

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

- [NAME](#)
- [SYNOPSIS](#)
- [DESCRIPTION](#)
- [Connecting to Sybase](#)
 - [The interfaces file](#)
 - [Specifying the server name](#)
 - [Specifying other connection specific parameters](#)
- [Handling Multiple Result Sets](#)
- [\\$sth->execute \(\) failure mode behavior](#)
- [Sybase Specific Attributes](#)
 - [Database Handle Attributes](#)
 - [Statement Handle Attributes](#)
- [Controlling DATETIME output formats](#)
- [Retrieving OUTPUT parameters from stored procedures](#)
- [Multiple active statements on one \\$dbh](#)
- [Working with IMAGE and TEXT columns](#)
 - [Handling IMAGE/TEXT data with syb_ct_get_data\(\)/syb_ct_send_data\(\)](#)
- [AutoCommit, Transactions and Transact-SQL](#)
 - [Behavior of \\$dbh->last_insert_id](#)
- [Using ? Placeholders & bind parameters to \\$sth->execute](#)
- [Stored Procedures and Placeholders](#)
- [Other Private Methods](#)
 - [DBD::Sybase private Statement Handle Methods](#)
- [Experimental Bulk-Load Functionality](#)
- [Using DBD::Sybase with MS-SQL](#)
- [nsql](#)
- [Multi-Threading](#)
- [BUGS](#)
- [SEE ALSO](#)
- [AUTHOR](#)
- [COPYRIGHT](#)
- [ACKNOWLEDGEMENTS](#)

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 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';
}
```

```
my $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.

host =item port

If you built DBD::Sybase with OpenClient 12.5.1 or later, then you can use the *host* and *port* values to define the server you want to connect to. This will by-pass the server name lookup in the interfaces file. This is useful in the case where the server hasn't been entered in the interfaces file.

```
$dbh = DBI->connect("dbi:Sybase:host=db1.domain.com;port=4100",
                    $user, $passwd);
```

maxConnect

By default DBD::Sybase (and the underlying OpenClient libraries) is limited to opening 25 simultaneous connections to one or more database servers. If you need more than 25 connections at the same time, you can use the *maxConnect* option to increase this number.

```
$dbh = DBI->connect("dbi:Sybase:maxConnect=100",
                    $user, $passwd);
```

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 sybsystemprocs");
```

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);
```

Note that the language has to have been installed on the server (via `langinstall` or `sp_addlanguage`) for this to work. If the language is not installed the session will default to the default language of the server.

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.

encryptPassword

Specify the use of the client password encryption supported by CT-Lib. Specify a value of 1 to use encrypted passwords.

```
$dbh=DBI->connect("dbi:Sybase:encryptPassword=1", $user, $password);
```

kerberos

Note: Requires `OpenClient 11.1.1` or later.

Sybase and `OpenClient` can use Kerberos to perform network-based login. If you use Kerberos for authentication you can use this feature and pass a `kerberos=`value parameter:

```
$dbh = DBI->connect("dbi:Sybase:kerberos=$serverprincipal", '', '');
```

In addition, if you have a system for retrieving Kerberos serverprincipals at run-time you can tell `DBD::Sybase` to call a perl subroutine to get the serverprincipal from `connect()`:

```
sub sybGetPrinc {
    my $srv = shift;
    return the serverprincipal...
}
$dbh = DBI->connect('dbi:Sybase:server=troll', '', '',
{ syb_kerberos_serverprincipal => \&sybGetPrinc });
```

The subroutine will be called with one argument (the server that we will connect to, using the normal Sybase behavior of checking the `DSQUERY` environment variable if no server is specified in the `connect()`) and is expected to return a string (the Kerberos serverprincipal) to the caller.

sslCAFile

Specify the location of an alternate *trusted.txt* file for SSL connection negotiation:

```
$dbh->DBI->connect("dbi:Sybase:sslCAFile=/usr/local/sybase/trusted.txt.
ENGINEERING", $user, $password);
```

bulkLogin

Set this to 1 if the connection is going to be used for a bulk-load operation (see *Experimental Bulk-Load functionality* elsewhere in this document.)

```
$dbh->DBI->connect("dbi:Sybase:bulkLogin=1", $user, $password);
```

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:

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

```
redo if $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 `$sth->syb_output_params` call to handle stored procedures that **only** return **OUTPUT** parameters.



`$sth->execute()` failure mode behavior

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)

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 nine parameters:

- o the Sybase error number
- o the severity
- o the state
- o the line number in the SQL batch
- o the server name (if available)
- o the stored procedure name (if available)
- o the message text
- o the current SQL command buffer
- o either of the strings "client" (for Client Library errors) or "server" (for server errors, such as SQL syntax errors, etc), allowing you to identify the error type.

As a contrived example, here is a port of the distinct error and message handlers from the Sybase documentation:

Example:

```
sub err_handler {
    my($err, $sev, $state, $line, $server,
        $proc, $msg, $sql, $err_type) = @_;
```

```
    my @msg = ();
    if($err_type eq 'server') {
        push @msg,
            ('',
             'Server message',
             sprintf('Message number: %ld, Severity %ld, State %ld, Line %ld',
                     $err,$sev,$state,$line),
             (defined($server) ? "Server '$server' " : ''),
             (defined($proc) ? "Procedure '$proc'" : ''),
             "Message String:$msg");
    } else {
        push @msg,
            ('',
             'Open Client Message:',
             sprintf('Message number: SEVERITY = (%ld) NUMBER = (%ld)',
                     $sev, $err),
             "Message String: $msg");
    }
    print STDERR join("\n",@msg);
    return 0; ## CS_SUCCEED
}
```

In a simpler and more focused example, this error handler traps showplan messages:

```
%showplan_msgs = map { $_ => 1 } (3612 .. 3615, 6201 .. 6299, 10201 ..
10299);
sub err_handler {
    my($err, $sev, $state, $line, $server,
        $proc, $msg, $sql, $err_type) = @_;
```

```
    if($showplan_msgs{$err}) { # it's a showplan message
        print SHOWPLAN "$err - $msg\n";
        return 0; # This is not an error
    }
    return 1;
}
```

and this is how you would use it:

```
$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;
```

NOTE - if you set the error handler in the DBI->connect() call like this

```
$dbh = DBI->connect('dbi:Sybase:server=troll', 'sa', '',
    { syb_err_handler => \&err_handler });
```

then the err_handler() routine will get called if there is an error during the connect itself. This is **new** behavior in DBD::Sybase 0.95.

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 dataserver 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 on if it is supported by the server.

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.

This attribute can only be set if the database handle is idle (no active statement handle.)

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.

This attribute can only be set if the database handle is idle.

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.

NOTE The result status is NOT the first result set that is fetched from a stored proc execution. If the procedure includes SELECT statements then these will be fetched first, which means that `$sth->execute` will NOT return a failure in that case as DBD::Sybase won't have seen the result status yet at that point.

The RaiseError will NOT be triggered by a non-0 return status if there isn't an associated error message either generated by Sybase (duplicate insert error, etc) or generated in the procedure via a T-SQL `raiserror` statement.

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_use_bin_0x

If set, BINARY and VARBINARY values are prefixed with '0x' in the result. The default is off.

syb_binary_images

If set, IMAGE data is returned in raw binary format. Otherwise the data is converted to a long hex string. The default is 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 07:50:21 1999
```

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

syb_server_version = item syb_server_version_string

These two attributes return the Sybase server version, respectively version string, and can be used to turn server-specific functionality on or off.

Example:

```
print "$dbh->{syb_server_version}\n$dbh->{syb_server_version_string}\n";
```

prints

```
12.5.2
Adaptive Server Enterprise/12.5.2/EBF 12061 ESD#2/P/Linux Intel/
Enterprise Linux/ase1252/1844/32-bit/OPT/Wed Aug 11 21:36:26 2004
```

syb_failed_db_fatal (bool)

If this is set, then a `connect()` request where the *database* specified doesn't exist or is not accessible will fail. This needs to be set in the attribute hash passed during the `DBI->connect()` call to be effective.

Default: off

syb_no_child_con (bool)

If this attribute is set then DBD::Sybase will **not** allow multiple simultaneously active statement handles on one database handle (i.e. multiple `$dbh->prepare()` calls without completely processing the results from any existing statement handle). This can be used to debug situations where incorrect or unexpected results are found due to the creation of a sub-connection where the connection attributes (in particular the current database) are different.

Default: off

syb_bind_empty_string_as_null (bool)

If this attribute is set then an empty string (i.e. ````) passed as a parameter to an `$sth->execute()` call will be converted to a NULL value. If the attribute is not set then an empty string is converted to a single space.

Default: off

syb_cancel_request_on_error (bool)

If this attribute is set then a failure in a multi-statement request (for example, a stored procedure execution) will cause `$sth->execute()` to return failure, and will cause any other results from this request to be discarded.

The default value (**on**) changes the behavior that DBD::Sybase exhibited up to version 0.94.

Default: on

syb_date_fmt (string)

Defines the date/time conversion string when fetching data. See the entry for the [syb_date_fmt\(\)](#) method elsewhere in this document for a description of the available formats.

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.

syb_no_bind_blob (bool)

If set then any IMAGE or TEXT columns in a query are **NOT** returned when calling `$sth->fetch` (or any variation).

Instead, you would use

```
$sth->syb_ct_get_data($column, \$data, $size);
```

to retrieve the IMAGE or TEXT data. If `$size` is 0 then the entire item is fetched, otherwise you can call this in a loop to fetch chunks of data:

```
while(1) {
    $sth->syb_ct_get_data($column, \$data, 1024);
    last unless $data;
    print OUT $data;
}
```

The fetched data is still subject to Sybase's TEXTSIZE option (see the SET command in the Sybase reference manual). This can be manipulated with DBI's **LongReadLen** attribute, but `$dbh-{LongReadLen}>` *must* be set before `$dbh->prepare()` is called to take effect (this is a change in 1.05 - previously you could call it after the `prepare()` but before the `execute()`). Note that **LongReadLen** has no effect when using DBD::Sybase with an MS-SQL server.

Note: The IMAGE or TEXT column that is to be fetched this way *must* be *last* in the select list.

See also the description of the `ct_get_data()` API call in the Sybase OpenClient manual, and the "Working with TEXT/IMAGE columns" section elsewhere in this document.



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 private `syb_date_fmt()` method.

The syntax is

```
$dbh->syb_date_fmt($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

DMY2_YYYY

15.11.1998

YMD3_YYYY

19981115

HMS

11:30:11

ISO

2004-08-21 14:36:48.080

ISO_strict

2004-08-21T14:36:48.080Z

Note that Sybase has no concept of a timezone, so the trailing ``Z" is really not correct (assumes that the

time is in UTC). However, there is no guarantee that the client and the server run in the same timezone, so assuming the timezone of the client isn't really a valid option either.



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 to
```

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

```
redo if $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.



Working with IMAGE and TEXT columns

DBD::Sybase can store and retrieve IMAGE or TEXT data (aka ``blob" data) via standard SQL statements. The **LongReadLen** handle attribute controls the maximum size of IMAGE or TEXT data being returned for each data element.

When using standard SQL the default for IMAGE data is to be converted to a hex string, but you can use the `syb_binary_images` handle attribute to change this behaviour. Alternatively you can use something like

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

to do the conversion.

IMAGE and TEXT datatypes can **not** be passed as parameters using `?`-style placeholders, and placeholders can't refer to IMAGE or TEXT columns (this is a limitation of the TDS protocol used by Sybase, not a DBD::Sybase limitation.)

There is an alternative way to access and update IMAGE/TEXT data using the native OpenClient API. This is done via `$h->func ()` calls, and is, unfortunately, a little convoluted.

Handling IMAGE/TEXT data with `syb_ct_get_data () / syb_ct_send_data ()`

With DBI 1.34 and later you can call all of these `ct_XXX ()` calls directly as statement handle methods by prefixing them with `syb_`, so for example

```
$sth->func($col, $dataref, $numbytes, 'ct_fetch_data');
```

becomes

```
$sth->syb_ct_fetch_data($col, $dataref, $numbytes);
```

\$len = ct_fetch_data(\$col, \$dataref, \$numbytes)

The `ct_get_data()` call allows you to fetch IMAGE/TEXT data in raw format, either in one piece or in chunks. To use this function you must set the `syb_no_bind_blob` statement handle to `TRUE`.

`ct_get_data()` takes 3 parameters: The column number (starting at 1) of the query, a scalar ref and a byte count. If the byte count is 0 then we read as many bytes as possible.

Note that the IMAGE/TEXT column **must** be **last** in the select list for this to work.

The call sequence is:

```
$sth = $dbh->prepare("select id, img from some_table where id = 1");
$sth->{syb_no_bind_blob} = 1;
$sth->execute;
while($d = $sth->fetchrow_arrayref) {
    # The data is in the second column
    $len = $sth->syb_ct_get_data(2, \$img, 0);
    # with DBI 1.33 and earlier, this would be
    # $len = $sth->func(2, \$img, 0, 'ct_get_data');
}
```

`ct_get_data()` returns the number of bytes that were effectively fetched, so that when fetching chunks you can do something like this:

```
while(1) {
    $len = $sth->syb_ct_get_data(2, $imgchunk, 1024);
    ... do something with the $imgchunk ...
    last if $len != 1024;
}
```

To explain further: Sybase stores IMAGE/TEXT data separately from normal table data, in a chain of pagesize blocks (a Sybase database page is defined at the server level, and can be 2k, 4k, 8k or 16k in size.) To update an IMAGE/TEXT column Sybase needs to find the head of this chain, which is known as the `text pointer`". As there is no `where` clause when the `ct_send_data()` API is used we need to retrieve the `text pointer` for the correct data item first, which is done via the `ct_data_info(CS_GET)` call. Subsequent `ct_send_data()` calls will then know which data item to update.

\$status = ct_data_info(\$action, \$column, \$attr)

`ct_data_info()` is used to fetch or update the CS_IODESC structure for the IMAGE/TEXT data item that you wish to update. `$action` should be one of `CS_SET` or `CS_GET`, `$column` is the column number of the active select statement (ignored for a `CS_SET` operation) and `$attr` is a hash ref used to set the values in the struct.

[ct_data_info\(\)](#) must be first called with CS_GET to fetch the CS_IODESC structure for the IMAGE/TEXT data item that you wish to update. Then you must update the value of the *total_txtlen* structure element to the length (in bytes) of the IMAGE/TEXT data that you are going to insert, and optionally set the *log_on_update* to **TRUE** to enable full logging of the operation.

[ct_data_info\(CS_GET\)](#) will *fail* if the IMAGE/TEXT data for which the CS_IODESC is being fetched is NULL. If you have a NULL value that needs updating you must first update it to some non-NULL value (for example an empty string) using standard SQL before you can retrieve the CS_IODESC entry. This actually makes sense because as long as the data item is NULL there is **no text pointer** and no TEXT page chain for that item.

See the [ct_send_data\(\)](#) entry below for an example.

ct_prepare_send()

[ct_prepare_send\(\)](#) must be called to initialize a IMAGE/TEXT write operation. See the [ct_send_data\(\)](#) entry below for an example.

ct_finish_send()

[ct_finish_send\(\)](#) is called to finish/commit an IMAGE/TEXT write operation. See the [ct_send_data\(\)](#) entry below for an example.

ct_send_data(\$image, \$bytes)

Send \$bytes bytes of \$image to the database. The request must have been set up via [ct_prepare_send\(\)](#) and [ct_data_info\(\)](#) for this to work. [ct_send_data\(\)](#) returns **TRUE** on success, and **FALSE** on failure.

In this example, we wish to update the data in the *img* column where the *id* column is 1. We assume that DBI is at version 1.34 or later and use the direct method calls:

```
# first we need to find the CS_IODESC data for the data
$sth = $dbh->prepare("select img from imgtable where id = 1");
$sth->execute;
while($sth->fetch) {      # don't care about the data!
    $sth->syb_ct_data_info('CS_GET', 1);
}
```

```
# OK - we have the CS_IODESC values, so do the update:
$sth->syb_ct_prepare_send();
# Set the size of the new data item (that we are inserting), and make
# the operation unlogged
$sth->syb_ct_data_info('CS_SET', 1, {total_txtlen => length($image),
log_on_update => 0});
# now transfer the data (in a single chunk, this time)
$sth->syb_ct_send_data($image, length($image));
# commit the operation
$sth->ct_finish_send();
```

The [ct_send_data\(\)](#) call can also transfer the data in chunks, however you must know the total size of the

image before you start the insert. For example:

```
# update a database entry with a new version of a file:
my $size = -s $file;
# first we need to find the CS_IODESC data for the data
$sth = $dbh->prepare("select img from imgtable where id = 1");
$sth->execute;
while($sth->fetch) {      # don't care about the data!
    $sth->syb_ct_data_info('CS_GET', 1);
}
```

```
# OK - we have the CS_IODESC values, so do the update:
$sth->syb_ct_prepare_send();
# Set the size of the new data item (that we are inserting), and make
# the operation unlogged
$sth->syb_ct_data_info('CS_SET', 1, {total_txtlen => $size, log_on_update
=> 0});
```

```
# open the file, and store it in the db in 1024 byte chunks.
open(IN, $file) || die "Can't open $file: $!";
while($size) {
    $to_read = $size > 1024 ? 1024 : $size;
    $bytesread = read(IN, $buff, $to_read);
    $size -= $bytesread;
```

```
    $sth->syb_ct_send_data($buff, $bytesread);
}
close(IN);
# commit the operation
$sth->syb_ct_finish_send();
```



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_helpprotect` 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.

Behavior of `$dbh->last_insert_id`

This version of `DBD::Sybase` includes support for the `last_insert_id()` call, with the following caveats:

The `last_insert_id()` call is simply a wrapper around a `select @@identity` query. To be successful (i.e. to return the correct value) this must be executed on the same connection as the `INSERT` that generated the new `IDENTITY` value. Therefore the statement handle that was used to perform the insert **must** have been closed/freed before `last_insert_id()` can be called. Otherwise `last_insert_id()` will be forced to open a different connection to perform the query, and will return an invalid value (usually in this case it will return 0).

`last_insert_id()` ignores any parameters passed to it, and will NOT return the last `@@identity` value generated in the case where placeholders were used, or where the insert was encapsulated in a stored procedure.



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

`DBD::Sybase` supports the use of `?` placeholders in SQL statements as long as the underlying library and database engine supports it. 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.

DBD::Sybase uses the underlying Sybase API calls to handle ?-style placeholders. For select/insert/update/delete statements DBD::Sybase calls the `ct_dynamic()` family of Client Library functions, which gives DBD::Sybase data type information for each parameter to the query.

You can only use ?-style placeholders for statements that return a single result set, and 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.

The DBI docs mention the following regarding NULL values and placeholders:

```
Binding an `undef' (NULL) to the placeholder will not
select rows which have a NULL `product_code'! Refer to the
SQL manual for your database engine or any SQL book for
the reasons for this. To explicitly select NULLs you have
to say "`WHERE product_code IS NULL'" and to make that
general you have to say:
```

```
... WHERE (product_code = ? OR (? IS NULL AND product_code IS NULL))
```

```
and bind the same value to both placeholders.
```

This will *not* work with a Sybase database server. If you attempt the above construct you will get the following error:

The datatype of a parameter marker used in the dynamic prepare statement could not be resolved.

The specific problem here is that when using ? placeholders the `prepare()` operation is sent to the database server for parameter resolution. This extracts the datatypes for each of the placeholders. Unfortunately the `? is null` construct doesn't tie the ? placeholder with an existing table column, so the database server can't find the data type. As this entire operation happens inside the Sybase libraries there is no easy way for DBD::Sybase to work around it.

Note that Sybase will normally handle the `foo = NULL` construct the same way that other systems handle `foo is NULL`, so the convoluted construct that is described above is not necessary to obtain the correct results when querying a Sybase database.

The underlying API does not support ?-style placeholders for stored procedures, but see the section on titled **Stored Procedures and Placeholders** elsewhere in this document.

?-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.

Out of curiosity I did some simple timings to see what the overhead of doing a prepare with ? placeholders is vs. a straight SQL prepare and vs. a stored procedure prepare. Against an 11.0.3.3 server (linux) the placeholder prepare is significantly slower, and you need to do ~30 `execute()` calls on the prepared statement to make up for the overhead. Against a 12.0 server (solaris) however the situation was very different, with placeholder `prepare()` calls *slightly* faster than straight SQL `prepare()`. This is something that I *really* don't understand, but the numbers were pretty clear.

In all cases stored proc `prepare()` calls were *clearly* faster, and consistently so.

This test did not try to gauge concurrency issues, however.

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/>



Stored Procedures and Placeholders

DBD::Sybase has the ability to use ?-style placeholders as parameters to stored proc calls. The requirements are that the stored procedure call be initiated with an `exec` and that it be the only statement in the batch that is being prepared():

For example, this prepares a stored proc call with named parameters:

```
my $sth = $dbh->prepare("exec my_proc \@p1 = ?, \@p2 = ?");
$sth->execute('one', 'two');
```

You can also use positional parameters:

```
my $sth = $dbh->prepare("exec my_proc ?, ?");
$sth->execute('one', 'two');
```

You may *not* mix positional and named parameter in the same prepare.

You *can't* mix placeholder parameters and hard coded parameters. For example

```
$sth = $dbh->prepare("exec my_proc \@p1 = 1, \@p2 = ?");
```

will *not* work - because the @p1 parameter isn't parsed correctly and won't be sent to the server.

You can specify *OUTPUT* parameters in the usual way, but you can **NOT** use `bind_param_inout()` to get the output result - instead you have to call `fetch()` and/or `$sth->func('syb_output_params')`:

```
my $sth = $dbh->prepare("exec my_proc \@p1 = ?, \@p2 = ?, \@p3 = ? OUTPUT ");
$sth->execute('one', 'two', 'three');
my (@data) = $sth->syb_output_params();
```

DBD::Sybase does not attempt to figure out the correct parameter type for each parameter (it would be possible to do this for most cases, but there are enough exceptions that I preferred to avoid the issue for the time being). DBD::Sybase defaults all the parameters to `SQL_CHAR`, and you have to use `bind_param()` with an explicit type value to set this to something different. The type is then remembered, so you only need to use the explicit call once for each parameter:

```
my $sth = $dbh->prepare("exec my_proc \@p1 = ?, \@p2 = ?");
$sth->bind_param(1, 'one', SQL_CHAR);
$sth->bind_param(2, 2.34, SQL_FLOAT);
$sth->execute;
....
$sth->execute('two', 3.456);
etc...
```

Note that once a type has been defined for a parameter you can't change it.

When binding `SQL_NUMERIC` or `SQL_DECIMAL` data you may get fatal conversion errors if the scale or the precision exceeds the size of the target parameter definition.

For example, consider the following stored proc definition:

```
declare proc my_proc @p1 numeric(5,2) as...
```

and the following prepare/execute snippet:

```
my $sth = $dbh->prepare("exec my_proc \@p1 = ?");
$sth->bind_param(1, 3.456, SQL_NUMERIC);
```

This generates the following error:

```
DBD::Sybase::st execute failed: Server message number=241 severity=16 state=2 line=0 procedure=dbitest
text=Scale error during implicit conversion of NUMERIC value '3.456' to a NUMERIC field.
```

You can tell Sybase (and DBD::Sybase) to ignore these sorts of errors by setting the *arithabort* option:

```
$dbh->do("set arithabort off");
```

See the *set* command in the Sybase Adaptive Server Enterprise Reference Manual for more information on the *set* command and on the *arithabort* option.



Other Private Methods

DBD::Sybase private Statement Handle Methods

@data = \$sth->syb_describe([\$assoc])

Retrieves the description of each of the output columns of the current result set. Each element of the returned array is a reference to a hash that describes the column. The following fields are set: NAME, TYPE, SYBTYPE, MAXLENGTH, SCALE, PRECISION, STATUS.

You could use it like this:

```
my $sth = $dbh->prepare("select name, uid from sysusers");
$sth->execute;
my @description = $sth->syb_describe;
print "$description[0]->{NAME}\n";           # prints "name"
print "$description[0]->{MAXLENGTH}\n";     # prints 30
....
```

```
while(my $row = $sth->fetch) {
    ....
}
```

The STATUS field is a string which can be tested for the following values: CS_CANBENULL, CS_HIDDEN, CS_IDENTITY, CS_KEY, CS_VERSION_KEY, CS_TIMESTAMP and CS_UPDATABLE. See table 3-46 of the Open Client Library Reference Manual for a description of each of these values.

The TYPE field is the data type that Sybase::CTlib converts the column to when retrieving the data, so a DATETIME column will be returned as a CS_CHAR_TYPE column.

The SYBTYPE field is the real Sybase data type for this column.

Note that the symbolic values of the `CS_XXX` symbols isn't available yet in `DBD::Sybase`.



Experimental Bulk-Load Functionality

Starting with release 1.04.2 `DBD::Sybase` has the ability to use Sybase's BLK (bulk-loading) API to perform fast data loads. Basic usage is as follows:

```
my $dbh = DBI->connect('dbi:Sybase:server=MY_SERVER;bulkLogin=1', $user, $pwd);
```

```
$dbh->begin_work; # optional.
my $sth = $dbh->prepare("insert the_table values(?, ?, ?, ?, ?)",
                        {syb_bcp_attribs => { identity_flag => 0,
                                              identity_column => 0 }});
while(<DATA>) {
  chomp;
  my @row = split(/\|/, $_); # assume a pipe-delimited file...
  $sth->execute(@row);
}
$dbh->commit;
print "Sent ", $sth->rows, " to the server\n";
$sth->finish;
```

First, you need to specify the new `bulkLogin` attribute in the connection string, which turns on the `CS_BULK_LOGIN` property for the connection. Without this property the BLK api will not be functional.

You call `$dbh->prepare()` with a regular `INSERT` statement and the special `syb_bcp_attribs` attribute to turn on BLK handling of the data. The `identity_flag` sub-attribute can be set to 1 if your source data includes the values for the target table's `IDENTITY` column. If the target table has an `IDENTITY` column but you want the insert operation to generate a new value for each row then leave `identity_flag` at 0, but set `identity_col` to the column number of the identity column (it's usually the first column in the table, but not always.)

The number of placeholders in the `INSERT` statement *must* correspond to the number of columns in the table, and the input data *must* be in the same order as the table's physical column order. Any column list in the `INSERT` statement (i. e. `insert table(a, b, c,...) values(...)`) is ignored.

The value of `AutoCommit` is ignored for BLK operations - rows are only committed when you call `$dbh->commit`.

You can call `$dbh->rollback` to cancel any uncommitted rows, but this *also* cancels the rest of the BLK operation: any attempt to load rows to the server after a call to `$dbh->rollback()` will fail.

If a row fails to load due to a `CLIENT` side error (such as a data conversion error) then `$sth->execute()` will return a failure (i.e. `false`) and `$sth->errstr` will have the reason for the error.

If a row fails on the `SERVER` side (for example due to a duplicate row error) then the entire batch (i.e. between two

`$dbh->commit()` calls) will fail. This is normal behavior for BLK/bcp.

The Bulk-Load API is very sensitive to data conversion issues, as all the conversions are handled on the client side, and the row is pre-formatted before being sent to the server. By default any conversion that is flagged by Sybase's `cs_convert()` call will result in a failed row. Some of these conversion errors are patently fatal (e.g. converting 'Feb 30 2001' to a DATETIME value...), while others are debatable (e.g. converting 123.456 to a NUMERIC(6,2) which results in a loss of precision). The default behavior of failing any row that has a conversion error in it can be modified by using a special error handler. Returning 0 from this handler tells DBD::Sybase to fail this row, and returning 1 means that we still want to try to send the row to the server (obviously Sybase's internal code can still fail the row at that point.) You set the handler like this:

```
DBD::Sybase::syb_set_cslib_cb(\&handler);
```

and a sample handler:

```
sub cslib_handler {
    my ($layer, $origin, $severity, $errno, $errmsg, $osmsg, $blkmsg) = @_;
```

```
    print "Layer: $layer, Origin: $origin, Severity: $severity, Error: $errno\n";
    print $msg;
    print $osmsg if($osmsg);
    print $blkmsg if $blkmsg;
```

```
    return 1 if($errno == 36)
```

```
    return 0;
}
```

Please see the `t/xbk.t` test script for some examples.

Reminder - this is an *experimental* implementation. It may change in the future, and it could be buggy.



Using DBD::Sybase with MS-SQL

MS-SQL started out as Sybase 4.2, and there are still a lot of similarities between Sybase and MS-SQL which makes it possible to use DBD::Sybase to query a MS-SQL dataserver using either the Sybase OpenClient libraries or the FreeTDS libraries (see <http://www.freetds.org>).

However, using the Sybase libraries to query an MS-SQL server has certain limitations. In particular `?`-style placeholders are not supported (although support when using the FreeTDS libraries is possible in a future release of the libraries), and certain **syb_** attributes may not be supported.

Sybase defaults the TEXTSIZE attribute (aka **LongReadLen**) to 32k, but MS-SQL 7 doesn't seem to do that correctly, resulting in very large memory requests when querying tables with TEXT/IMAGE data columns. The work-around is to

set TEXTSIZE to some decent value via `$dbh->{LongReadLen}` (if that works - I haven't had any confirmation that it does) or via `$dbh->do(``set textsize <some size>");`



nsql

The `nsql()` call is a direct port of the function of the same name that exists in `Sybase::DBlib`.

Usage:

```
@data = $dbh->func($sql, $type, $callback, 'nsql');
```

If the DBI version is 1.34 or later, then you can also call it this way:

```
@data = $dbh->syb_nsql($sql, $type, $callback);
```

This executes the query in `$sql`, and returns all the data in `@data`. The `$type` parameter can be used to specify that each returned row be in array form (i.e. `$type` passed as 'ARRAY', which is the default) or in hash form (`$type` passed as 'HASH') with column names as keys.

If `$callback` is specified it is taken as a reference to a perl sub, and each row returned by the query is passed to this subroutine *instead* of being returned by the routine (to allow processing of large result sets, for example).

`nsql` also checks three special attributes to enable deadlock retry logic (*Note* none of these attributes have any effect anywhere else at the moment):

syb_deadlock_retry count

Set this to a non-0 value to enable deadlock detection and retry logic within `nsql()`. If a deadlock error is detected (error code 1205) then the entire batch is re-submitted up to `syb_deadlock_retry` times. Default is 0 (off).

syb_deadlock_sleep seconds

Number of seconds to sleep between deadlock retries. Default is 60.

syb_deadlock_verbose (bool)

Enable verbose logging of deadlock retry logic. Default is off.

syb_nsql_nostatus (bool)

If true then stored procedure return status values (i.e. results of type `CS_STATUS_RESULT`) are ignored.

Deadlock detection will be added to the `$dbh->do()` method in a future version of `DBD::Sybase`.



Multi-Threading

DBD::Sybase is thread-safe (i.e. can be used in a multi-threaded perl application where more than one thread accesses the database server) with the following restrictions:

- **perl version >= 5.8**

DBD::Sybase requires the use of *ithreads*, available in the perl 5.8.0 release. It will not work with the older 5.005 threading model.

- **Sybase thread-safe libraries**

Sybase's Client Library comes in two flavors. DBD::Sybase must find the thread-safe version of the libraries (ending in `_r` on Unix/linux). This means Open Client 11.1.1 or later. In particular this means that you can't use the 10.0.4 libraries from the free 11.0.3.3 release on linux if you want to use multi-threading.

- **use DBD::Sybase**

You *must* include the `use DBD::Sybase;` line in your program. This is needed because DBD::Sybase needs to do some setup *before* the first thread is started.

You can check to see if your version of DBD::Sybase is thread-safe at run-time by calling `DBD::Sybase::thread_enabled()`. This will return *true* if multi-threading is available.

See `t/thread.t` for a simple example.



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 and Sybase 11.5.x or older. This is because Sybase creates stored procedures for each `prepare()` call. In 11.9.x and later Sybase will create ``light-weight'' stored procedures which don't use up any space in the `tempdb` database.

The `primary_key_info()` method will only return data for tables where a declarative ``primary key'' constraint was included when the table was created.

I have a simple bug tracking database at <http://www.peppler.org/cgi-bin/bug.cgi> . You can use it to view known problems, or to report new ones.



SEE ALSO

DBI

Sybase OpenClient C manuals.

Sybase Transact SQL manuals.



AUTHOR

DBD::Sybase by Michael Pepler



COPYRIGHT

The DBD::Sybase module is Copyright (c) 1996-2004 Michael Pepler. The DBD::Sybase module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.



ACKNOWLEDGEMENTS

Tim Bunce for DBI, obviously!

See also *DBI/ACKNOWLEDGEMENTS*.



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

[DBD-Sybase documentation](#) | [view source](#)
