NAME

DBD::mysql - MySQL driver for the Perl5 Database Interface (DBI)

SYNOPSIS

```
use DBI;

$dsn = "DBI:mysql:database=$database;host=$hostname;port=$port";

$dbh = DBI->connect($dsn, $user, $password);


$drh = DBI->install_driver("mysql");
@databases = DBI->data_sources("mysql");
or
@databases = DBI->data_sources("mysql",
{"host" => $host, "port" => $port, "user" => $user, password => $pass});

$sth = $dbh->prepare("SELECT * FROM foo WHERE bla");
or
$sth = $dbh->prepare("LISTFIELDS $table");
or
$sth = $dbh->prepare("LISTINDEX $table $index");
$sth->execute;
$numRows = $sth->rows;
$numFields = $sth->{'NUM_OF_FIELDS'};
$sth->finish;

$rc = $drh->func('createdb', $database, $host, $user, $password, 'admin');
$rc = $drh->func('dropdb', $database, $host, $user, $password, 'admin');
$rc = $drh->func('shutdown', $host, $user, $password, 'admin');
$rc = $drh->func('reload', $host, $user, $password, 'admin');

$rc = $dbh->func('createdb', $database, 'admin');
$rc = $dbh->func('dropdb', $database, 'admin');
$rc = $dbh->func('shutdown', 'admin');
$rc = $dbh->func('reload', 'admin');
```

EXAMPLE

```
#!/usr/bin/perl

use strict;
use DBI();

# Connect to the database.
my $dbh = DBI->connect("DBI:mysql:database=test;host=localhost",
"joe", "joe's password",
{'RaiseError' => 1});

# Drop table 'foo'. This may fail, if 'foo' doesn't exist.
# Thus we put an eval around it.
eval { $dbh->do("DROP TABLE foo") };
print "Dropping foo failed: $@\n" if $@;

# Create a new table 'foo'. This must not fail, thus we don't
# catch errors.
$dbh->do("CREATE TABLE foo (id INTEGER, name VARCHAR(20)
```

```
# INSERT some data into 'foo'. We are using $dbh->quote() for
# quoting the name.
$dbh->do("INSERT INTO foo VALUES (1, " . $dbh->quote("Tim") . ")");

# Same thing, but using placeholders
$dbh->do("INSERT INTO foo VALUES (?, ?)", undef, 2, "Jochen");

# Now retrieve data from the table.
my $sth = $dbh->prepare("SELECT * FROM foo");
$sth->execute();
while (my $ref = $sth->fetchrow_hashref()) {
print "Found a row: id = $ref->{'id'}, name = $ref->{'name'}\n";
}
$sth->finish();

# Disconnect from the database.
$dbh->disconnect();
```

DESCRIPTION

DBD::mysql is the Perl5 Database Interface driver for the MySQL database. In other words: DBD::mysql is an interface between the Perl programming language and the MySQL programming API that comes with the MySQL relational database management system. Most functions provided by this programming API are supported. Some rarely used functions are missing, mainly because no-one ever requested them. :-)

In what follows we first discuss the use of DBD::mysql, because this is what you will need the most. For installation, see the sections on INSTALLATION, and "WIN32 INSTALLATION" below. See EXAMPLE for a simple example above.

From perl you activate the interface with the statement

```
use DBI;
```

After that you can connect to multiple MySQL database servers and send multiple queries to any of them via a simple object oriented interface. Two types of objects are available: database handles and statement handles. Perl returns a database handle to the connect method like so:

```
$dbh = DBI->connect("DBI:mysql:database=$db;host=$host",
$user, $password, {RaiseError => 1});
```

Once you have connected to a database, you can execute SQL statements with:

```
my $query = sprintf("INSERT INTO foo VALUES (%d, %s)",
$number, $dbh->quote("name"));
$dbh->do($query);
```

See DBI for details on the quote and do methods. An alternative approach is

```
$dbh->do("INSERT INTO foo VALUES (?, ?)", undef,
$number, $name);
```

in which case the quote method is executed automatically. See also the bind_param method in DBI. See "DATABASE HANDLES" below for more details on database handles.

If you want to retrieve results, you need to create a so-called statement handle with:

```
$sth = $dbh->prepare("SELECT * FROM $table");
$sth->execute();
```

This statement handle can be used for multiple things. First of all you can retrieve a row of data:

```
my $row = $sth->fetchrow_hashref();
```

If your table has columns ID and NAME, then `$row` will be hash ref with keys ID and NAME. See ''STATEMENT HANDLES'' below for more details on statement handles.

But now for a more formal approach:

Class Methods

connect

```
use DBI;

$dsn = "DBI:mysql:$database";
$dsn = "DBI:mysql:database=$database;host=$hostname";
$dsn = "DBI:mysql:database=$database;host=$hostname;port=$port";

$dbh = DBI->connect($dsn, $user, $password);
```

A `database` must always be specified.

host
port

The hostname, if not specified or specified as '' or 'localhost', will default to a MySQL server running on the local machine using the default for the UNIX socket. To connect to a MySQL server on the local machine via TCP, you must specify the loopback IP address (127.0.0.1) as the host.

Should the MySQL server be running on a non-standard port number, you may explicitly state the port number to connect to in the `hostname` argument, by concatenating the *hostname* and *port number* together separated by a colon (:) character or by using the `port` argument.

To connect to a MySQL server on localhost using TCP/IP, you must specify the hostname as 127.0.0.1 (with the optional port).

mysql_client_found_rows

Enables (TRUE value) or disables (FALSE value) the flag CLIENT_FOUND_ROWS while connecting to the MySQL server. This has a somewhat funny effect: Without mysql_client_found_rows, if you perform a query like

```
UPDATE $table SET id = 1 WHERE id = 1
```

then the MySQL engine will always return 0, because no rows have changed. With mysql_client_found_rows however, it will return the number of rows that have an id 1, as some people are expecting. (At least for compatibility to other engines.)

mysql_compression

As of MySQL 3.22.3, a new feature is supported: If your DSN contains the option ''mysql_compression=1'', then the communication between client and server will be compressed.

mysql_connect_timeout

If your DSN contains the option ''mysql_connect_timeout=##'', the connect request to the server will timeout if it has not been successful after the given number of seconds.

mysql_write_timeout

If your DSN contains the option ''mysql_write_timeout=##'', the write operation to the server will timeout if it has not been successful after the given number of seconds.

mysql_read_timeout

If your DSN contains the option ''mysql_read_timeout=##'', the read operation to the server will timeout if it has not been successful after the given number of seconds.

mysql_init_command
: If your DSN contains the option "mysql_init_command=##", then this SQL statement is executed when connecting to the MySQL server. It is automatically re-executed if reconnection occurs.

mysql_skip_secure_auth
: This option is for older mysql databases that don't have secure auth set

mysql_read_default_file
mysql_read_default_group
: These options can be used to read a config file like /etc/my.cnf or ~/.my.cnf. By default MySQL's C client library doesn't use any config files unlike the client programs (mysql, mysqladmin, ...) that do, but outside of the C client library. Thus you need to explicitly request reading a config file, as in

```
$dsn = "DBI:mysql:test;mysql_read_default_file=/home/joe/my.cnf";
$dbh = DBI->connect($dsn, $user, $password)
```

: The option mysql_read_default_group can be used to specify the default group in the config file: Usually this is the *client* group, but see the following example:

```
[client]
host=localhost

[perl]
host=perlhost
```

: (Note the order of the entries! The example won't work, if you reverse the [client] and [perl] sections!)

: If you read this config file, then you'll be typically connected to *localhost*. However, by using

```
$dsn = "DBI:mysql:test;mysql_read_default_group=perl;"
. "mysql_read_default_file=/home/joe/my.cnf";
$dbh = DBI->connect($dsn, $user, $password);
```

: you'll be connected to *perlhost* Note that if you specify a default group and do not specify a file, then the default config files will all be read. See the documentation of the C function *mysql_options()* for details.

mysql_socket
: As of MySQL 3.21.15, it is possible to choose the Unix socket that is used for connecting to the server. This is done, for example, with

```
mysql_socket=/dev/mysql
```

: Usually there's no need for this option, unless you are using another location for the socket than that built into the client.

mysql_ssl
: A true value turns on the CLIENT_SSL flag when connecting to the MySQL database:

```
mysql_ssl=1
```

: This means that your communication with the server will be encrypted.

: If you turn mysql_ssl on, you might also wish to use the following flags:

mysql_ssl_client_key
mysql_ssl_client_cert

mysql_ssl_ca_file
mysql_ssl_ca_path
mysql_ssl_cipher
: These are used to specify the respective parameters of a call to mysql_ssl_set, if mysql_ssl is turned on.

mysql_local_infile
: As of MySQL 3.23.49, the LOCAL capability for LOAD DATA may be disabled in the MySQL client library by default. If your DSN contains the option ‘‘mysql_local_infile=1’’, LOAD DATA LOCAL will be enabled. (However, this option is *ineffective* if the server has also been configured to disallow LOCAL.)

mysql_multi_statements
: As of MySQL 4.1, support for multiple statements separated by a semicolon (;) may be enabled by using this option. Enabling this option may cause problems if server-side prepared statements are also enabled.

Prepared statement support (server side prepare)
: As of 3.0002_1, server side prepare statements were on by default (if your server was >= 4.1.3). As of 3.0009, they were off by default again due to issues with the prepared statement API (all other mysql connectors are set this way until C API issues are resolved). The requirement to use prepared statements still remains that you have a server >= 4.1.3

  To use server side prepared statements, all you need to do is set the variable mysql_server_prepare in the connect:

  `$dbh` = DBI->connect(‘‘DBI:mysql:database=test;host=localhost;mysql_server_prepare=1’’, ‘‘, { RaiseError => 1, AutoCommit => 1 });

  * Note: delimiter for this param is ';'

  There are many benefits to using server side prepare statements, mostly if you are performing many inserts because of that fact that a single statement is prepared to accept multiple insert values.

  To make sure that the 'make test' step tests whether server prepare works, you just need to export the env variable MYSQL_SERVER_PREPARE:

  export MYSQL_SERVER_PREPARE=1

mysql_embedded_options
: The option <mysql_embedded_options> can be used to pass 'command-line' options to embedded server.

  Example:

  use DBI;
  `$testdsn`=‘‘DBI:mysqlEmb:database=test;mysql_embedded_options=--help,--verbose’’;
  `$dbh` = DBI->connect($testdsn,‘‘a’’,‘‘b’’);

  This would cause the command line help to the embedded MySQL server library to be printed.

mysql_embedded_groups
: The option <mysql_embedded_groups> can be used to specify the groups in the config file(*my.cnf*) which will be used to get options for embedded server. If not specified [server] and [embedded] groups will be used.

  Example:

  `$testdsn`=‘‘DBI:mysqlEmb:database=test;mysql_embedded_groups=embedded_server,common’’;

Private MetaData Methods

ListDBs

```
my $drh = DBI->install_driver("mysql");
@dbs = $drh->func("$hostname:$port", '_ListDBs');
@dbs = $drh->func($hostname, $port, '_ListDBs');
@dbs = $dbh->func('_ListDBs');
```

Returns a list of all databases managed by the MySQL server running on `$hostname`, port `$port`. This is a legacy method. Instead, you should use the portable method

```
@dbs = DBI->data_sources("mysql");
```

Server Administration

admin

```
$rc = $drh->func("createdb", $dbname, [host, user, password,], 'admin');
$rc = $drh->func("dropdb", $dbname, [host, user, password,], 'admin');
$rc = $drh->func("shutdown", [host, user, password,], 'admin');
$rc = $drh->func("reload", [host, user, password,], 'admin');

or

$rc = $dbh->func("createdb", $dbname, 'admin');
$rc = $dbh->func("dropdb", $dbname, 'admin');
$rc = $dbh->func("shutdown", 'admin');
$rc = $dbh->func("reload", 'admin');
```

For server administration you need a server connection. For obtaining this connection you have two options: Either use a driver handle (drh) and supply the appropriate arguments (host, defaults localhost, user, defaults to '' and password, defaults to ''). A driver handle can be obtained with

```
$drh = DBI->install_driver('mysql');
```

Otherwise reuse the existing connection of a database handle (dbh).

There's only one function available for administrative purposes, comparable to the mysqladmin programs. The command being execute depends on the first argument:

createdb
: Creates the database `$dbname`. Equivalent to "mysqladmin create `$dbname`".

dropdb
: Drops the database `$dbname`. Equivalent to "mysqladmin drop `$dbname`".

  It should be noted that database deletion is *not prompted for*in an y way. Nor is it undoable from DBI.

  ```
  Once you issue the dropDB() method, the database will be gone!
  ```

  These method should be used at your own risk.

shutdown
: Silently shuts down the database engine. (Without prompting!) Equivalent to "mysqladmin shutdown".

reload
: Reloads the servers configuration files and/or tables. This can be particularly important if you modify access privileges or create new users.

DATABASE HANDLES

The DBD::mysql driver supports the following attributes of database handles (read only):

```
$errno = $dbh->{'mysql_errno'};
$error = $dbh->{'mysql_error'};
$info = $dbh->{'mysql_hostinfo'};
$info = $dbh->{'mysql_info'};
$insertid = $dbh->{'mysql_insertid'};
$info = $dbh->{'mysql_protoinfo'};
$info = $dbh->{'mysql_serverinfo'};
$info = $dbh->{'mysql_stat'};
$threadId = $dbh->{'mysql_thread_id'};
```

These correspond to *mysql_errno()*, *mysql_error()*, *mysql_get_host_info()*, *mysql_info()*, *mysql_insert_id()*, *mysql_get_proto_info()*, *mysql_get_server_info()*, *mysql_stat()* and *mysql_thread_id()*, respectively.

```
$info_hashref = $dhb->{mysql_dbd_stats}
```

DBD::mysql keeps track of some statistics in the mysql_dbd_stats attribute. The following stats are being maintained:

auto_reconnects_ok
: The number of times that DBD::mysql successfully reconnected to the mysql server.

auto_reconnects_failed
: The number of times that DBD::mysql tried to reconnect to mysql but failed.

The DBD::mysql driver also supports the following attribute(s) of database handles (read/write):

```
$bool_value = $dbh->{mysql_auto_reconnect};
$dbh->{mysql_auto_reconnect} = $AutoReconnect ? 1 : 0;
```

mysql_auto_reconnect
: This attribute determines whether DBD::mysql will automatically reconnect to mysql if the connection be lost. This feature defaults to off; however, if either the GATEWAY_INTERFACE or MOD_PERL environment variable is set, DBD::mysql will turn mysql_auto_reconnect on. Setting mysql_auto_reconnect to on is not advised if 'lock tables' is used because if DBD::mysql reconnect to mysql all table locks will be lost. This attribute is ignored when AutoCommit is turned off, and when AutoCommit is turned off, DBD::mysql will not automatically reconnect to the server.

  It is also possible to set the default value of the `mysql_auto_reconnect` attribute for the `$dbh` by passing it in the `\%attr` hash for `DBI-connect>`.

  Note that if you are using a module or framework that performs reconnections for you (for example DBIx::Connector in fixup mode), this value must be set to 0.

mysql_use_result
: This attribute forces the driver to use mysql_use_result rather than mysql_store_result. The former is faster and less memory consuming, but tends to block other processes. mysql_store_result is the default due to that fact storing the result is expected behavior with most applications.

  It is possible to set the default value of the `mysql_use_result` attribute for the `$dbh` using several ways:

```
- through DSN

$dbh= DBI->connect("DBI:mysql:test;mysql_use_result=1", "root", "");

- after creation of database handle

$dbh->{'mysql_use_result'}=0; #disable
$dbh->{'mysql_use_result'}=1; #enable
```

It is possible to set/unset the `mysql_use_result` attribute after creation of the statement handle. See below.

mysql_enable_utf8
: This attribute determines whether DBD::mysql should assume strings stored in the database are utf8. This feature defaults to off.

  When set, a data retrieved from a textual column type (char, varchar, etc) will have the UTF-8 flag turned on if necessary. This enables character semantics on that string. You will also need to ensure that your database / table / column is configured to use UTF8. See Chapter 10 of the mysql manual for details.

  Additionally, turning on this flag tells MySQL that incoming data should be treated as UTF-8. This will only take effect if used as part of the call to *connect()*. If you turn the flag on after connecting, you will need to issue the command `SET NAMES utf8` to get the same effect.

  This option is experimental and may change in future versions.

mysql_bind_type_guessing
: This attribute causes the driver (emulated prepare statements) to attempt to guess if a value being bound is a numeric value, and if so, doesn't quote the value. This was created by Dragonchild and is one way to deal with the performance issue of using quotes in a statement that is inserting or updating a large numeric value. This was previously called `unsafe_bind_type_guessing` because it is experimental. I have successfully run the full test suite with this option turned on, the name can now be simply `mysql_bind_type_guessing`.

  CAVEAT: Even though you can insert an integer value into a character column, if this column is indexed, if you query that column with the integer value not being quoted, it will not use the index:

  MariaDB [test]> explain select * from test where value0 = '3' G *************************** 1. row *************************** id: 1 select_type: SIMPLE
  table: test type: ref possible_keys: value0 key: value0 key_len: 13 ref: const rows: 1 Extra: Using index condition 1 row in set (0.00 sec)

  MariaDB [test]> explain select * from test where value0 = 3 -> G *************************** 1. row *************************** id: 1 select_type: SIMPLE
  table: test type: ALL possible_keys: value0 key: NULL
  key_len: NULL
  ref: NULL
  rows: 6 Extra: Using where 1 row in set (0.00 sec)

  See bug: https://rt.cpan.org/Ticket/Display.html?id=43822

  `mysql_bind_type_guessing` can be turned on via

```
  - through DSN

  my $dbh= DBI->connect('DBI:mysql:test', 'username', 'pass',
  { mysql_bind_type_guessing => 1})

  - OR after handle creation

  $dbh->{mysql_bind_type_guessing} = 1;
```

mysql_bind_comment_placeholders
: This attribute causes the driver (emulated prepare statements) will cause any placeholders in comments to be bound. This is not correct prepared statement behavior, but some developers

have come to depend on this behavior, so I have made it available in 4.015

mysql_no_autocommit_cmd
: This attribute causes the driver to not issue 'set autocommit' either through explicit or using *mysql_autocommit()*. This is particularly useful in the case of using MySQL Proxy.

  See the bug report:

  https://rt.cpan.org/Public/Bug/Display.html?id=46308

  `mysql_no_autocommit_cmd` can be turned on via

```
- through DSN

my $dbh= DBI->connect('DBI:mysql:test', 'username', 'pass',
{ mysql_no_autocommit_cmd => 1})

- OR after handle creation

$dbh->{mysql_no_autocommit_cmd} = 1;
```

STATEMENT HANDLES

The statement handles of DBD::mysql support a number of attributes. You access these by using, for example,

```
my $numFields = $sth->{'NUM_OF_FIELDS'};
```

Note, that most attributes are valid only after a successful *execute*. An`undef` v alue will returned in that case. The most important exception is the `mysql_use_result` attribute: This forces the driver to use mysql_use_result rather than mysql_store_result. The former is faster and less memory consuming, but tends to block other processes. (That's why mysql_store_result is the default.)

To set the `mysql_use_result` attribute, use either of the following:

```
my $sth = $dbh->prepare("QUERY", { "mysql_use_result" => 1});
```

or

```
my $sth = $dbh->prepare("QUERY");
$sth->{"mysql_use_result"} = 1;
```

Column dependent attributes, for example *NAME*, the column names, are returned as a reference to an array. The array indices are corresponding to the indices of the arrays returned by *fetchrow* and similar methods. For example the following code will print a header of table names together with all rows:

```
my $sth = $dbh->prepare("SELECT * FROM $table");
if (!$sth) {
die "Error:" . $dbh->errstr . "\n";
}
if (!$sth->execute) {
die "Error:" . $sth->errstr . "\n";
}
my $names = $sth->{'NAME'};
my $numFields = $sth->{'NUM_OF_FIELDS'} - 1;
for my $i ( 0..$numFields ) {
printf("%s%s", $i ? "," : "", $$names[$i]);
}
print "\n";
while (my $ref = $sth->fetchrow_arrayref) {
for my $i ( 0..$numFields ) {
```

```
printf("%s%s", $i ? "," : "", $$ref[$i]);
}
print "\n";
}
```

For portable applications you should restrict yourself to attributes with capitalized or mixed case names. Lower case attribute names are private to DBD::mysql. The attribute list includes:

ChopBlanks
: this attribute determines whether a *fetchrow* will chop preceding and trailing blanks off the column values. Chopping blanks does not have impact on the *max_length* attribute.

mysql_insertid
: MySQL has the ability to choose unique key values automatically. If this happened, the new ID will be stored in this attribute. An alternative way for accessing this attribute is via `$dbh->{'mysql_insertid'}`. (Note we are using the `$dbh` in this case!)

mysql_is_blob
: Reference to an array of boolean values; TRUE indicates, that the respective column is a blob. This attribute is valid for MySQL only.

mysql_is_key
: Reference to an array of boolean values; TRUE indicates, that the respective column is a key. This is valid for MySQL only.

mysql_is_num
: Reference to an array of boolean values; TRUE indicates, that the respective column contains numeric values.

mysql_is_pri_key
: Reference to an array of boolean values; TRUE indicates, that the respective column is a primary key.

mysql_is_auto_increment
: Reference to an array of boolean values; TRUE indicates that the respective column is an AUTO_INCREMENT column. This is only valid for MySQL.

mysql_length
mysql_max_length
: A reference to an array of maximum column sizes. The *max_length* is the maximum physically present in the result table, *length* gives the theoretically possible maximum. *max_length* is valid for MySQL only.

mysql_clientinfo
: List information of the MySQL client library that DBD::mysql was built against:

  print ''$dbh->{mysql_clientinfo}n'';

  5.2.0-MariaDB

mysql_clientversion
: print ''$dbh->{mysql_clientversion}n'';

  50200

mysql_serverversion
: print ''$dbh->{mysql_serverversion}n'';

  50200

NAME
: A reference to an array of column names.

NULLABLE
: A reference to an array of boolean values; TRUE indicates that this column may contain NULL's.

NUM_OF_FIELDS
: Number of fields returned by a *SELECT* or *LISTFIELDS* statement. You may use this for checking whether a statement returned a result: A zero value indicates a non-SELECT statement like *INSERT*, *DELETE* or *UPDATE*.

mysql_table
: A reference to an array of table names, useful in a *JOIN* result.

TYPE
: A reference to an array of column types. The engine's native column types are mapped to portable types like *DBI::SQL_INTEGER or DBI::SQL_VARCHAR as good as possible. Not all native types have a meaningful equivalent, for example DBD::mysql::FIELD_TYPE_INTERVAL is mapped to DBI::SQL_VARCHAR If you need the native column types, use mysql_type*. See below.

mysql_type
: A reference to an array of MySQL's native column types, for example *DBD::mysql::FIELD_TYPE_SHORT()* or *DBD::mysql::FIELD_TYPE_STRING()*. Use the *TYPE* attribute, if you want portable types like *DBI::SQL_SMALLINT or DBI::SQL_VARCHAR*

mysql_type_name
: Similar to mysql, but type names and not numbers are returned. Whenever possible, the ANSI SQL name is preferred.

mysql_warning_count
: The number of warnings generated during execution of the SQL statement. This attribute is available on both statement handles and database handles.

TRANSACTION SUPPORT

Beginning with DBD::mysql 2.0416, transactions are supported. The transaction support works as follows:

- By default AutoCommit mode is on, following the DBI specifications.
- If you execute

  ```
  $dbh->{'AutoCommit'} = 0;
  ```

  or

  ```
  $dbh->{'AutoCommit'} = 1;
  ```

  then the driver will set the MySQL server variable autocommit to 0 or 1, respectively. Switching from 0 to 1 will also issue a COMMIT, following the DBI specifications.
- The methods

  ```
  $dbh->rollback();
  $dbh->commit();
  ```

  will issue the commands COMMIT and ROLLBACK, respectively. A ROLLBACK will also be issued if AutoCommit mode is off and the database handles DESTROY method is called. Again, this is following the DBI specifications.

Given the above, you should note the following:

- You should never change the server variable autocommit manually, unless you are ignoring DBI's transaction support.
- Switching AutoCommit mode from on to off or vice versa may fail. You should always check for errors, when changing AutoCommit mode. The suggested way of doing so is using the

DBI flag RaiseError. If you don't like RaiseError, you have to use code like the following:

```
$dbh->{'AutoCommit'} = 0;
if ($dbh->{'AutoCommit'}) {
# An error occurred!
}
```

- If you detect an error while changing the AutoCommit mode, you should no longer use the database handle. In other words, you should disconnect and reconnect again, because the transaction mode is unpredictable. Alternatively you may verify the transaction mode by checking the value of the server variable autocommit. However, such behaviour isn't portable.
- DBD::mysql has a "reconnect" feature that handles the so-called MySQL "morning bug": If the server has disconnected, most probably due to a timeout, then by default the driver will reconnect and attempt to execute the same SQL statement again. However, this behaviour is disabled when AutoCommit is off: Otherwise the transaction state would be completely unpredictable after a reconnect.
- The "reconnect" feature of DBD::mysql can be toggled by using the mysql_auto_reconnect attribute. This behaviour should be turned off in code that uses LOCK TABLE because if the database server time out and DBD::mysql reconnect, table locks will be lost without any indication of such loss.

MULTIPLE RESULT SETS

As of version 3.0002_5, DBD::mysql supports multiple result sets (Thanks to Guy Harrison!). This is the first release of this functionality, so there may be issues. Please report bugs if you run into them!

The basic usage of multiple result sets is

```
do
{
while (@row= $sth->fetchrow_array())
{
do stuff;
}
} while ($sth->more_results)
```

An example would be:

```
$dbh->do("drop procedure if exists someproc") or print $DBI::errstr;

$dbh->do("create procedure someproc() deterministic
begin
declare a,b,c,d int;
set a=1;
set b=2;
set c=3;
set d=4;
select a, b, c, d;
select d, c, b, a;
select b, a, c, d;
select c, b, d, a;
end") or print $DBI::errstr;

$sth=$dbh->prepare('call someproc()') ||
die $DBI::err.": ".$DBI::errstr;

$sth->execute || die DBI::err.": ".$DBI::errstr; $rowset=0;
```

```
do {
print "\nRowset ".++$i."\n----------------------------------------\n\n";
foreach $colno (0..$sth->{NUM_OF_FIELDS}-1) {
print $sth->{NAME}->[$colno]."\t";
}
print "\n";
while (@row= $sth->fetchrow_array()) {
foreach $field (0..$#row) {
print $row[$field]."\t";
}
print "\n";
}
} until (!$sth->more_results)
```

For more examples, please see the eg/ directory. This is where helpful DBD::mysql code snippets will be added in the future.

Issues with Multiple result sets

So far, the main issue is if your result sets are ''jagged'', meaning, the number of columns of your results vary. Varying numbers of columns could result in your script crashing. This is something that will be fixed soon.

MULTITHREADING

The multithreading capabilities of DBD::mysql depend completely on the underlying C libraries: The modules are working with handle data only, no global variables are accessed or (to the best of my knowledge) thread unsafe functions are called. Thus DBD::mysql is believed to be completely thread safe, if the C libraries are thread safe and you don't share handles among threads.

The obvious question is: Are the C libraries thread safe? In the case of MySQL the answer is ''mostly'' and, in theory, you should be able to get a ''yes'', if the C library is compiled for being thread safe (By default it isn't.) by passing the option -with-thread-safe-client to configure. See the section on *How to make a threadsafe client*in the man ual.

ASYNCHRONOUS QUERIES

You can make a single asynchronous query per MySQL connection; this allows you to submit a long-running query to the server and have an event loop inform you when it's ready. An asynchronous query is started by either setting the 'async' attribute to a true value in the ''do'' in DBI method, or in the ''prepare'' in DBI method. Statements created with 'async' set to true in prepare always run their queries asynchronously when ''execute'' in DBI is called. The driver also offers three additional methods: `mysql_async_result`, `mysql_async_ready`, and `mysql_fd`. `mysql_async_result` returns what do or execute would have; that is, the number of rows affected. `mysql_async_ready` returns true if `mysql_async_result` will not block, and zero otherwise. They both return `undef` if that handle is not currently running an asynchronous query. `mysql_fd` returns the file descriptor number for the MySQL connection; you can use this in an event loop.

Here's an example of how to use the asynchronous query interface:

```
use feature 'say';
$dbh->do('SELECT SLEEP(10)', { async => 1 });
until($dbh->mysql_async_ready) {
say 'not ready yet!';
sleep 1;
}
my $rows = $dbh->mysql_async_result;
```

INSTALLATION

Windows users may skip this section and pass over to WIN32 INSTALLATION below. Others, go on reading.

Environment Variables

For ease of use, you can now set environment variables for DBD::mysql installation. You can set any or all of the options, and export them by putting them in your .bashrc or the like:

```
export DBD_MYSQL_CFLAGS=-I/usr/local/mysql/include/mysql
export DBD_MYSQL_LIBS="-L/usr/local/mysql/lib/mysql -lmysqlclient"
export DBD_MYSQL_EMBEDDED=
export DBD_MYSQL_CONFIG=mysql_config
export DBD_MYSQL_NOCATCHSTDERR=0
export DBD_MYSQL_NOFOUNDROWS=0
export DBD_MYSQL_SSL=
export DBD_MYSQL_TESTDB=test
export DBD_MYSQL_TESTHOST=localhost
export DBD_MYSQL_TESTPASSWORD=s3kr1+
export DBD_MYSQL_TESTPORT=3306
export DBD_MYSQL_TESTUSER=me
```

The most useful may be the host, database, port, socket, user, and password.

Installation will first look to your mysql_config, and then your environment variables, and then it will guess with intelligent defaults.

Installing with CPAN

First of all, you do not need an installed MySQL server for installing DBD::mysql. However, you need at least the client libraries and possibly the header files, if you are compiling DBD::mysql from source. In the case of MySQL you can create a client-only version by using the configure option --without-server. If you are using precompiled binaries, then it may be possible to use just selected RPM's like MySQL-client and MySQL-devel or something similar, depending on the distribution.

I recommend trying automatic installation via the CPAN module. Try

```
cpan
```

If you are using the CPAN module for the first time, it will prompt you a lot of questions. If you finally receive the CPAN prompt, enter

```
install DBD::mysql
```

Manual Installation

If this fails (which may be the case for a number of reasons, for example because you are behind a firewall or don't have network access), you need to do a manual installation. First of all you need to fetch the modules from CPAN

```
L<https://metacpan.org>
```

The following modules are required

```
DBI
DBD::mysql
```

Then enter the following commands (note - versions are just examples):

```
gzip -cd DBI-(version).tar.gz | tar xf -
cd DBI-(version)
perl Makefile.PL
make
make test
make install

cd ..
gzip -cd DBD-mysql-(version)-tar.gz | tar xf -
cd DBD-mysql-(version)
```

```
perl Makefile.PL
make
make test
make install
```

During "perl Makefile.PL" you will be prompted some questions. Other questions are the directories with header files and libraries. For example, of your file *mysql.h* is in */usr/include/mysql/mysql.h*, then enter the header directory */usr*, likewise for */usr/lib/mysql/libmysqlclient.a* or */usr/lib/libmysqlclient.so*.

MARIADB NATIVE CLIENT INSTALLATION

The MariaDB native client is another option for connecting to a MySQL database licensed LGPL 2.1. To build DBD::mysql against this client, you will first need to build the client. Generally, this is done with the following:

```
cd path/to/src/mariadb-native-client
cmake -G "Unix Makefiles'
make
sudo make install
```

Once the client is built and installed, you can build DBD::mysql against it:

```
perl Makefile.PL --testuser=xxx --testpassword=xxx --testsocket=/path/to//mysqld.sock --mysq
make
make test
make install
```

WIN32 INSTALLATION

If you are using ActivePerl, you may use ppm to install DBD-mysql.

```
ppm install DBI
ppm install DBD::mysql
```

If you need an HTTP proxy, you might need to set the environment variable http_proxy, for example like this:

```
set http_proxy=http://myproxy.com:8080/
```

I recommend using the win32clients package for installing DBD::mysql under Win32, available for download on www.tcx.se. The following steps have been required for me:

- Extract sources into *C:*. This will create a directory *C:mysql* with subdirectories include and lib.

  IMPORTANT: Make sure this subdirectory is not shared by other TCX files! In particular do *not* store the MySQL server in the same directory. If the server is already installed in *C:mysql*, choose a location like *C:tmp*, extract the win32clients there. Note that you can remove this directory entirely once you have installed DBD::mysql.

- Extract the DBD::mysql sources into another directory, for example *C:srcsiteperl*

- Open a CMD.exe shell and change directory to *C:srcsiteperl*.

- The next step is only required if you repeat building the modules: Make sure that you have a clean build tree by running

  ```
  nmake realclean
  ```

  If you don't have VC++, replace nmake with your flavor of make. If error messages are reported in this step, you may safely ignore them.

- Run

  ```
  perl Makefile.PL
  ```

  which will prompt you for some settings. The really important ones are:

```
Which DBMS do you want to use?
```

enter a 1 here (MySQL only), and

```
Where is your mysql installed? Please tell me the directory that
contains the subdir include.
```

where you have to enter the win32clients directory, for example *C:mysql* or *C:tmpmysql*.

- Continued in the usual way:

```
nmake
nmake install
```

AUTHORS

Originally, there was a non-DBI driver, Mysql, which was much like PHP drivers such as mysql and mysqli. The **Mysql** module was originally written by Andreas König <koenig@kulturbox.de> who still, to this day, contributes patches to DBD::mysql. An emulated version of Mysql was provided to DBD::mysql from Jochen Wiedmann, but eventually deprecated as it was another bundle of code to maintain.

The first incarnation of DBD::mysql was developed by Alligator Descartes, who was also aided and abetted by Gary Shea, Andreas König and Tim Bunce.

The current incarnation of **DBD::mysql** was written by Jochen Wiedmann, then numerous changes and bug-fixes were added by Rudy Lippan. Next, prepared statement support was added by Patrick Galbraith and Alexy Stroganov (who also solely added embedded server support).

For the past nine years DBD::mysql has been maintained by Patrick Galbraith (*patg@patg.net*), and recently with the great help of Michiel Beijen (*michiel.beijen@gmail.com*), along with the entire community of Perl developers who keep sending patches to help continue improving DBD::mysql

CONTRIBUTIONS

Anyone who desires to contribute to this project is encouraged to do so. Currently, the source code for this project can be found at Github:

<https://github.com/perl5-dbi/DBD-mysql/>

Either fork this repository and produce a branch with your changeset that the maintainer can merge to his tree, or create a diff with git. The maintainer is more than glad to take contributions from the community as many features and fixes from DBD::mysql have come from the community.

COPYRIGHT

LICENSE

MAILING LIST SUPPORT

This module is maintained and supported on a mailing list, dbi-users.

To subscribe to this list, send an email to

dbi-users-subscribe@perl.org

Mailing list archives are at

<http://groups.google.com/group/perl.dbi.users?hl=en&lr=>

ADDITIONAL DBI INFORMATION

Additional information on the DBI project can be found on the World Wide Web at the following URL:

<http://dbi.perl.org>

where documentation, pointers to the mailing lists and mailing list archives and pointers to the most current versions of the modules can be used.

Information on the DBI interface itself can be gained by typing:

```
perldoc DBI
```

Information on DBD::mysql specifically can be gained by typing:

```
perldoc DBD::mysql
```

(this will display the document you're currently reading)

BUG REPORTING, ENHANCEMENT/FEATURE REQUESTS

Please report bugs, including all the information needed such as DBD::mysql version, MySQL version, OS type/version, etc to this link:

<https://rt.cpan.org/Dist/Display.html?Name=DBD-mysql>

Note: until recently, MySQL/Sun/Oracle responded to bugs and assisted in fixing bugs which many thanks should be given for their help! This driver is outside the realm of the numerous components they support, and the maintainer and community solely support DBD::mysql