Perl DBI API Reference

This appendix describes the Perl DBI application programming interface. The API consists of a set of methods and attributes for communicating with database servers and accessing databases from Perl scripts. The appendix also describes MySQL-specific extensions to DBI provided by DBD::mysql, the MySQL database driver.

I assume here a minimum version of DBI 1.50, although most of the material applies to earlier versions as well. DBI 1.50 requires at least Perl 5.6.0 (with 5.6.1 preferred). As of DBI 1.611, the minimum Perl version is 5.8.1. I also assume a minimum version of DBD::mysql 4.00. To determine your versions of DBI and DBD::mysql (assuming that they are installed), run this program:

```perl
#!/usr/bin/perl
# dbi-version.pl - display DBI and DBD::mysql versions
use DBI;
print "DBI::VERSION: $DBI::VERSION\n";
use DBD::mysql;
print "DBD::mysql::VERSION: $DBD::mysql::VERSION\n";
```

If you need to install the DBI software, see Appendix A, “Software Required to Use This Book.”

Some DBI methods and attributes are not discussed here, either because they do not apply to MySQL or because they are experimental methods that may change as they are developed or may even be dropped. Some MySQL-specific DBD methods are not discussed because they are obsolete. For more information about new or obsolete methods, see the DBI or DBD::mysql documentation, available at http://dbi.perl.org or by running the following commands:

```
% perldoc DBI
% perldoc DBI::FAQ
% perldoc DBD::mysql
```

The examples in this appendix are only brief code fragments. For complete scripts and instructions for writing them, see Chapter 8, “Writing MySQL Programs Using Perl DBI.”
H.1 Writing Scripts

Every Perl script that uses the DBI module must include the following line:

```perl
use DBI;
```

It's normally unnecessary to include a `use` line for a particular DBD-level module (such as DBD::mysql) because DBI activates the proper module when you connect to the server.

Typically, a DBI script opens a connection to a MySQL server using the `connect()` method and closes the connection with `disconnect()`. While the connection is open, SQL statements may be executed. The methods used to do this vary depending on the type of statement. Non-SELECT statements typically are performed with the `do()` method. SELECT statements typically are performed by passing the statement to `prepare()`, calling `execute()`, and finally retrieving the result set a row at a time in a loop that repeatedly invokes a row-fetching method such as `fetchrow_array()` or `fetchrow_hashref()`.

When you execute statements from within a DBI script, each statement string must consist of a single SQL statement, and should not end with a semicolon character (';') or a \g sequence. The '; ' and \g terminators are conventions of the mysql client program and are not used for DBI.

H.2 DBI Methods

The method descriptions here use a somewhat different format than the C functions in Appendix G, "C API Reference," or the PHP functions in Appendix I, "PHP API Reference." Functions in those appendixes are written in prototype form, with return value types and parameter types listed explicitly. The descriptions here indicate parameter and return value types using variables, where the leading character of each variable indicates its type: ‘$’ for a scalar, ‘@’ for an array, and ‘%’ for a hash (associative array). In addition, any parameter listed with a leading ‘\’ signifies a reference to a variable of the given type, not the variable itself. A variable name suffix of `ref` indicates that the variable’s value is a reference.

Certain variable names recur throughout this appendix and have the conventional meanings shown in Table H.1.

Table H.1 Conventional Perl DBI Variable Names

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$drh</td>
<td>A handle to a driver object</td>
</tr>
<tr>
<td>$dbh</td>
<td>A handle to a database object</td>
</tr>
<tr>
<td>$sth</td>
<td>A handle to a statement (query) object</td>
</tr>
<tr>
<td>$fh</td>
<td>A handle to an open file</td>
</tr>
<tr>
<td>$h</td>
<td>A “generic” handle; the meaning depends on context</td>
</tr>
</tbody>
</table>
1131 H.2 DBI Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rc</td>
<td>The return code from operations that return true or false</td>
</tr>
<tr>
<td>$rv</td>
<td>The return value from operations that return an integer</td>
</tr>
<tr>
<td>$rows</td>
<td>The return value from operations that return a row count</td>
</tr>
<tr>
<td>$str</td>
<td>The return value from operations that return a string</td>
</tr>
<tr>
<td>@ary</td>
<td>An array representing a list of values</td>
</tr>
<tr>
<td>@row_ary</td>
<td>An array representing a row of values returned by a query</td>
</tr>
</tbody>
</table>

Many methods accept a hash argument %attr containing attributes that affect the way the method works. This hash should be passed by reference, which you can do two ways:

- Initialize the contents of the hash value %attr before invoking the method, then pass a reference to it to the method:

```perl
my %attr = {AttrName1 => value1, AttrName2 => value2}; $ret_val = $h->method (... , %attr);
```

- Supply an anonymous hash directly in the method invocation:

```perl
$ret_val = $h->method (... , {AttrName1 => value1, AttrName2 => value2});
```

The way a method or function is used is indicated by the calling sequence. DBI-> indicates a DBI class method, DBI:: indicates a DBI function, and $DBI:: indicates a DBI variable. For methods that are called using handles, the handle name indicates the scope of the method.

- $dbh-> indicates a database-handle method, $sth-> indicates a statement-handle method, and $h-> indicates a method that may be called with different kinds of handles. Square brackets ([ ]) designate optional information. Here's an example calling sequence:

```perl
@row_ary = $dbh->selectrow_array ($stmt , [%attr , @bind_values]);
```

This indicates that the `selectrow_array()` method is called as a database-handle method, because it's invoked using $dbh->. The parameters are $stmt (a scalar value), %attr (a hash that should be passed as a reference, as indicated by the leading '\'), and @bind_values (an array). The second and third parameters are optional. The return value, @row_ary, is an array representing the row of values returned by the method.

Each method description indicates what the return value is when an error occurs, but that value is returned on error only if the RaiseError attribute is disabled. If RaiseError is enabled, the method raises an exception rather than returning, and the script automatically terminates.

In the descriptions that follow, the term "SELECT statement" should be taken to mean a SELECT statement or any other statement that returns rows, such as DESCRIBE, EXPLAIN, or SHOW.
H.2.1  DBI Class Methods

The %attr parameter for methods in this section may be used to specify method-processing attributes. (An attribute parameter that is missing or undef means “no attributes.”) For MySQL, the most important attributes are PrintError, RaiseError, and AutoCommit. Attributes passed to connect() or connect_cached() become part of the resulting database handle returned by those methods. For example, to turn on automatic script termination when a DBI error occurs within any method associated with a given database handle, enable RaiseError when you create the handle:

```perl
$dbh = DBI->connect ($data_source, $user_name, $password, {RaiseError => 1});
```

PrintError, RaiseError, and AutoCommit are discussed in Section H.4, “DBI Attributes.”

- @ary = DBI->available_drivers ([%quiet]);

  Returns a list of available DBI drivers. The default value of the optional $quiet parameter is 0, which causes a warning to be issued if multiple drivers with the same name are found. To suppress the warning, pass a $quiet value of 1.

- $dbh = DBI->connect ($data_source, $user_name, $password [], {%attr});

  Establishes a connection to a database server and returns a database handle, or undef if the connection attempt fails. To terminate a successfully established connection, invoke disconnect() using the database handle returned by connect().

```perl
$dbh = DBI->connect ("DBI:mysql:sampdb:localhost", "sampadm", "secret", {%attr})
  or die "Could not connect\n";
$dbh->disconnect ();
```

The data source can be given in several forms. The first part is always DBI:mysql:, where DBI may be given in any lettercase and the driver name, mysql, must be lowercase. Everything after the second colon (which must be present) is interpreted by the driver, so the syntax described in the following discussion does not necessarily apply to any driver module other than DBD:mysql.

Following the second colon, you may also specify a database name and hostname in the initial part of the data source string:

```perl
$data_source = "DBI:mysql:db_name";
$data_source = "DBI:mysql:db_name:hostname";
```

The database may be specified as db_name or as database=db_name. The hostname may be specified as hostname or as host=hostname.

Username and Password attributes can be passed in the %attr parameter to specify the username and password. These attributes take precedence over values passed in the $user_name and $password parameters.
my %attr = (Username => "sampadm", Password => "secret");
$dbh = DBI->connect("DBI:mysql:sampdb:localhost",
    "someuser", "somepass", \%attr)
or die "Could not connect\n";

Attributes also can be specified in the data source following the driver name, separated by commas and enclosed within parentheses. Attributes specified this way take precedence over those specified in the %attr, $user_name, and $password parameters.

my $dsn = "DBI:mysql(Username=>sampadm,Password=>secret):sampdb:localhost";
$dbh = DBI->connect ($dsn, "someuser", "somepass", \%attr)
or die "Could not connect\n";

Following the initial part of the data source string, you may specify options in attribute=value format. Each option setting should be preceded by a semicolon. For example:

DBI:mysql:sampdb:localhost;mysql_socket=/tmp/mysql.sock;mysql_compression=1

The MySQL driver understands the following options:

- **host=host_name**
  
  The host to connect to. For TCP/IP connections, a port number also may be specified by using host_name:port_num format, or by using the port attribute.

  On Unix systems, connections to the host localhost use Unix domain sockets by default. In this case, you may use mysql_socket to specify the socket filename. Use host=127.0.0.1 to connect to the local host using TCP/IP.

  On Windows systems, connections to the host "." connect to the local server using a named pipe, or TCP/IP if that doesn’t work. In this case, you may use mysql_socket to specify the pipe name.

- **port=port_num**
  
  The port number to connect to. This option is ignored for non-TCP/IP connections (for example, connections to localhost under Unix).

- **mysql_client_found_rows=val**
  
  The type of row count to return for UPDATE statements. The MySQL server can return the number of rows affected (changed), or the number of rows matched (regardless of whether they were changed). For example, an UPDATE that selects a row in its WHERE clause but sets row values to their current values matches the row but does not change it. Disabling mysql_client_found_rows by setting it to 0 tells the server to return the number of rows changed. Enabling mysql_client_found_rows by setting it to 1 tells the server to return the number of rows matched.

  By default, mysql_client_found_rows is enabled in DBD:mysql. *This differs from the C client library*, for which the default is number of rows changed.
Appendix H  Perl DBI API Reference

- **mysql_compression=1**
  Requests use of the compressed client/server communication protocol if the client and server both support it.

- **mysql_connect_timeout=seconds**
  The number of seconds to wait during the connection attempt before timing out and returning failure.

- **mysql_local_infile=val**
  Controls availability of the LOCAL capability for the LOAD DATA statement. Setting the option to 1 enables LOCAL if it is disabled in the MySQL client library by default (as long as the server has not also been configured to prohibit it). Setting the option to 0 disables LOCAL if it is enabled in the client library.

- **mysql_read_default_file=file_name**
  By default, DBI scripts do not check any MySQL option files for connection parameters. `mysql_read_default_file` enables you to specify an option file to read. The filename should be a full pathname. (Otherwise, it is interpreted relative to the current directory, and you will get inconsistent results depending on where the script is run.)

  On Unix, if you expect a script to be used by multiple users and you want each of them to connect using parameters specified in their own option file (rather than using parameters that you hardwire into the script), specify the filename as `ENV{HOME}/.my.cnf`. The script then uses the `.my.cnf` file in the home directory of whatever user happens to be running the script.

  Specifying an option filename that includes a drive letter doesn’t work under Windows, because the colon (’:) character that separates the drive letter and the following pathname is also used by DBI as a separator within the data source string. For a workaround for this problem, see Section 8.2.9, “Specifying Connection Parameters.”

- **mysql_read_default_group=group_name**
  Specifies an option file group in which to look for connection parameters. If `mysql_read_default_group` is used without `mysql_read_default_file`, the standard option files are read. If both `mysql_read_default_group` and `mysql_read_default_file` are used, only the file named by the latter is read.

  The [client] option file group is always read from option files. `mysql_read_default_group` enables you to specify a group to be read in addition to the [client] group. For example, `mysql_read_default_group=dbi` specifies that both the [dbi] and [client] groups should be used. To read only the [client] group, use `mysql_read_default_group=client`. 
- `mysql_server_prepare=val`
  Setting this option to 1 enables server-side statement preparation. Setting it to 0 (the default) causes statement preparation to be emulated on the client side.

- `mysql_socket=socket_name`
  Under Unix, this option specifies the pathname of the Unix domain socket to use for connections to localhost. Under Windows, it indicates a named-pipe name. This option is ignored for TCP/IP connections (for example, connections to hosts other than localhost on Unix).

- `mysql_ssl=val`
  `mysql_ssl_ca_file=file_name`
  `mysql_ssl_ca_path=dir_name`
  `mysql_ssl_cipher=str`
  `mysql_ssl_client_cert=file_name`
  `mysql_ssl_client_key=file_name`
  These options are used to establish a secure connection to the server using SSL. Setting `mysql_ssl` to 0 prohibits use of SSL. If `mysql_ssl` is not specified or is set to 1, SSL connections are permitted, using the values of the other options to specify connection characteristics. Their meanings are the same as the corresponding arguments of the `mysql_ssl_set()` function in the C API. For details, see the entry for that function in Appendix G, “C API Reference.” If you enable `mysql_ssl`, you should also specify values for at least the `mysql_ssl_ca_file`, `mysql_ssl_client_cert`, and `mysql_ssl_client_key` options.
  These options require SSL support in the MySQL C client library that is linked into DBD::mysql, and any MySQL server to which you connect must permit SSL connections.

- `mysql_use_result=val`
  This option affects result set retrieval. If the value is 0 (the default), DBD::mysql uses the `mysql_store_result()` C API function to retrieve rows. If the value is 1, DBD::mysql uses `mysql_use_result()` instead. For a discussion of these two functions and how they differ, see Appendix G, “C API Reference.” See also the discussion of the `mysql_use_result` statement-handle attribute in Section H.4.5, “MySQL-Specific Statement-Handle Attributes.”

If connection parameters are not specified explicitly in the arguments to `connect()`, or in any option files that the connection attributes cause to be read, DBI examines several environment variables to determine which parameters to use:

- If the data source is undefined or empty, DBI uses the value of the `DBI_DSN` variable.
• If the driver name is missing from the data source, DBI uses the value of the
  DBI_DRIVER variable.
• If the user_name or password parameters of the connect() call are undefined,
  DBI uses the values of the DBI_USER and DBI_PASS variables. This does not occur
  if the parameters are empty strings. (Use of DBI_PASS is a security risk, so you
  shouldn’t use it on multiple-user systems where environment variable values may
  be visible to other users by means of system-monitoring commands.)

DBI uses default values for any connection parameters that remain unknown after all
information sources have been consulted. If the hostname is unspecified, it defaults to
localhost. If the username is unspecified, it defaults to your login name under Unix
and to ODBC under Windows. If the password is unspecified, there is no default; instead,
no password is sent.

• $dbh = DBI->connect_cached($data_source,
  $user_name,
  $password
  [, \%attr]);

This method is like connect(), except that DBI caches the database handle internally. If
a subsequent call is made to connect_cached() with the same connection parameters
while the connection is still active, DBI returns the cached handle rather than opening
a new connection. If the cached handle is no longer valid, DBI establishes a new
connection, and then caches and returns the new handle.

• @ary = DBI->data_sources($driver_name [, \%attr]);

  Returns a list of data sources available through the named driver. For MySQL, the
  $driver_name value is "mysql" (it must be lowercase). If $driver_name is undef or
  the empty string, DBI checks the value of the DBI_DRIVER environment variable to
  get the driver name. You can use the optional \%attr parameter to supply connection
  parameters.

  For many DBI drivers, data_sources() returns an empty or incomplete list.

• $drh = DBI->install_driver($driver_name);

  Activates a DBD-level driver and returns a driver handle for it, or dies with an error
  message if the driver cannot be found. For MySQL, the $driver_name value is "mysql"
  (it must be lowercase). Normally, it is not necessary to use this method because DBI
  activates the proper driver automatically when you invoke the connect() method.
  However, install_driver() may be helpful if you’re using the func() method to
  perform administrative operations. (See Section H.2.5, “MySQL-Specific Administrative
  Methods.”)

• %drivers = DBI->installed_drivers();

  Returns a hash of driver name/driver handle pairs for the drivers loaded into the current
  process.
H.2.2 Database-Handle Methods

The methods in this section are invoked through a database handle and may be used after you have obtained such a handle by calling the connect(), connect_cached(), or clone() method.

The $attr parameter for methods in this section may be used to specify method-processing attributes. (An attribute parameter of undef means “no attributes.”) For MySQL, the most important of these attributes are PrintError and RaiseError. For example, if RaiseError currently is disabled, you can enable it while processing a particular statement to cause automatic script termination if a DBI error occurs:

```
$rows = $dbh->do ($stmt, {RaiseError => 1});
```

PrintError and RaiseError are discussed in Section H.4, “DBI Attributes.”

- $rc = $dbh->begin_work ();
  
  Turns off autocommit mode by disabling the AutoCommit database-handle attribute. This enables a transaction to be performed. AutoCommit remains disabled until the next call to commit() or rollback(), after which it becomes enabled again. Use of begin_work() differs from disabling the AutoCommit attribute manually; in the latter case, you must also re-enable AutoCommit manually after committing or rolling back.

  begin_work() returns true if AutoCommit was disabled successfully, or false if it was already disabled.

- $dbh2 = $dbh->clone ($\%attr);
  
  Duplicates the existing connection $dbh and returns a new database handle. The new connection is made with the same parameters used for the original one. Any attributes given are added to the original attributes. This replaces any original attributes that have the same names.

- $rc = $dbh->commit ();
  
  Commits the current transaction if AutoCommit is disabled. Otherwise, invoking commit() has no effect and results in a warning.

- $rc = $dbh->disconnect ();
  
  Terminates the connection associated with the database handle. If the connection is still active when the script exits, DBI terminates it automatically but issues a warning.

  The behavior of disconnect() for DBI is undefined with respect to active transactions. For MySQL, the server rolls back any transaction that is active if you disconnect without committing. For portability, terminate any active transaction explicitly by invoking commit() or rollback() before calling disconnect().

- $rows = $dbh->do ($stmt
  [, \%attr
  [, @bind_values]]);
Prepares and executes the statement indicated by $stmt. The return value is the number of rows affected, −1 if the number of rows is unknown, and undef if an error occurred. If the number of rows affected is zero, the return value is the string "0E0", which evaluates as zero in numeric contexts but is considered true in boolean contexts.

do() is used primarily for statements that do not retrieve rows, such as DELETE, INSERT, or UPDATE. Trying to use do() for a SELECT statement is ineffective; you don’t get back a statement handle, so you won’t be able to fetch any rows.

Normally, no attributes are passed to do(), so the %attr parameter can be specified as undef. @bind_values represents a list of values to be bound to placeholders, which are indicated by '?\' characters within the statement string.

If a statement includes no placeholders, you can omit both the %attr parameter and the value list:

```
$rows = $dbh->do (  
    "UPDATE member SET expiration = NOW() WHERE member_id = 39"
);
```

If the statement does contain placeholders, the list must contain as many values as there are placeholders, and must be preceded by the %attr argument. In the following example, the attribute argument is undef and is followed by two data values to be bound to the two placeholders in the statement string:

```
$rows = $dbh->do ("UPDATE member SET expiration = ? WHERE member_id = ?",  
    undef,  
    "2007-11-30", 39);
```

- $rv = $dbh->get_info ($info_type);

Returns a characteristic of the DBI or driver implementation.

```
my $version = $dbh->get_info (18); # get database version
```

For information about the permitted information types, consult the DBI documentation.

- $rc = $dbh->ping ()

Re-establishes the connection to the server if the connection has timed out. Returns true if the connection was still active or was re-established successfully, and false otherwise.

- $sth = $dbh->prepare ($stmt [, \%attr]);

Prepares the statement indicated by $stmt for later execution and returns a statement handle, or undef if an error occurs. The statement handle returned from a successful invocation of prepare() may be used with execute() to execute the statement.

- $sth = $dbh->prepare_cached ($stmt  
    [, \%attr  
    [, $if_active]]);

This method is like prepare(), except that DBI caches the statement handle internally. If a subsequent call is made to prepare_cached() with the same $stmt and $attr
arguments, DBI returns the cached handle rather than creating a new one. The
$if_active argument determines how this method behaves if the cached handle is still
active. If this argument is missing or has a value of 0, DBI calls finish() and issues a
warning before returning the handle. If $if_active is 1, DBI calls finish() but issues
no warning. If $if_active is 2, DBI does not check whether the handle is active. If
$if_active is 3, DBI removes the cached active handle from the cache and prepares and
caches a new handle. This leaves the existing handle unchanged but no longer cached.

- $str = $dbh->quote ($value [, $data_type]);

Processes a string to perform quoting and escaping of characters that are special in SQL.
The resulting string may be used as a data value in a statement without causing a syntax
error when you execute the statement. For example, the string I'm happy is returned
as 'I\'m happy'. If $value is undef, it is returned as the literal word NULL. The return
value includes surrounding quote characters as necessary, so do not add extra quotes
around it when you insert the value into a statement string.

For values that you are going to insert into a statement using placeholders, do not use
quote(). DBI quotes such values automatically.

The $data_type parameter usually is unnecessary because MySQL converts string values
in statements to other data types as necessary. $data_type may be specified as a hint
about the value type. For example, DBI::SQL_INTEGER indicates that $value represents
an integer.

- $str = $dbh->quote_identifier ($name [, $name, ... [, \$attr ]]);

Treats the given name as an identifier and returns it as a quoted identifier. For example,
abc becomes `abc` and a`c becomes `a``c`. If you specify multiple arguments,
quote_identifier() quotes each one and joins them with periods in between.
This enables construction of quoted qualified identifiers. For example, quote_identifier('db','tbl','col')
becomes `db`.`tbl`.`col`.

quote_identifier() serves the same function as quote(), but for identifiers such as
database, table, column, index, and alias names rather than for data values. This method
is useful for constructing statements that refer to identifiers containing spaces or other
characters that normally are illegal in names. For example, a table named my table
cannot be used as follows in a statement, because the name contains a space:

SELECT * FROM my table

In MySQL, you can quote the name by enclosing it within backticks:

SELECT * FROM `my table`

To construct this statement in DBI, use quote_identifier():

$stmt = "SELECT * FROM * . $dbh->quote_identifier ("my table");

- $rc = $dbh->rollback();

Rolls back the current transaction if AutoCommit is disabled. Otherwise, invoking
rollback() has no effect and results in a warning.
• `$ary_ref = $dbh->selectall_arrayref ($stmt
  [, %attr
  [, @bind_values]]);`

Combines the effect of `prepare()`, `execute()`, and `fetchall_arrayref()` to execute the statement specified by `$stmt`. If `$stmt` is a handle to a previously prepared statement, the `prepare()` step is omitted. The `%attr` and `@bind_values` parameters have similar meanings as for the `do()` method.

The return value is a reference to an array. Each array element is a reference to an array containing the values for one row of the result set. The array is empty if the result set contains no rows.

If an error occurred, `selectall_arrayref()` returns `undef` unless a partial result set already has been fetched. In that case, it returns the rows retrieved to that point. To determine whether a non-`undef` return value represents success or failure, check `$dbh->err()` or `$DBI::err`.

• `$hash_ref = $dbh->selectall_hashref ($stmt,
  $key_col
  [, %attr
  [, @bind_values]]);`

Combines the effect of `prepare()`, `execute()`, and `fetchall_hashref()` to execute the statement specified by `$stmt`. If `$stmt` is a handle to a previously prepared statement, the `prepare()` step is omitted. The `%attr` and `@bind_values` parameters have the same meaning as for the `do()` method.

The return value is a reference to a hash that contains one element for each row of the result set. Hash keys are the values of the column indicated by `$key_col`, which should be either the name of a column selected by the statement, or a column number. Column values begin with 1. Values in the key column should be unique to avoid loss of rows due to key collisions in the hash. The hash is empty if the result set contains no rows. Otherwise, the value of each hash element is a reference to a hash containing one row of the result set, keyed by the names of the columns selected by the statement.

If an error occurred, `selectall_hashref()` returns `undef` unless a partial result set already has been fetched. In that case, it returns the rows retrieved to that point. To determine whether a non-`undef` return value represents success or failure, check `$dbh->err()` or `$DBI::err`.

• `$ary_ref = $dbh->selectcol_arrayref ($stmt,
  [
  %attr
  [, @bind_values]]);`

Combines the effect of `prepare()`, `execute()`, and a row-fetching operation to execute the statement specified by `$stmt`. If `$stmt` is a handle to a previously prepared statement, the `prepare()` step is omitted. The `%attr` and `@bind_values` parameters have similar meanings as for the `do()` method.
The return value is a reference to an array containing the first column from each row. If an error occurred, `selectcol_arrayref()` returns `undef` unless a partial result set already has been fetched. In that case, it returns the rows retrieved to that point. To determine whether a non-`undef` return value represents success or failure, check `$dbh->err()` or `$DBI::err`.

- `@row_ary = $dbh->selectrow_array ($stmt [, %attr [, @bind_values]]);`

Combines the effect of `prepare()`, `execute()`, and `fetchrow_array()` to execute the statement specified by `$stmt`. If `$stmt` is a handle to a previously prepared statement, the `prepare()` step is omitted. The `%attr` and `@bind_values` parameters have the same meaning as for the `do()` method.

When called in a list context, `selectrow_array()` returns an array representing the values in the first row of the result set, or an empty array if no row was returned or an error occurred. In a scalar context, `selectrow_array()` returns one element of the array, or `undef` if no row was returned or if an error occurred. Which element is returned is undefined; see the note about this behavior in the entry for `fetchrow_array()`.

To distinguish between no row and an error in list context, check `$sth->err()` or `$DBI::err`. A value of zero indicates that no row was returned. However, in the absence of an error, an `undef` return value in scalar context may represent either a `NULL` column value or that no row was returned.

- `$ary_ref = $dbh->selectrow_arrayref ($stmt [, %attr [, @bind_values]]);`

Combines the effect of `prepare()`, `execute()`, and `fetchrow_arrayref()` to execute the statement specified by `$stmt`. If `$stmt` is a handle to a previously prepared statement, the `prepare()` step is omitted. The `%attr` and `@bind_values` parameters have the same meaning as for the `do()` method.

The return value is a reference to an array containing the values in the first row of the result set, or `undef` if an error occurred.

- `$hash_ref = $dbh->selectrow_hashref ($stmt [, %attr [, @bind_values]]);`

Combines the effect of `prepare()`, `execute()`, and `fetchrow_hashref()` to execute the statement specified by `$stmt`. If `$stmt` is a handle to a previously prepared statement, the `prepare()` step is omitted. The `%attr` and `@bind_values` parameters have the same meaning as for the `do()` method.

The return value is a reference to a hash containing the first row of the result set, or `undef` if an error occurred. The hash keys are the names of the columns selected by the statement.
A number of additional database-handle methods for getting database and table metadata have appeared in recent versions of DBI. These include `column_info()`, `foreign_key_info()`, `last_insert_id()`, `primary_key()`, `primary_key_info()`, `statistics_info()`, `table_info()`, `tables()`, `type_info()`, and `type_info_all()`. For more information about them, consult the DBI documentation.

The level of support for these methods varies among drivers, and some of them are experimental. For MySQL, you should try them with your version of DBD::mysql to see which are implemented and what information can be obtained.

### H.2.3 Statement-Handle Methods

The methods in this section are invoked through a statement handle, which you obtain by calling a method such as `prepare()` or `prepareCached()`.

- `$_rc = $sth->bind_col($col_num, \$var);`
  Binds a given output column from a `SELECT` statement to a Perl variable, which should be passed as a reference. `$col_num` should be in the range from 1 to the number of columns selected by the statement. Each time a row is fetched, the variable is updated automatically with the column value.
  
  Call `bind_col()` after `execute()` and before fetching rows.

  `bind_col()` returns false if the column number is not in the range from 1 to the number of columns selected by the statement.

- `$_rc = $sth->bind_columns(\$var1, \$var2, ...);`
  Binds a list of variables to columns returned by a prepared `SELECT` statement. See the description of the `bind_col()` method. Like `bind_col()`, call `bind_columns()` after `execute()` and before fetching rows.

  `bind_columns()` returns false if the number of arguments doesn’t match the number of columns selected by the statement.

- `$_rv = $sth->bind_param($n, $value [, \%attr]);`
  `$_rv = $sth->bind_param($n, $value [, $bind_type]);`
  Binds a value to a placeholder in a statement string so that the value is included in the statement when it is sent to the server. Placeholders are represented by ‘?’ characters in the statement string. This method should be called after `prepare()` and before `execute()`.

  $n specifies the number of the placeholder to which the value $value should be bound and should be in the range from 1 to the number of placeholders. To bind a NULL value, $value should be `undef`.

  The `\%attr` or `$bind_type` parameter may be supplied as a hint about the type of the value to be bound. The default is to treat the variable as a `VARCHAR`, so non-NULL values
are quoted when bound to the statement. This is normally sufficient because MySQL converts string values in statements to other data types as necessary, but can cause problems in some contexts. For example, any argument to a LIMIT clause must be an integer. To specify that a value represents an integer, invoke bind_param() either of the following ways:

```perl
$rv = $sth->bind_param ($n, $value, { TYPE => DBI::SQL_INTEGER });
$rv = $sth->bind_param ($n, $value, DBI::SQL_INTEGER);
```

This function is similar to bind_param(), except that it is intended for use with a prepared statement to be executed with execute_array(). The $values argument can be either a reference to an array of values, or a single scalar value. For an array reference, successive values in the array are used for successive executions of the statement. For a scalar, the value is re-used for each execution.

- $rows = $sth->dump_results ([{$maxlen
[, $line_sep
[, $field_sep
[, $fh]]]}]);

Fetches all rows from the statement handle $sth, formats them by calling the utility function DBI::neat_list(), and prints them to the given file handle. Returns the number of rows fetched.

The defaults for the $maxlen, $line_sep, $field_sep, and $fh parameters are 35, "\n", ",", and STDOUT, respectively.

- $rv = $sth->execute (@bind_values);

Executes a prepared statement. For SELECT statements, execute() returns true if the statement executed successfully, or undef if an error occurred. For non-SELECT statements, the return value is the number of rows affected, −1 if the number of rows is unknown, and undef if an error occurred. If the number of rows affected is zero, the return value is the string "0E0", which evaluates as zero in numeric contexts but is considered true in boolean contexts.

The @bind_values parameter has the same meaning as for the do() method.

- $rv = $sth->execute_array ($attr [, @bind_values]);

Executes a prepared statement multiple times. The number of executions is determined by the number of values passed via @bind_values, the values bound to the statement by earlier calls to bind_param_array(), or by the attribute reference.

- $ary_ref = $sth->fetch ();

fetch() is an alias for fetchrow_arrayref().
• $ary_ref = $sth->fetchall_arrayref ([[$slice_ref [, $max_rows]]]);

Fetches all rows from the statement handle $sth and returns a reference to an array that contains one reference for each row fetched. This array is empty if the result set contains no rows. Otherwise, each element of $ary_ref is a reference to one row of the result set. The meaning of the row references depends on the type of $slice_ref argument you pass. With no argument or an array slice argument, each row reference points to an array of column values. A nonempty array slice should contain array index numbers to select specific columns. Index numbers begin at 0 because they are Perl array indices. Negative values count back from the end of the row. Thus, to fetch the first and last columns of each row, do this:

$ary_ref = $sth->fetchall_arrayref ([0, -1]);

With a hash slice argument, each row reference points to a hash of column values, indexed by the names of the columns you want to retrieve. To specify a hash slice, column names should be given as hash keys and each key should have a value of 1:

$ary_ref = $sth->fetchall_arrayref ({id => 1, name => 1});

To fetch all columns as a hash, pass an empty hash reference:

$ary_ref = $sth->fetchall_arrayref ({});

The $max_rows argument can be given to limit the number of rows fetched. In this case, you can continue to call fetchall_arrayref() until it returns no more rows.

If an error occurred, fetchall_arrayref() returns the rows fetched up to the point of the error. Check $sth->err() or $DBI::err to determine whether an error occurred.

• $hash_ref = $sth->fetchall_hashref ($key_col);

Fetches the result set and returns a reference to a hash that contains one element for each row of the result set. Hash keys are the values of the column indicated by $key_col, which should be either the name of a column selected by the statement, or a column number. Column values begin with 1. Values in the key column should be unique to avoid loss of rows due to key collisions in the hash. The hash is empty if the result set contains no rows. Otherwise, the value of each hash element is a reference to a hash containing one row of the result set, keyed by the names of the columns selected by the statement.

If an error occurred due to an invalid key column argument, fetchall_hashref() returns undef. Otherwise, it returns the rows fetched up to the point of the error. To determine whether a non-undef return value represents success or failure, check $sth->err() or $DBI::err.

• @ary = $sth->fetchrow_array();

When called in a list context, fetchrow_array() returns an array containing column values for the next row of the result set, or an empty array if there are no more rows or an error occurred. To distinguish between normal exhaustion of the result set and an
error, check $sth->err() or $DBI::err. A value of zero indicates that you've reached
the end of the result set without error.

In a scalar context, fetchrow_array() returns one element of the array, or undef if
there are no more rows or an error occurred. However, it is undefined which element is
returned; you can tell for sure only for statements that select a single column. Also, an
undef return value in the absence of an error is ambiguous; it may represent either a
NULL column value or the end of the result set.

$ary_ref = $sth->fetchrow_arrayref();

Returns a reference to an array containing column values for the next row of the result
set, or undef if there are no more rows or an error occurred.

To distinguish between normal exhaustion of the result set and an error,
check $sth->err() or $DBI::err. A value of zero indicates that you've reached the end
of the result set without error.

$hash_ref = $sth->fetchrow_hashref([$name]);

Returns a reference to a hash containing column values for the next row of the result
set, or undef if there are no more rows or an error occurred. Hash index values are the
column names, and elements of the hash are the column values.

The $name argument may be specified to control hash key lettercase. It defaults to
"NAME" (use column names as specified in the statement). To force hash keys to be
lowercase or uppercase, you can specify a $name value of "NAME_lc" or "NAME_uc"
instead. (Another way to control hash key letter case is with the FetchHashKeyName
attribute, which is discussed in Section H.4, "DBI Attributes.")

To distinguish between normal exhaustion of the result set and an error, check
$sth->err() or $DBI::err. A value of zero indicates that you've reached the end of the
result set without error.

$rc = $sth->finish();

Frees any resources associated with the statement handle. Normally, you need not invoke
this method yourself, because row-fetching methods invoke it implicitly when they reach
the end of the result set. If you fetch only part of a result set, calling finish() explicitly
lets DBI know that you are done fetching data from the handle.

finish() invalidates statement attributes, and because this method may be invoked
implicitly by row-fetching methods when they detect the end of the result set, it's best
to access any attributes you need immediately after invoking execute(), rather than
waiting until later.

$r = $sth->rows();

Returns the number of rows affected by the statement associated with $sth, or -1 if an
error occurred. This method is used primarily for statements such as UPDATE or DELETE
that do not return rows. For SELECT statements, you should not rely on the rows() method. Instead, count the rows as you fetch them.
H.2.4 General Handle Methods

The methods in this section are not specific to particular types of handles. They may be invoked using driver, database, or statement handles.

- `$rv = $h->err();`
  
  Returns the numeric error code for the most recently invoked driver operation. For MySQL, this is the error number returned by the MySQL server. A return value of 0 or `undef` indicates that no error occurred. An empty string as the return value means “success with information,” in which case, `errstr()` returns the additional information.

- `$str = $h->errstr();`
  
  Returns the string error message for the most recently invoked driver operation. For MySQL, this is the error message returned by the MySQL server. A return value of the empty string or `undef` indicates that no error occurred.

- `DBI->trace ($trace_level [, $trace_filename]);`
  
  Sets a trace level. Tracing provides information about DBI operation. The trace level can be in the range from 0 (off) to 9 (maximum information). Tracing can be enabled for all DBI operations within a script by invoking trace as a DBI class method, or for an individual handle:

  ```perl
  DBI->trace (2);     # Turn on global script tracing
  $sth->trace (2);    # Turn on per-handle tracing
  ```

  To enable tracing on a global level for all DBI scripts that you run, set the `DBI_TRACE` environment variable.

  Trace output goes to `STDERR` by default. To direct output to a different file, the `$filename` parameter may be supplied. Output is appended to any existing contents of the file; the file is not overwritten. The special filenames `STDOUT` and `STDERR` are understood to stand for the standard output and standard error output, respectively, which have their conventional meanings.

  Each trace call causes output from all traced handles to go to the same file. If the call names a file, all trace output goes there. Otherwise, all trace output goes to `STDERR`.

- `DBI->trace_msg ($str [, $min_level]);`
  
  When called as a class method (`DBI->trace_msg()`), writes the message in `$str` to the trace output if tracing has been enabled at the DBI level. When called as a handle method (`$h->trace_msg()`), writes the message if the handle is being traced or if tracing has been enabled at the DBI level.

  The `$min_level` parameter may be supplied to specify that the message should be written only if the trace level is at least at that level.
**H.2.5 MySQL-Specific Administrative Methods**

This section describes the `func()` method that DBI provides as a means of accessing driver-specific operations directly. Note that `func()` is not related to the use of stored functions. Stored function methods currently are not defined by DBI.

- `rc = $drh->func("createdb", $db_name, $host_name, $user_name, $password, "admin");`
- `rc = $drh->func("dropdb", $db_name, $host_name, $user_name, $password, "admin");`
- `rc = $drh->func("reload", $host_name, $user_name, $password, "admin");`
- `rc = $drh->func("shutdown", $host_name, $user_name, $password, "admin");`

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

The `func()` method is accessed either through a driver handle or through a database handle. A driver handle is not associated with an open connection, so if you access `func()` that way, you must supply arguments for the hostname, username, and password to enable the method to establish a connection. If you access `func()` with a database handle, those arguments are unnecessary. A driver handle may be obtained, if necessary, as follows:

```
$drh = DBI->install_driver("mysql");  # "mysql" must be lowercase
```

`func()` understands the following actions:

- **createdb**
  Creates the database named by `$db_name`. You must have the `CREATE` privilege for the database.

- **dropdb**
  Drops (removes) the database named by `$db_name`. You must have the `DROP` privilege for the database.

- **reload**
  Tells the server to reload the grant tables. This is necessary if you modify the contents of the grant tables directly using statements such as `DELETE`, `INSERT`, or `UPDATE` rather than using `GRANT` or `REVOKE`. You must have the `RELOAD` privilege.

- **shutdown**
  Shuts down the server. You must have the `SHUTDOWN` privilege.
Note that the only func() action that cannot be performed through the usual DBI statement-processing mechanism is shutdown. For the other actions, it is preferable to issue a CREATE DATABASE, DROP DATABASE, or FLUSH PRIVILEGES statement rather than invoking func().

H.3 DBI Utility Functions

DBI provides a few utility routines that can be used for testing or printing values. These functions are invoked as DBI::func_name() rather than as DBI->func_name().

- @bool = DBI::looks_like_number (@ary);
  Takes a list of values and returns an array with one member for each element of the list. Each member indicates whether the corresponding argument looks like a number: true if it does, false if it doesn’t, and undef if the argument is undefined or empty.

- $str = DBI::neat ($value [, $maxlen]);
  Returns a string containing a nicely formatted representation of the $value argument. Strings are returned with surrounding quotes; numbers are not. (But note that quoted numbers are considered to be strings.) Undefined values are reported as undef, and unprintable characters are reported as ’.’ characters. For example, if you execute the following loop:
  
  for my $val (“a”, “3”, 3, undef, “\x01\x02")
  {
    print DBI::neat ($val), “\n”;}
  
  The results look like this:
  
  ’a’
  ’3’
  3
  undef
  ’.’

  The $maxlen argument controls the maximum length of the result. If the result is longer than $maxlen, it is shortened to $maxlen-4 characters and “…” is added. If $maxlen is 0, undef, or missing, it defaults to the current value of $DBI::neat_maxlen, which itself has a default value of 400 (1000 as of DBI 1.605).

  Don’t use neat() for statement construction; if you need to perform quoting or escaping of data values to be placed into a statement string, use placeholders or the quote() method instead.

- $str = DBI::neat_list (@ary
  [, $maxlen
  [, $sep]]);
Calls neat() for each element of the list pointed to by the first argument, joins them with the separator string $sep, and returns the result as a single string.

The $maxlen argument is passed to neat() and thus applies to individual arguments, not to the combined result of the neat() calls.

If $sep is missing, the default is "",.

H.4   DBI Attributes

DBI provides attribute information at several levels. Most attributes are associated with database handles or statement handles, but not with both. Some attributes, such as PrintError and RaiseError, may be associated with either database handles or statement handles. In general, each handle has its own attributes, but some attributes that hold error information, such as err and errstr, are dynamic in that they associate with the most recently used handle.

Attributes passed to connect() or connect_cached() become part of the resulting database handle returned by those methods.

H.4.1   Database-Handle Attributes

The attributes in this section are associated with database handles.

- $dbh->{AutoCommit};
  
  This attribute can be set to true or false to enable or disable transaction autocommit mode. The default is true. Setting AutoCommit to false enables transactions to be performed, each of which is terminated by calling commit() for a successful transaction or rollback() to abort an unsuccessful transaction. See also the description of the begin_work() database-handle method.

- $dbh->{Statement};
  
  Holds the statement string most recently passed to prepare() through the handle.

H.4.2   General Handle Attributes

These attributes may be applied to individual handles or specified in the $attr parameter to methods that take such a parameter to affect the operation of the method.

- $h->{ChopBlanks};
  
  This attribute can be set to true or false to determine whether row-fetching methods chop trailing spaces from character column values. ChopBlanks is false by default for most database drivers.
$h->{FetchHashKeyName};

Controls the lettercase used for hash keys in result set rows returned by
fetchrow_hashref() or other methods that invoke fetchrow_hashref(). The default
is "NAME" (use column names as specified in the SELECT statement). Other permitted
values are "NAME_lc" or "NAME_uc", which cause column name hash keys to be forced
to lowercase or uppercase. This attribute applies only to database and driver handles.

$h->{HandleError};

This attribute is used for error processing. It can be set to a reference to a subroutine
to be invoked when an error occurs, prior to the usual RaiseError and PrintError
processing. If the subroutine returns true, RaiseError and PrintError processing is
skipped; otherwise, it is performed as usual. (The error routine can of course terminate
the script rather than returning.)

DBI passes three arguments to the error routine: The text of the error message, the DBI
handle being used at the point of occurrence of the error, and the first value returned by
the method that failed.

$h->{PrintError};

If set true, the occurrence of a DBI-related error causes a warning message to be printed.
PrintError is false by default. This attribute does not affect the value returned by
DBI methods when they fail. It determines only whether they print a message before
returning.

$h->{RaiseError};

If set to true, the occurrence of a DBI-related error causes an exception to be raised.
Normally this causes the script to terminate unless it arranges to catch the exception.
RaiseError is false by default.

$h->{ShowErrorStatement};

When set to true, messages produced as a result of errors have the relevant statement text
appended to them. ShowErrorStatement is false by default. The effect of this attribute is
limited to statement handles and to the prepare() and do() methods.

$h->{TraceLevel};

Sets or gets the trace level for the given handle. This attribute provides an alternative to
the trace() method.

H.4.3 MySQL-Specific Database-Handle Attributes

These attributes are specific to DBD::mysql, the DBI MySQL driver. Most of them correspond to
a function in the MySQL C API, as indicated in the attribute descriptions. For more information
about the C functions, see Appendix G, "C API Reference."
$rv = $dbh->{mysql_auto_reconnect};

Whether the driver automatically reconnects to the server after the connection goes down. Normally, auto-reconnect is disabled by default, but will be enabled if the GATEWAY_INTERFACE or MOD_PERL environment variable is set. If AutoCommit is disabled, the mysql_auto_reconnect setting is ignored and no reconnects are attempted.

$hash_ref = $dbh->{mysql_dbd_stats};

A hash reference containing driver statistics. Currently this hash has two keys, auto_reconnects_ok and auto_reconnects_failed, indicating the number of times the driver tried successfully and unsuccessfully to reconnect to the server.

$rv = $dbh->{mysql_errno};

The most recent error number, like the mysql_errno() C API function.

$str = $dbh->{mysql_error};

The most recent error string, like the mysql_error() C API function.

$str = $dbh->{mysql_hostinfo};

A string describing the given connection, like the mysql_get_host_info() C API function.

$str = $dbh->{mysql_info};

Information about statements that affect multiple rows, like the mysql_info() C API function.

$rv = $dbh->{mysql_insertid};

The AUTO_INCREMENT value that was most recently generated on the connection associated with $dbh, like the mysql_insert_id() C API function.

$rv = $dbh->{mysql_protoinfo};

A number indicating the client/server protocol version used for the given connection, like the mysql_get_proto_info() C API function.

$rv = $dbh->{mysql_server_prepare};

True if server-side statement preparation is enabled; false if statement preparation is emulated on the client side. Assign to this attribute to enable or disable server-side prepared statement execution for statement handles created from $dbh:

$dbh->{mysql_server_prepare} = 1; # enable server-side preparation
$dbh->{mysql_server_prepare} = 0; # disable server-side preparation

$str = $dbh->{mysql_serverinfo};

A string describing the server version; for example, "5.5.30-debug-log". The value consists of a version number, possibly followed by one or more suffixes. This attribute returns the same information as the mysql_get_server_info() C API function or VERSION() SQL function.
Appendix H  Perl DBI API Reference

- `$str = $dbh->{mysql_stat};`
  A string containing server status information, like the `mysql_stat()` C API function.
- `$rv = $dbh->{mysql_thread_id};`
  The connection ID for the connection associated with `$dbh`, like the `mysql_thread_id()` C API function or `CONNECTION_ID()` SQL function.
- `$rv = $dbh->{mysql_use_result};`
  Whether to use the `mysql_store_result()` or `mysql_use_result()` C API function for retrieving result sets. For more information, see the description of the corresponding statement-handle attribute in Section H.4.5, “MySQL-Specific Statement-Handle Attributes.”

H.4.4 Statement-Handle Attributes

Statement-handle attributes generally apply to `SELECT` (or `SELECT-like`) statements and are not valid until the statement has been passed to `prepare()` to obtain a statement handle and `execute()` has been called for that handle. In addition, `finish()` may invalidate statement attributes; in general, it is not safe to access them after invoking `finish()`, or after reaching the end of a result set, which causes implicit `finish()` invocation.

Many statement-handle attributes have a value that is a reference to an array of values, one value per column selected by the statement. The `$sth->{NUM_OF_FIELDS}` attribute indicates the number of elements in the array. For a statement attribute `stmt_attr` that is a reference to an array, you can refer to the entire array as `@$sth->{stmt_attr}`, or loop through the elements in the array like this:

```perl
for (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)
    { my $value = $sth->{stmt_attr}->[$i]; }
```

The `NAME_hash`, `NAME_lc_hash`, and `NAME_uc_hash` attributes return a reference to a hash. You can loop through the hash elements like this:

```perl
foreach my $key (keys %{$sth->{stmt_attr}})
    { my $value = $sth->{stmt_attr}->{$key}; }
```

- `$ary_ref = $sth->{NAME};`
  A reference to an array of strings indicating the name of each column. The lettercase of the names is as specified in the `SELECT` statement.
$ary_ref = $sth->{NAME_hash};

A reference to a hash of strings indicating the name of each column. The lettercase of the names is as specified in the SELECT statement. The value of each hash element indicates the position of the corresponding column within result set rows (beginning with 0).

$ary_ref = $sth->{NAME_lc};

Like NAME, but the names are lowercase.

$ary_ref = $sth->{NAME_lc_hash};

Like NAME_hash, but the names are lowercase.

$ary_ref = $sth->{NAME_uc};

Like NAME, but the names are uppercase.

$ary_ref = $sth->{NAME_uc_hash};

Like NAME_hash, but the names are uppercase.

$ary_ref = $sth->{NULLABLE};

A reference to an array of values indicating whether each column can be NULL. Values for each element can be 0 or an empty string (no), 1 (yes), or 2 (unknown).

$rv = $sth->{NUM_OF_FIELDS};

The number of columns in a result set, or zero for a non-SELECT statement.

$rv = $sth->{NUM_OF_PARAMS};

The number of placeholders in a prepared statement.

$ary_ref = $sth->{PRECISION};

A reference to an array of values indicating the precision of each column. DBI uses “precision” in the ODBC sense, which for MySQL means the maximum width of the column. For numeric columns, this is the display width. For string columns, it’s the maximum length of the column, in octets (bytes), not characters.

$ary_ref = $sth->{SCALE};

A reference to an array of values indicating the scale of each column. DBI uses “scale” in the ODBC sense, which for MySQL means the number of decimal places for floating-point columns. For other columns where scale is not applicable, the scale is undef.

$str = $sth->{Statement};

The text of the statement associated with $sth, as seen by prepare() before any placeholder substitution takes place.

$ary_ref = $sth->{TYPE};

A reference to an array of values indicating the numeric type of each column. Values for this attribute are portable among DBD-level drivers. To obtain MySQL-specific type numbers, access the mysql_type attribute.
H.4.5 MySQL-Specific Statement-Handle Attributes

These attributes are specific to DBD::mysql, the DBI MySQL driver. Most of them should be considered read only and should be accessed after invoking execute(). The exception is the mysql_use_result attribute, which should be set after prepare() but before execute(). See the mysql_use_result description for an example.

- `$rv = $sth->{mysql_insertid};`
  The AUTO_INCREMENT value most recently generated on the connection associated with `$sth`.

- `$ary_ref = $sth->{mysql_is_auto_increment};`
  A reference to an array of values indicating whether each column is an AUTO_INCREMENT column.

- `$ary_ref = $sth->{mysql_is_blob};`
  A reference to an array of values indicating whether each column is a BLOB type. Values in this array are true for the TEXT types as well.

- `$ary_ref = $sth->{mysql_is_key};`
  A reference to an array of values indicating whether each column is part of a key.

- `$ary_ref = $sth->{mysql_is_num};`
  A reference to an array of values indicating whether each column is a numeric type.

- `$ary_ref = $sth->{mysql_is_pri_key};`
  A reference to an array of values indicating whether each column is part of a PRIMARY KEY.

- `$ary_ref = $sth->{mysql_length};`
  This is like the PRECISION attribute.

- `$ary_ref = $sth->{mysql_max_length};`
  A reference to an array of values indicating the actual maximum length of the values in each column of the result set.

- `$rv = $sth->{mysql_server_prepare};`
  True if server-side statement preparation is enabled; false if statement preparation is emulated on the client side.

- `$ary_ref = $sth->{mysql_table};`
  A reference to an array of values indicating the name of the table containing each column. The table name for a calculated column is the empty string.
H.4 DBI Attributes

- \$ary_ref = \$sth->{mysql_type};
  A reference to an array of values indicating the MySQL-specific type number for each column.

- \$ary_ref = \$sth->{mysql_type_name};
  A reference to an array of values indicating the MySQL-specific type name for each column.

- \$rv = \$sth->{mysql_use_result};
  This attribute affects which C API function DBD::mysql uses to retrieve result sets. By default, mysql_use_result is 0, so DBI::mysql uses mysql_store_result(). To use mysql_use_result() instead, set mysql_use_result to 1. For a discussion of these two functions and how they differ, see Appendix G, “C API Reference.”

  Note that enabling mysql_use_result causes some statement-handle attributes such as mysql_max_length to become invalid. It also invalidates use of the rows() method, although it’s better to count rows when you fetch them anyway.

  If you set the mysql_use_result attribute, do so after invoking prepare() and before invoking execute():
  ```perl
  \$sth = \$dbh->prepare (\$stmt_str);
  \$sth->{mysql_use_result} = 1;
  \$sth->execute();
  ```

  Alternatively, do this:
  ```perl
  \$sth = \$dbh->prepare (\$stmt_str, \{ mysql_use_result => 1 \});
  ```

- \$rv = \$sth->{mysql_warning_count};
  The number of warnings generated during execution of the statement.

H.4.6 Dynamic Attributes

These attributes are associated with the most recently used handle, represented by \$h in the following descriptions. They should be used immediately after invoking whatever handle method sets them, and before invoking another method that resets them.

- \$rv = \$DBI::err;
  This is the same as calling \$h->err().

- \$str = \$DBI::errstr;
  This is the same as calling \$h->errstr().

- \$rows = \$DBI::rows;
  This is the same as calling \$h->rows().
H.5  DBI Environment Variables

DBI consults several environment variables, listed in Table H.2. The `connect()` method uses all of them except `DBI_TRACE`. The `data_sources()` method uses `DBI_DRIVER` and `trace()` uses `DBI_TRACE`.

Table H.2  DBI Environment Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DBI_DRIVER</code></td>
<td>DBD-level driver name (&quot;mysql&quot; for MySQL)</td>
</tr>
<tr>
<td><code>DBI_DSN</code></td>
<td>Data source name</td>
</tr>
<tr>
<td><code>DBI_PASS</code></td>
<td>Password</td>
</tr>
<tr>
<td><code>DBI_TRACE</code></td>
<td>Trace level and/or trace output file</td>
</tr>
<tr>
<td><code>DBI_USER</code></td>
<td>Username</td>
</tr>
</tbody>
</table>