See application(3)

```
sql_query(Ref, SQLQuery) ->
sql_query(Ref, SQLQuery, TimeOut) -> ResultTuple | [ResultTuple] |{error,
Reason}
```

Types:

**`Ref = connection_reference()`**

**`SQLQuery = string() - The string may be composed by several SQL-queries separated by a ";", this is called a batch.`**

**`TimeOut = time_out()`**

**`ResultTuple = result_tuple()`**

**`Reason = process_not_owner_of_odbc_connection | common_reason()`**

Executes a SQL query or a batch of SQL queries. 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, and for batched queries, if the driver supports them, this function can also return a list of result tuples.

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

Types:

**`Ref = connection_reference()`**

**`SelectQuery = string()`**

SQL SELECT query.

**`TimeOut = time_out()`**

**`NrRows = n_rows()`**

**`Reason = process_not_owner_of_odbc_connection | common_reason()`**

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

Types:

**`Ref = connection_reference()`**

**`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.

**`N = integer()`**

**`TimeOut = time_out()`**

**`Reason = result_set_does_not_exist | driver_does_not_support_function | scrollable_cursors_disabled | process_not_owner_of_odbc_connection | common_reason()`**

Selects `N` consecutive rows of the result set. If `Position` is `next` it is semantically 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.

## REFERENCES

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