1

DBD::Sybase

Version

Version 0.13.

Author and Contact Details

The driver author is Michael Peppler. He can be contacted via the *dbi-users* mailing list, or at *mpeppler@peppler.org*.

Supported Database Versions and Options

The `DBD::Sybase` module supports Sybase 10.x and 11.x, and offers limited support for accessing Microsoft MS-SQL 6.x, but not 7.x, server. Assuming that OpenClient 10.x or 11.x is available, `DBD::Sybase` can be used to connect to Sybase 4.x servers.

Connect Syntax

The DSN for `DBD::Sybase` is of the general form `dbi:Sybase:attr=value;attr=value`. The following attributes are supported:

server

Specify the Sybase server to connect to.

database

Specify the database within the server that should be made the default database (via `USE database`).

charset

Specify the client character set to use. Useful if the client's default character set is different from the server. Using this will enable automatic character conversion from one character set to the other.

packetSize

Set the network packetSize. Setting a larger packet size can increase the network throughput. See the Sybase documentation on how to use this as it may require changing the server configuration values.

hostname

Set the hostname that will be stored in the sysprocesses table for this process.

loginTimeout

Specify the number of seconds that `DBI->connect()` will wait for a response from the Sybase server. The default is 60 seconds. This was added in the 0.14 release.

timeout

Specify the number of seconds that `DBD::Sybase` will wait for a server response. If no response is received within that timeframe the command fails with a timeout error and the connection is marked dead. The default is to not timeout. Setting a timeout of 0 is the same as no timeout. This was added in the 0.14 release.

Numeric Data Handling

The driver supports INTEGER, SMALLINT, TINYINT, MONEY, SMALLMONEY, FLOAT, REAL, DOUBLE, `NUMERIC(p,s)`, and `DECIMAL(p,s)`.

All but the NUMERIC/DECIMAL datatypes are hardware specific, but INTEGER is always a 32bit int, SMALLINT is 16bit, and TINYINT is 8bit.

Precision for numeric/decimal is from 1 to 38, and scale is from 0 to 38.

Numeric/decimal values are returned as perl strings by default, even if the scale is 0 and the precision is small enough to fit in an integer value. All other numbers are returned in native format.

String Data Handling

`DBD::Sybase` supports CHAR, VARCHAR, BINARY, and VARBINARY, all limited to 255 characters in length. The CHAR type is fixed length and blank padded.

Sybase automatically converts CHAR and VARCHAR data between the character set of the server (see the `syscharset` system table) and the character set of the client, defined by the locale setting of the client. The BINARY and VARBINARY types are not converted. UTF-8 is supported.

See the OpenClient International Developer's Guide in the Sybase OpenClient manuals for more on character set issues.

Strings can be concatenated using the + SQL operator.

Date Data Handling

Sybase supports the DATETIME and SMALLDATETIME values. A DATETIME can have a value from Jan 1 1753 to Dec 31, 9999 with a 300th of a second resolution. A SMALLDATETIME has a range of Jan 1 1900 to Jun 6 2079 with a 1 minute resolution.

The current date on the server is obtained with the `GETDATE()` SQL function.

The Sybase date format depends on the locale settings for the client. The default date format is based on the ''C'' locale:

```
Feb 16 1999 12:07PM
```

In this same locale, Sybase understands several input formats in addition to the one above:

```
2/16/1998 12:07PM
1998/02/16 12:07
1998-02-16 12:07
19980216 12:07
```

If the time portion is omitted, it is set to 00:00. If the date portion is omitted, it is set to Jan 1 1900. If the century is omitted, it is assumed to be 1900 if year < 50, and 2000 if year >= 50.

You can use the special `_date_fmt()` private method (accessed *via* `$dbh-func()>`) to change the date input and output format. 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
```

Use the `CONVERT()` SQL function to convert date and time values from other formats. For example:

```
UPDATE a_table
   SET date_field = CONVERT(datetime_field, '1999-02-21', 105)
```

`CONVERT()` is a generic conversion function that can convert to and from most datatypes. See the `CONVERT()` function in Chapter 2 of the Sybase Reference Manual.

Arithmetic on date time types is done on dates via the `DATEADD()`, `DATEPART()`, and `DATEDIFF()` Transact SQL functions. For example:

```
SELECT DATEDIFF(ss, date1, date2)
```

returns the difference in seconds between *date1* and *date2*.

Sybase does not understand time zones at all, except that the `GETDATE()` SQL function returns the date in the time zone that the server is running in (*via* `localtime`).

The following SQL expression can be used to convert an integer ‘‘seconds since 1-jan-1970’’ value (‘‘unix time’’) to the corresponding database date time:

```
DATEADD(ss, unixtime_field, 'Jan 1 1970')
```

Note however that the server does not understand time zones, and will therefore give the local unixtime on the server, and not the correct value for the GMT time zone.

If you know that the server runs in the same timezone as the client, you can use:

```
use Time::Local;
$time_to_database = timegm(localtime($unixtime));
```

to convert the unixtime value before sending it to Sybase.

To do the reverse, converting from a database date time value to unix time, you can use:

```
DATEDIFF(ss, 'Jan 1 1970', datetime_field)
```

The same GMT vs localtime caveat applies in this case. If you know that the server runs in the same timezone as the client, you can convert the returned value to the correct GMT based value with this Perl expression:

```
use Time::Local;
$time = timelocal(gmtime($time_from_database));
```

LONG/BLOB Data Handling

Sybase supports an IMAGE and a TEXT type for LONG/BLOB data. Each type can hold up to 2GB of binary data, including nul characters. The main difference between an IMAGE and a TEXT column lies in how the client libraries treat the data on input and output. TEXT data is entered and returned ‘‘as is’’. IMAGE data is returned as a long hex string, and should be entered in the same way.

LongReadLen and *LongTrunkOk* attributes have no effect. The default limit for TEXT/IMAGE data is 32Kb, but this can be changed by the SET TEXTSIZE Transact-SQL command.

Bind parameters can *not* be used to insert TEXT or IMAGE data to Sybase.

Other Data Handling issues

The `DBD::Sybase` driver does not support the `type_info()` method yet.

Sybase does not automatically convert numbers to strings or strings to numbers. You need to explicitly call the `CONVERT` SQL function. However, placeholders don't need special handling because `DBD::Sybase` knows what type each placeholder needs to be.

Transactions, Isolation and Locking

`DBD::Sybase` supports transactions. The default transaction isolation level is ''Read Commited''.

Sybase supports READ COMMITED, READ UNCOMMITED and SERIALIZABLE isolation levels. The level be changed per-connection or per-statement by executing `SET TRANSACTION_ISOLATION LEVEL x`, where *x* is 0 for READ UNCOMMITED, 1 for READ COMMITED, and 3 for SERIALIZABLE.

By default, a READ query will acquire a shared lock on each page that it reads. This will allow any other process to read from the table, but will block any process trying to obtain an exclusive lock (for update). The shared lock is only maintained for the time the server needs to actually read the page, not for the entire length of the SELECT operation. (Disclaimer: 11.9.2 and later servers have various new locking mechanisms that I'm not familiar with yet.)

There is no explicit LOCK TABLE statement. Appending ''WITH HOLDLOCK'' to a SELECT statement can be used to force an exclusive lock to be acquired on a table. It is usually called within a transaction. This call is generally not needed.

The correct way to do a multi-table update with Sybase is to wrap the entire operation in a transaction. This will ensure that locks will be acquired in the correct order, and that no intervening action from another process will modify any rows that your operation is currently modifying.

No-Table Expression Select Syntax

To select a constant expression, that is, an expression that doesn't involve data from a database table or view, you can select it without naming a table:

```
SELECT getdate()
```

Table Join Syntax

Outer joins are supported using the `=*` (right outer join) and `*=` (left outer join) operators:

```
SELECT customer_name, order_date
FROM customers, orders
WHERE customers.cust_id =* orders.cust_id
```

For all rows in the customers table that have no matching rows in the orders table, Sybase returns NULL for any select list expressions containing columns from the orders table.

Table and Column Names

The names of Sybase identifiers, such as tables and columns, cannot exceed 30 characters in length.

The first character must be an alphabetic character (as defined by the current server character set) or an underscore (_). Subsequent characters can be alphabetic, and may include currency symbols, @, #, and _. Identifiers can't include embedded spaces or the %, !, ^, *, or . symbols. In addition, identifiers must not be on the ''reserved word'' list (see the Sybase documentation for a complete list).

Table names or column names *may* be quoted if the `set quoted_identifier` option is turned on. This allows the user to get around the reserved word limitation. When this option is set, character strings enclosed in double quotes are treated as identifiers, and strings enclosed in single quotes are treated as literal strings.

By default identifiers are case-sensitive. This can be turned off by changing the default sort order for the server.

National characters can be used in identifier names without quoting.

Case Sensitivity of LIKE Operator

The Sybase LIKE operator is case sensitive.

The `UPPER` function can be used to force a case insensitive match, *e.g.*, `UPPER(name) LIKE 'TOM%'` although that does prevent Sybase from making use of any index on the name column to speed up the query.

Note however that case sensitivity can be modified at the Sybase server level by loading a different *sort order*. See the Sybase System Administration manuals for details.

Row ID

Sybase does not support a pseudo ''row ID'' column.

Automatic Key or Sequence Generation

Sybase supports an IDENTITY feature for automatic key generation. Declaring a table with an IDENTITY column will generate a new value for each insert. The values are monotonically increasing, but are not guaranteed to be sequential.

To fetch the value generated and used by the last insert, you can:

```
SELECT @@IDENTITY
```

Sybase does not support sequence generators, although ad-hoc stored procedures to generate sequence numbers are quite easy to write*.

```
* See:

    http://techinfo.sybase.com/css/techinfo.nsf/DocID/ID=860

  for a complete explanation of the various possibilities.
```

Automatic Row Numbering and Row Count Limiting

Sybase does not offer a pseudocolumn that sequentially numbers the rows fetched by a select statement.

Parameter Binding

Parameter binding is directly suported by Sybase. However, there are two downsides that one should be aware of:

Firstly, Sybase creates an internal stored procedure for each `prepare()` call that includes `?` style parameters. These stored procedures live in the *tempdb* database, and are only destroyed when the connection is closed. It is quite possible to run out of *tempdb* space if a lot of `prepare()` calls with placeholders are being made in a script.

Secondly, because all the temporary stored procedures are created in *tempdb*, this causes a potential hot-spot due to the locking of system tables in *tempdb*. I'm told that this performance problem will be removed in an upcoming release of Sybase (possibly 11.9.4 or 12.0).

The `:1` placeholder style is not supported and the TYPE attribute to `bind_param()` is currently ignored, so unsupported values don't generate a warning. However, trying to bind a TEXT or IMAGE datatype will fail.

Stored Procedures

Sybase stored procedures are written in Transact-SQL, Sybase's procedural extension to SQL.

Stored procedures are called exactly the same way as regular SQL, and can return the same types of results (*i.e.*, a SELECT in the stored procedure can be retrieved with $sth->fetch>).

If the stored procedure returns data via OUTPUT parameters, then these must be declared first:

```
$sth = $dbh->prepare(qq[
   declare \@name varchar(50)
   exec getName 1234, \@name output
]);
```

Stored procedures can't be called with bind (?) parameters. So the following would be illegal:

```
$sth = $dbh->prepare("exec my_proc ?");
$sth->execute('foo');
```

Use this code instead:

```
$sth = $dbh->prepare("exec my_proc 'foo'");
$sth->execute;
```

Because Sybase stored procedures almost always return more than one result set, you should always make sure to use a loop until *syb_more_results* is 0:

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

Table Metadata

DBD::Sybase supports the table_info() method.

The *syscolumns* table has one row per column per table. See the definitions of the Sybase system tables for details. However, the easiest method is to use the sp_help stored procedure.

The easiest way to get detailed information about the indexes of a table is to use the sp_helpindex (or sp_helpkey) stored procedure.

Driver-specific Attributes and Methods

`DBD::Sybase` has the following driver specific database handle attributes:

syb_show_sql

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

syb_show_eed

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.

And the following driver specific statement handle attributes:

syb_more_results

See the discussion on handling multiple result sets above.

syb_result_type

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 . . .).

One private method is provided:

_date_fmt

Set the default date conversion and display formats. See the description elsewhere in this document.

Positioned updates and deletes

Sybase does not support positioned updates or deletes.

Differences from the DBI Specification

The *LongReadLen* and *LongTruncOk* attributes are not supported.

Note that `DBD::Sybase` does not fully parse the statement until it's executed. Thus, attributes like *$sth->{NUM_OF_FIELDS}* are not available until after `$sth->execute()` has been called. This is valid behavior but is important to note when porting applications originally written for other drivers.

URLs to More Database/Driver Specific Information

```
http://www.sybase.com
http://techinfo.sybase.com
http://sybooks.sybase.com
```

Concurrent use of Multiple Handles

DBD::Sybase supports an unlimited number of concurrent database connections to one or more databases.

It is not normally possible for Sybase clients to prepare/execute a new statement handle while still fetching data from another statment handle that is associated with the same database handle. However, DBD::Sybase emulates this by opening a new connection that will automatically be closed when the new statement handle is destroyed. You should be aware that there are some subtle but significant transaction issues with this approach.

Other Significant Database or Driver Features

Sybase and DBD::Sybase allow multiple statements to be prepared with one call and then executed with one call. The results are fed back to the client as a stream of tabular data. Stored procedures can also return a stream of multiple data sets. Each distinct set of results is treated as a normal single result set, so `fetch()` returns `undef` at the end of each set. To see if there are more data sets to follow, the *syb_more_results* attribute can be checked. Here is a typical loop making use of this Sybase specific feature:

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

Sybase also has rich and powerful stored procedure and trigger functionality and encourages you to use them.