

NAME

DBD::Sybase - Sybase database driver for the DBI module

SYNOPSIS

```
use DBI;

$dbh = DBI->connect("dbi:Sybase:", $user, $passwd);

# See the DBI module documentation for full details
```

DESCRIPTION

DBD::Sybase is a Perl module which works with the DBI module to provide access to Sybase databases.

Connecting to Sybase

The interfaces file

The DBD::Sybase module is built on top of the Sybase *Open Client Library* API. This library makes use of the Sybase *interfaces* file (*sql.ini* on Win32 machines) to make a link between a logical server name (e.g. SYBASE) and the physical machine / port number that the server is running on. The OpenClient library uses the environment variable SYBASE to find the location of the *interfaces* file, as well as other files that it needs (such as locale files). The SYBASE environment is the path to the Sybase installation (eg '/usr/local/sybase'). If you need to set it in your scripts, then you *must* set it in a 'BEGIN{}' block:

```
BEGIN {
    $ENV{SYBASE} = '/opt/sybase/11.0.2';
}

$dbh = DBI->connect('dbi:Sybase:', $user, $passwd);
```

Specifying the server name

The server that DBD::Sybase connects to defaults to *SYBASE*, but can be specified in two ways.

You can set the *DSQUERY* environment variable:

```
$ENV{DSQUERY} = "ENGINEERING";
$dbh = DBI->connect('dbi:Sybase:', $user, $passwd);
```

Or you can pass the server name in the first argument to connect():

```
$dbh = DBI->connect("dbi:Sybase:server=ENGINEERING", $user, $passwd);
```

Specifying other connection specific parameters

It is sometimes necessary (or beneficial) to specify other connection properties. Currently the following are supported:

server

Specify the server that we should connect to

```
$dbh = DBI->connect("dbi:Sybase:server=BILLING",
    $user, $passwd);
```

The default server is *SYBASE*, or the value of the *\$DSQUERY* environment variable, if it is set.

database

Specify the database that should be made the default database.

```
$dbh = DBI->connect("dbi:Sybase:database=sybsystemprocs",
```

```
$user, $passwd);
```

This is equivalent to

```
$dbh = DBI->connect('dbi:Sybase:', $user, $passwd);  
$dbh->do("use sybssystemprocs");
```

charset

Specify the character set that the client uses.

```
$dbh = DBI->connect("dbi:Sybase:charset=iso_1",  
$user, $passwd);
```

language

Specify the language that the client uses.

```
$dbh = DBI->connect("dbi:Sybase:language=us_english",  
$user, $passwd);
```

packetSize

Specify the network packet size that the connection should use. Using a larger packet size can increase performance for certain types of queries. See the Sybase documentation on how to enable this feature on the server.

```
$dbh = DBI->connect("dbi:Sybase:packetSize=8192",  
$user, $passwd);
```

interfaces

Specify the location of an alternate *interfaces* file:

```
$dbh = DBI->connect("dbi:Sybase:interfaces=/usr/local/sybase/interfaces",  
$user, $passwd);
```

loginTimeout

Specify the number of seconds that DBI->connect() will wait for a response from the Sybase server. If the server fails to respond before the specified number of seconds the DBI->connect() call fails with a timeout error. The default value is 60 seconds, which is usually enough, but on a busy server it is sometimes necessary to increase this value:

```
$dbh = DBI->connect("dbi:Sybase:loginTimeout=240", # wait up to 4 minutes  
$user, $passwd);
```

timeout

Specify the number of seconds after which any Open Client calls will timeout the connection and mark it as dead. Once a timeout error has been received on a connection it should be closed and re-opened for further processing.

Setting this value to 0 or a negative number will result in an unlimited timeout value. See also the Open Client documentation on CS_TIMEOUT.

```
$dbh = DBI->connect("dbi:Sybase:timeout=240", # wait up to 4 minutes  
$user, $passwd);
```

scriptName

Specify the name for this connection that will be displayed in sp_who (ie in the sysprocesses table in the *program_name* column).

```
$dbh->DBI->connect("dbi:Sybase:scriptName=myScript", $user, $password);
```

hostname

Specify the hostname that will be displayed by sp_who (and will be

stored in the hostname column of sysprocesses)..

```
$dbh->DBI->connect("dbi:Sybase:hostname=kiruna", $user, $password);
```

tdsLevel

Specify the TDS protocol level to use when connecting to the server. Valid values are CS_TDS_40, CS_TDS_42, CS_TDS_46, CS_TDS_495 and CS_TDS_50. In general this is automatically negotiated between the client and the server, but in certain cases this may need to be forced to a lower level by the client.

```
$dbh->DBI->connect("dbi:Sybase:tdsLevel=CS_TDS_42", $user, $password);
```

NOTE: Setting the tdsLevel below CS_TDS_495 will disable a number of features, ?-style placeholders and CHAINED non-AutoCommit mode, in particular.

These different parameters (as well as the server name) can be strung together by separating each entry with a semi-colon:

```
$dbh = DBI->connect("dbi:Sybase:server=ENGINEERING;packetSize=8192;language=us_english;charset=iso_1",  
                  $user, $pwd);
```

Handling Multiple Result Sets

Sybase's Transact SQL has the ability to return multiple result sets from a single SQL statement. For example the query:

```
select b.title, b.author, s.amount  
       from books b, sales s  
       where s.authorID = b.authorID  
       order by b.author, b.title  
       compute sum(s.amount) by b.author
```

which lists sales by author and title and also computes the total sales by author returns two types of rows. The DBI spec doesn't really handle this situation, nor the more hairy

```
exec my_proc @p1='this', @p2='that', @p3 out
```

where 'my_proc' could return any number of result sets (ie it could perform an unknown number of 'select' statements.

I've decided to handle this by returning an empty row at the end of each result set, and by setting a special Sybase attribute in \$sth which you can check to see if there is more data to be fetched. The attribute is syb_more_results which you should check to see if you need to re-start the 'fetch()' loop.

To make sure all results are fetched, the basic 'fetch' loop can be written like this:

```
do {  
    while($d = $sth->fetch) {  
        ... do something with the data  
    }  
} while($sth->{syb_more_results});
```

You can get the type of the current result set with \$sth->{syb_result_type}. This returns a numerical value, as defined in \$SYBASE/include/cspublic.h:

```
#define CS_ROW_RESULT          (CS_INT)4040  
#define CS_CURSOR_RESULT      (CS_INT)4041  
#define CS_PARAM_RESULT       (CS_INT)4042  
#define CS_STATUS_RESULT      (CS_INT)4043
```

```
#define CS_MSG_RESULT          (CS_INT)4044
#define CS_COMPUTE_RESULT     (CS_INT)4045
```

In particular, the return status of a stored procedure is returned as CS_STATUS_RESULT (4043), and is normally the last result set that is returned in a stored proc execution.

If you add a

```
use DBD::Sybase;
```

to your script then you can use the symbolic values (CS_XXX_RESULT) instead of the numeric values in your programs, which should make them easier to read.

See also the 'syb_output_param' func() call to handle stored procedures that only return OUTPUT parameters.

```
$sth->execute() failure mode behavior
THIS HAS CHANGED IN VERSION 0.21!
```

DBD::Sybase has the ability to handle multi-statement SQL commands in a single batch. For example, you could insert several rows in a single batch like this:

```
$sth = $dbh->prepare("
insert foo(one, two, three) values(1, 2, 3)
insert foo(one, two, three) values(4, 5, 6)
insert foo(one, two, three) values(10, 11, 12)
insert foo(one, two, three) values(11, 12, 13)
");
$sth->execute;
```

If anyone of the above inserts fails for any reason then \$sth->execute will return 'undef', HOWEVER the inserts that didn't fail will still be in the database, unless 'AutoCommit' is off.

It's also possible to write a statement like this:

```
$sth = $dbh->prepare("
insert foo(one, two, three) values(1, 2, 3)
select * from bar
insert foo(one, two, three) values(10, 11, 12)
");
$sth->execute;
```

If the second 'insert' is the one that fails, then \$sth->execute will NOT return 'undef'. The error will get flagged after the rows from 'bar' have been fetched.

I know that this is not as intuitive as it could be, but I am constrained by the Sybase API here.

As an aside, I know that the example above doesn't really make sense, but I need to illustrate this particular sequence... You can also see the t/fail.t test script which shows this particular behavior.

Sybase Specific Attributes

There are a number of handle attributes that are specific to this driver. These attributes all start with syb_ so as to not clash with any normal DBI attributes.

Database Handle Attributes

The following Sybase specific attributes can be set at the Database handle level:

`syb_show_sql` (bool)
If set then the current statement is included in the string returned by `$dbh->errstr`.

`syb_show_eed` (bool)
If set, then extended error information is included in the string returned by `$dbh->errstr`. Extended error information include the index causing a duplicate insert to fail, for example.

`syb_err_handler` (subroutine ref)
Note: The syntax for the error handler is experimental and may change in future versions.

This attribute is used to set an ad-hoc error handler callback (ie a perl subroutine) that gets called before the normal error handler does it's job. If this subroutine returns 0 then the error is ignored. This is useful for handling PRINT statements in Transact-SQL, for handling messages from the Backup Server, showplan output, dbcc output, etc.

The subroutine is called with 7 parameters: the Sybase error number, the severity, the state, the line number in the SQL batch, the server name (if available), the stored procedure name (if available), and the message text.

For client-side errors the state and line number are always 0, and the server name and procedure name are always undef.

Example:

```
%showplan_msgs = map { $_ => 1 } (3612 .. 3615, 6201 .. 6225);
sub err_handler {
    my($err, $sev, $state, $line, $server, $proc, $msg) = @_;

    if($showplan_msgs{$err}) { # it's a showplan message
        print SHOWPLAN "$err - $msg\n";
        return 0; # This is not an error
    }

    return 1;
}

$dbh = DBI->connect('dbi:Sybase:server=troll', 'sa', '');
$dbh->{syb_err_handler} = \&err_handler;
$dbh->do("set showplan on");
open(SHOWPLAN, ">>/var/tmp/showplan.log") || die "Can't open showplan log: $!"
;

$dbh->do("exec someproc"); # get the showplan trace for this proc.
$dbh->disconnect;
```

`syb_flush_finish` (bool)
If `$dbh->{syb_flush_finish}` is set then `$dbh->finish` will drain any results remaining for the current command by actually fetching them. The default behaviour is to issue a `ct_cancel(CS_CANCEL_ALL)`, but this *appears* to cause connections to hang or to fail in certain cases (although I've never witnessed this myself.)

`syb_dynamic_supported` (bool)
This is a read-only attribute that returns TRUE if the dataserer you are connected to supports ?-style placeholders. Typically placeholders are not supported when using `DBD::Sybase` to connect to a MS-SQL server.

`syb_chained_txn` (bool)
If set then we use CHAINED transactions when AutoCommit is off.

Otherwise we issue an explicit BEGIN TRAN as needed. The default is off.

This attribute should usually be used only during the connect() call:

```
$dbh = DBI->connect('dbi:Sybase:', $user, $pwd, {syb_chained_txn => 1});
```

Using it at any other time with AutoCommit turned off will force a commit on the current handle.

syb_quoted_identifier (bool)

If set, then identifiers that would normally clash with Sybase reserved words can be quoted using '"identifier"'. In this case strings must be quoted with the single quote.

Default is for this attribute to be off.

syb_rowcount (int)

Setting this attribute to non-0 will limit the number of rows returned by a *SELECT*, or affected by an *UPDATE* or *DELETE* statement to the *rowcount* value. Setting it back to 0 clears the limit.

Default is for this attribute to be 0.

syb_do_proc_status (bool)

Setting this attribute causes \$sth->execute() to fetch the return status of any executed stored procs in the SQL being executed. If the return status is non-0 then \$sth->execute() will report that the operation failed (ie it will return 'undef')

Setting this attribute does NOT affect existing \$sth handles, only those that are created after setting it. To change the behavior of an existing \$sth handle use \$sth->{syb_do_proc_status}.

The default is for this attribute to be off.

syb_oc_version (string)

Returns the identification string of the version of Client Library that this binary is currently using. This is a read-only attribute.

For example:

```
troll (7:59AM):348 > perl -MDBI -e '$dbh = DBI->connect("dbi:Sybase:", "sa");
print "$dbh->{syb_oc_version}\n";'
Sybase Client-Library/11.1.1/P/Linux Intel/Linux 2.2.5 i586/1/OPT/Mon Jun 7 0
7:50:21 1999
```

This is very useful information to have when reporting a problem.

Statement Handle Attributes

The following read-only attributes are available at the statement level:

syb_more_results (bool)

See the discussion on handling multiple result sets above.

syb_result_type (int)

Returns the numeric result type of the current result set. Useful when executing stored procedures to determine what type of information is currently fetchable (normal select rows, output parameters, status results, etc...).

syb_do_proc_status (bool)

See above (under Database Handle Attributes) for an explanation.

Controlling DATETIME output formats

By default DBD::Sybase will return *DATETIME* and *SMALLDATETIME* columns in the *Nov 15 1998 11:13AM* format. This can be changed via a special `_date_fmt()` function that is accessed via the `$dbh->func()` method.

The syntax is

```
$dbh->func($fmt, '_date_fmt');
```

where `$fmt` is a string representing the format that you want to apply.

The formats are based on Sybase's standard conversion routines. The following subset of available formats has been implemented:

LONG

```
Nov 15 1998 11:30:11:496AM
```

SHORT

```
Nov 15 1998 11:30AM
```

DMY4_YYYY

```
15 Nov 1998
```

MDY1_YYYY

```
11/15/1998
```

DMY1_YYYY

```
15/11/1998
```

HMS 11:30:11

Retrieving OUTPUT parameters from stored procedures

Sybase lets you pass define OUTPUT parameters to stored procedures, which are a little like parameters passed by reference in C (or perl.)

In Transact-SQL this is done like this

```
declare @id_value int, @id_name char(10)
exec my_proc @name = 'a string', @number = 1234, @id = @id_value OUTPUT, @out_name
= @id_name OUTPUT
-- Now @id_value and @id_name are set to whatever 'my_proc' set @id and @out_name t
o
```

So how can we get at @param using DBD::Sybase?

If your stored procedure only returns OUTPUT parameters, then you can use this shorthand:

```
$sth = $dbh->prepare('...');
$sth->execute;
@results = $sth->func('syb_output_params');
```

This will return an array for all the OUTPUT parameters in the proc call, and will ignore any other results. The array will be undefined if there are no OUTPUT params, or if the stored procedure failed for some reason.

The more generic way looks like this:

```
$sth = $dbh->prepare("declare \@id_value int, \@id_name
exec my_proc @name = 'a string', @number = 1234, @id = @id_value OUTPUT, @out_na
me = @id_name OUTPUT");
$sth->execute;
do {
```

```

while($d = $sth->fetch) {
    if($sth->{syb_result_type} == 4042) { # it's a PARAM result
        $id_value = $d->[0];
        $id_name = $d->[1];
    }
}
} while($sth->{syb_more_results});

```

So the OUTPUT params are returned as one row in a special result set.

Multiple active statements on one \$dbh

It is possible to open multiple active statements on a single database handle. This is done by opening a new physical connection in `$dbh->prepare()` if there is already an active statement handle for this `$dbh`.

This feature has been implemented to improve compatibility with other drivers, but should not be used if you are coding directly to the Sybase driver.

If `AutoCommit` is OFF then multiple statement handles on a single `$dbh` is NOT supported. This is to avoid various deadlock problems that can crop up in this situation, and because you will not get real transactional integrity using multiple statement handles simultaneously as these in reality refer to different physical connections.

IMAGE and TEXT datatypes

DBD::Sybase uses the standard OpenClient conversion routines to convert data retrieved from the server into either string or numeric format.

The conversion routines convert IMAGE datatypes to a hexadecimal string. If you need the binary representation you can use something like

```
$binary = pack("H*", $hex_string);
```

to do the conversion. Note that TEXT columns are not treated this way and will be returned exactly as they were stored. Internally Sybase makes no distinction between TEXT and IMAGE columns - both can be used to store either text or binary data.

Note that IMAGE or TEXT datatypes can not be passed as parameters when using `?-style` placeholders, and `?-style` placeholders can't refer to TEXT or IMAGE columns.

AutoCommit, Transactions and Transact-SQL

When `$h->{AutoCommit}` is `*off*` all data modification SQL statements that you issue (insert/update/delete) will only take effect if you call `$dbh->commit`.

DBD::Sybase implements this via two distinct methods, depending on the setting of the `$h->{syb_chained_txn}` attribute and the version of the server that is being accessed.

If `$h->{syb_chained_txn}` is `*off*`, then the DBD::Sybase driver will send a BEGIN TRAN before the first `$dbh->prepare()`, and after each call to `$dbh->commit()` or `$dbh->rollback()`. This works fine, but will cause any SQL that contains any `*CREATE TABLE*` (or other DDL) statements to fail. These `*CREATE TABLE*` statements can be burried in a stored procedure somewhere (for example, `'sp_helprotect'` creates two temp tables when it is run). You *can* get around this limit by setting the `'ddl in tran'` option (at the database level, via `'sp_dboption'`.) You should be aware that this can have serious effects on performance as this causes locks to be held on certain system tables for the duration of the transaction.

If `$h->{syb_chained_txn}` is `*on*`, then DBD::Sybase sets the `*CHAINED*` option, which tells Sybase not to commit anything automatically. Again,

you will need to call `$dbh->commit()` to make any changes to the data permanent. In this case Sybase will not let you issue `*BEGIN TRAN*` statements in the SQL code that is executed, so if you need to execute stored procedures that have `*BEGIN TRAN*` statements in them you must use `$h->{syb_chained_txn} = 0`, or `$h->{AutoCommit} = 1`.

Using ? Placeholders & bind parameters to `$sth->execute`

This version supports the use of ? placeholders in SQL statements. It does this by using what Sybase calls `*Dynamic SQL*`. The ? placeholders allow you to write something like:

```
$sth = $dbh->prepare("select * from employee where empno = ?");

# Retrieve rows from employee where empno == 1024:
$sth->execute(1024);
while($data = $sth->fetch) {
    print "@$data\n";
}

# Now get rows where empno = 2000:

$sth->execute(2000);
while($data = $sth->fetch) {
    print "@$data\n";
}
```

When you use ? placeholders Sybase goes and creates a temporary stored procedure that corresponds to your SQL statement. You then pass variables to `$sth->execute` or `$dbh->do`, which get inserted in the query, and any rows are returned.

For those of you who are used to Transact-SQL there are some limitations to using this feature: In particular you can not pass parameters this way to stored procedures, and you can only execute a SQL statement that returns a single result set. In addition, the ? placeholders can only appear in a WHERE clause, in the SET clause of an UPDATE statement, or in the VALUES list of an INSERT statement. In particular you can't pass ? as a parameter to a stored procedure.

?-style placeholders can NOT be used to pass TEXT or IMAGE data items to the server. This is a limitation of the TDS protocol, not of DBD::Sybase.

There is also a performance issue: OpenClient creates stored procedures in tempdb for each `prepare()` call that includes ? placeholders. Creating these objects requires updating system tables in the tempdb database, and can therefore create a performance hotspot if a lot of `prepare()` statements from multiple clients are executed simultaneously. This problem has been corrected for Sybase 11.9.x and later servers, as they create "lightweight" temporary stored procs which are held in the server memory cache and don't affect the system tables at all.

In general however I find that if your application is going to run against Sybase it is better to write ad-hoc stored procedures rather than use the ? placeholders in embedded SQL.

It is not possible to retrieve the last `*IDENTITY*` value after an insert done with ?-style placeholders. This is a Sybase limitation/bug, not a DBD::Sybase problem. For example, assuming table `*foo*` has an identity column:

```
$dbh->do("insert foo(col1, col2) values(?, ?)", undef, "string1", "string2");
$sth = $dbh->prepare('select @@identity')
|| die "Can't prepare the SQL statement: $DBI::errstr";
$sth->execute || die "Can't execute the SQL statement: $DBI::errstr";
```

```
#Get the data back.
while (my $row = $sth->fetchrow_arrayref()) {
    print "IDENTITY value = $row->[0]\n";
}
```

will always return an identity value of 0, which is obviously incorrect. This behaviour is due to the fact that the handling of ?-style placeholders is implemented using temporary stored procedures in Sybase, and the value of '@@identity' is reset when the stored procedure has executed. Using an explicit stored procedure to do the insert and trying to retrieve '@@identity' after it has executed results in the same behaviour.

Please see the discussion on Dynamic SQL in the OpenClient C Programmer's Guide for details. The guide is available on-line at <http://sybooks.sybase.com/>

BUGS

You can run out of space in the tempdb database if you use a lot of calls with bind variables (ie ?-style placeholders) without closing the connection. On my system, with an 8 MB tempdb database I run out of space after 760 prepare() statements with ?-parameters. This is because Sybase creates stored procedures for each prepare() call. So my suggestion is to only use ?-style placeholders if you really need them (i.e. if you are going to execute the same prepared statement multiple times).

I have a simple bug tracking database at <http://gw.peppler.org/cgi-bin/bug.cgi>. You can use it to view known problems, or to report new ones. Keep in mind that peppler.org is connected to the net via a K56 dialup line, so it may be slow.

SEE ALSO

the DBI manpage

Sybase OpenClient C manuals.

Sybase Transact SQL manuals.

AUTHOR

DBD::Sybase by Michael Peppler

COPYRIGHT

The DBD::Sybase module is Copyright (c) 1997-1999 Michael Peppler. The DBD::Sybase module is free software; you can redistribute it and/or modify it under the same terms as Perl itself with the exception that it cannot be placed on a CD-ROM or similar media for commercial distribution without the prior approval of the author.

ACKNOWLEDGEMENTS

Tim Bunce for DBI, obviously!

See also the ACKNOWLEDGEMENTS entry in the DBI manpage.