

## NAME

DBI - Database independent interface for Perl

## SYNOPSIS

```
use DBI;

@driver_names = DBI->available_drivers;
@data_sources = DBI->data_sources($driver_name);

$dbh = DBI->connect($data_source, $username, $auth, \%attr);

$rv = $dbh->do($statement);
$rv = $dbh->do($statement, \%attr);
$rv = $dbh->do($statement, \%attr, @bind_values);

$array_ref = $dbh->selectall_arrayref($statement);
@row_ary = $dbh->selectrow_array($statement);
$array_ref = $dbh->selectcol_arrayref($statement);

$sth = $dbh->prepare($statement);
$sth = $dbh->prepare_cached($statement);

$rv = $sth->bind_param($p_num, $bind_value);
$rv = $sth->bind_param($p_num, $bind_value, $bind_type);
$rv = $sth->bind_param($p_num, $bind_value, \%attr);

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

$rc = $sth->bind_col($col_num, \$col_variable);
$rc = $sth->bind_columns(@list_of_refs_to_vars_to_bind);

@row_ary = $sth->fetchrow_array;
$array_ref = $sth->fetchrow_arrayref;
$hash_ref = $sth->fetchrow_hashref;

$array_ref = $sth->fetchall_arrayref;

$rv = $sth->rows;

$rc = $dbh->commit;
$rc = $dbh->rollback;

$sql = $dbh->quote($string);

$rc = $h->err;
$str = $h->errstr;
$rv = $h->state;

$rc = $dbh->disconnect;
```

## NOTE

This is the DBI specification that corresponds to the DBI version 1.13 (\$Date: 1999/07/12 02:02:33 \$).

The DBI specification is evolving at a steady pace so it's important to check that you have the latest copy. The RECENT CHANGES section below has a summary of user-visible changes and the Changes file supplied with the DBI holds more detailed change information.

Note also that whenever the DBI changes the drivers take some time to catch up. Recent versions of the DBI have added new features (marked \*NEW\* in the text) that may not yet be supported by the drivers you use. Talk to the authors of those drivers if you need the features.

Please also read the DBI FAQ which is installed as a DBI::FAQ module so you can use perldoc to read it by executing the 'perldoc DBI::FAQ' command.

Extensions to the DBI and other DBI related modules use the 'DBIx::\*' namespace. See the Naming Conventions and Name Space entry elsewhere in this document and:

<http://www.perl.com/CPAN/modules/by-module/DBIx/>

#### RECENT CHANGES

A brief summary of significant user-visible changes in recent versions (if a recent version isn't mentioned it simply means that there were no significant user-visible changes in that version).

Between DBI 1.00 and DBI 1.09

Added \$dbh->selectcol\_arrayref(\$statement) method.

Connect now allows you to specify attribute settings within the DSN  
E.g., "dbi:Driver(RaiseError=>1,Taint=>1,AutoCommit=>0):dbname"

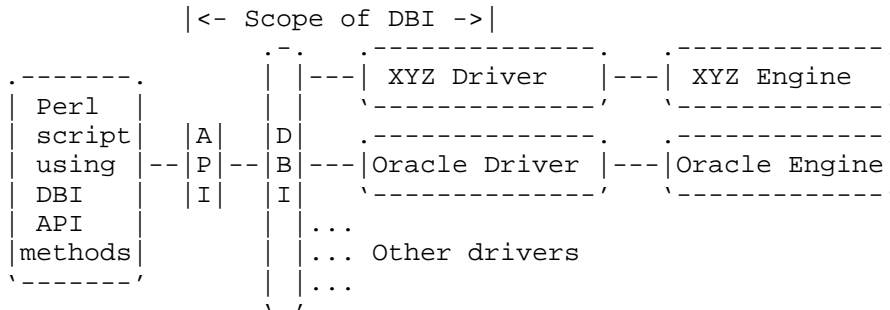
Added \$h->{Taint}, \$sth->{NAME\_uc} and \$sth->{NAME\_lc} attributes.

#### DESCRIPTION

The DBI is a database access module for the Perl Language. It defines a set of methods, variables and conventions that provide a consistent database interface independant of the actual database being used.

It is important to remember that the DBI is just an interface. A layer of 'glue' between an application and one or more \*Database Driver\* modules. It is the driver modules which do the real work. The DBI provides a standard interface and framework for the drivers to operate within.

#### Architecture of a DBI Application



The API is the Application Perl-script (or Programming) Interface. The call interface and variables provided by DBI to perl scripts. The API is implemented by the DBI Perl extension.

The DBI 'dispatches' the method calls to the appropriate Driver for actual execution. The DBI is also responsible for the dynamic loading of Drivers, error checking/handling and other duties.

The Drivers implement support for a given type of Engine (database). Drivers contain implementations of the DBI methods written using the private interface functions of the corresponding Engine. Only authors of sophisticated/multi-database applications or generic library functions need be concerned with Drivers.

#### Notation and Conventions

DBI        static 'top-level' class name

`$dbh` Database handle object  
`$sth` Statement handle object  
`$drh` Driver handle object (rarely seen or used in applications)  
`$h` Any of the `$$$h` handle types above  
`$rc` General Return Code (boolean: true=ok, false=error)  
`$rv` General Return Value (typically an integer)  
`@ary` List of values returned from the database, typically a row of data  
`$rows` Number of rows processed (if available, else -1)  
`$fh` A filehandle  
`undef` NULL values are represented by undefined values in perl  
`\%attr` Reference to a hash of attribute values passed to methods

Note that Perl will automatically destroy database and statement objects if all references to them are deleted.

## Outline Usage

First you need to load the DBI module:

```
use DBI;
```

(also adding `'use strict;'` is recommended). Then you need to the connect entry elsewhere in this document to your data source and get a `*handle*` for that connection:

```
$dbh = DBI->connect($dsn, $user, $password,
                   { RaiseError => 1, AutoCommit => 0 });
```

Since connecting can be expensive you generally just connect at the start of your program and disconnect at the end.

Explicitly defining the required `AutoCommit` behaviour is strongly recommended and may become mandatory in a later version.

The DBI allows an application to `'prepare'` statements for later execution. A prepared statement is identified by a statement handle. We'll call the perl variable `$sth`.

Typical method call sequence for a select statement:

```
prepare,
execute, fetch, fetch, ...
execute, fetch, fetch, ...
execute, fetch, fetch, ...
```

for example:

```
$sth = $dbh->prepare("select foo, bar from table where baz=?");

$sth->execute( $baz );

while ( @row = $sth->fetchrow_array ) {
    print "@row\n";
}
```

Typical method call sequence for a non-select statement:

```
prepare,
execute,
execute,
execute.
```

for example:

```
$sth = $dbh->prepare("insert into table(foo,bar,baz) values (?,?,?);");
```

```

while(<CSV>) {
    chop;
    my ($foo,$bar,$baz) = split /,/;
    $sth->execute( $foo, $bar, $baz );
}

```

The 'do()' method can be used for non repeated non-select statement (or with drivers that don't support placeholders):

```
$rows_affected = $dbh->do("update your_table set foo = foo + 1");
```

To commit your changes to the database (when the AutoCommit entry elsewhere in this document is off):

```
$dbh->commit; # or call $dbh->rollback; to undo changes
```

Finally, when you have finished working with the data source you should the disconnect entry elsewhere in this document from it:

```
$dbh->disconnect;
```

### General Interface Rules & Caveats

The DBI does not have a concept of a 'current session'. Every session has a handle object (i.e., a \$dbh) returned from the connect method and that handle object is used to invoke database related methods.

Most data is returned to the perl script as strings (null values are returned as undef). This allows arbitrary precision numeric data to be handled without loss of accuracy. Be aware that perl may not preserve the same accuracy when the string is used as a number.

Dates and times are returned as character strings in the native format of the corresponding Engine. Time Zone effects are Engine/Driver dependent.

Perl supports binary data in perl strings and the DBI will pass binary data to and from the Driver without change. It is up to the Driver implementors to decide how they wish to handle such binary data.

Character sets: Most databases which understand character sets have a default global charset and text stored in the database is, or should be, stored in that charset (if it's not then that's the fault of either the database or the application that inserted the data). When text is fetched it should be (automatically) converted to the charset of the client (presumably based on the locale). If a driver needs to set a flag to get that behaviour then it should do so. It should not require the application to do that.

Multiple SQL statements may not be combined in a single statement handle, e.g., a single \$sth (though some drivers do support this).

Non-sequential record reads are not supported in this version of the DBI. In other words, records can only be fetched in the order that the database returned them and once fetched they are forgotten.

Positioned updates and deletes are not directly supported by the DBI. See the description of the CursorName attribute for an alternative.

Individual Driver implementors are free to provide any private functions and/or handle attributes that they feel are useful. Private driver functions can be invoked using the DBI 'func' method. Private driver attributes are accessed just like standard attributes.

Many methods have an optional \%attr parameter which can be used to pass information to the driver implementing the method. Except where

specifically documented the `\%attr` parameter can only be used to pass driver specific hints. In general you can ignore `\%attr` parameters or pass it as `undef`.

## Naming Conventions and Name Space

The DBI package and all packages below it `'DBI::*'` are reserved for use by the DBI. Extensions and related modules use the `'DBIx::'` namespace. Package names beginning with `'DBD::'` are reserved for use by DBI database drivers. All environment variables used by the DBI or DBD's begin with `'DBI_'` or `'DBD_'`.

The letter case used for attribute names is significant and plays an important part in the portability of DBI scripts. The case of the attribute name is used to signify who defined the meaning of that name and its values.

Case of name	Has a meaning defined by
UPPER_CASE	Standards, e.g., X/Open, SQL92 etc (portable)
MixedCase	DBI API (portable), underscores are not used.
lower_case	Driver or Engine specific (non-portable)

It is of the utmost importance that Driver developers only use lowercase attribute names when defining private attributes. Private attribute names must be prefixed with the driver name or suitable abbreviation (e.g., `ora_` for Oracle, `ing_` for Ingres etc).

## Driver Specific Prefix Registry:

<code>ora_</code>	<code>DBD::Oracle</code>
<code>ing_</code>	<code>DBD::Ingres</code>
<code>odbc_</code>	<code>DBD::ODBC</code>
<code>syb_</code>	<code>DBD::Sybase</code>
<code>db2_</code>	<code>DBD::DB2</code>
<code>ix_</code>	<code>DBD::Informix</code>
<code>f_</code>	<code>DBD::File</code>
<code>csv_</code>	<code>DBD::CSV</code>
<code>file_</code>	<code>DBD::TextFile</code>
<code>xbase_</code>	<code>DBD::XBase</code>
<code>solid_</code>	<code>DBD::Solid</code>
<code>proxy_</code>	<code>DBD::Proxy</code>
<code>msql_</code>	<code>DBD::mSQL</code>
<code>mysql_</code>	<code>DBD::mysql</code>

## Placeholders and Bind Values

Some drivers support Placeholders and Bind Values. These drivers allow a database statement to contain placeholders, sometimes called parameter markers, that indicate values that will be supplied later, before the prepared statement is executed. For example, an application might use the following to insert a row of data into the SALES table:

```
insert into sales (product_code, qty, price) values (?, ?, ?)
```

or the following, to select the description for a product:

```
select description from products where product_code = ?
```

The `'?'` characters are the placeholders. The association of actual values with placeholders is known as binding and the values are referred to as bind values.

When using placeholders with the SQL `'LIKE'` qualifier you must remember that the placeholder substitutes for the whole string. So you should use `"... LIKE ? ..."` and include any wildcard characters in the value that

you bind to the placeholder.

## Null Values

Undefined values or 'undef' can be used to indicate null values. However, care must be taken in the particular case of trying to use null values to qualify a select statement. Consider:

```
select description from products where product_code = ?
```

Binding an undef (NULL) to the placeholder will *not* select rows which have a NULL product\_code! Refer to the SQL manual for your database engine or any SQL book for the reasons for this. To explicitly select NULLs you have to say "where product\_code is NULL" and to make that general you have to say:

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

and bind the same value to both placeholders.

## Performance

Without using placeholders, the insert statement above would have to contain the literal values to be inserted and it would have to be re-prepared and re-executed for each row. With placeholders, the insert statement only needs to be prepared once. The bind values for each row can be given to the execute method each time it's called. By avoiding the need to re-prepare the statement for each row the application typically runs many times faster! Here's an example:

```
my $sth = $dbh->prepare(q{
    INSERT INTO sales (product_code, qty, price) VALUES (?, ?, ?)
}) || die $dbh->errstr;
while (<>) {
    chop;
    my ($product_code, $qty, $price) = split /,/;
    $sth->execute($product_code, $qty, $price) || die $dbh->errstr;
}
$dbh->commit || die $dbh->errstr;
```

See the execute and bind\_param entries elsewhere in this document for more details.

The 'q{...}' style quoting used in this example avoids clashing with quotes that may be used in the SQL statement. Use the double-quote like 'qq{...}' operator if you want to interpolate variables into the string. See the section on "Quote and Quote-like Operators" in the perlop manpage for more details.

See the bind\_column entry elsewhere in this document for a related method used to associate perl variables with the *output* columns of a select statement.

## SQL - A Query Language

Most DBI drivers require applications to use a dialect of SQL (the Structured Query Language) to interact with the database engine. These links provide useful information and further links about SQL, the first is a good tutorial with much useful information and many links:

```
http://www.geocities.com/ResearchTriangle/Node/9672/sqltut.html
http://www.jcc.com/sql_stnd.html
http://www.contrib.andrew.cmu.edu/~shadow/sql.html
```

The DBI itself does not mandate or require any particular language to be used. It is language independant. In ODBC terms the DBI is in

'pass-thru' mode (individual drivers might not be). The only requirement is that queries and other statements must be expressed as a single string of letters passed as the first argument to the the prepare entry elsewhere in this document method.

For an interesting diversion on the \*real\* history of RDBMS and SQL, from the people who made it happen, see

<http://ftp.digital.com/pub/DEC/SRC/technical-notes/SRC-1997-018-html/sqlr95.html>

Follow the "And the rest" and "Intergalactic dataspeak" links for the SQL history.

## THE DBI CLASS

### DBI Class Methods

#### connect

```
$dbh = DBI->connect($data_source, $username, $password)
    || die $DBI::errstr;
$dbh = DBI->connect($data_source, $username, $password, \%attr)
    || die $DBI::errstr;
```

Establishes a database connection, or session, to the requested `data_source`. Returns a database handle object if the connect succeeds. Use `$dbh->disconnect` to terminate the connection.

If the connect fails (see below) it returns undef and sets `$DBI::err` and `$DBI::errstr` (it does \*not\* set `$!` etc). Generally you should test the return status of connect and print `$DBI::errstr` if it has failed.

Multiple simultaneous connections to multiple databases through multiple drivers can be made via the DBI. Simply make one connect call for each and keep a copy of each returned database handle.

The `$data_source` value should begin with 'dbi:driver\_name:'. The `driver_name` part specifies the driver that will be used to make the connection (letter case is significant).

As a convenience, if the `$data_source` parameter is undefined or empty the DBI will substitute the value of the environment variable `DBI_DSN`. If just the `driver_name` part is empty (i.e., `data_source` prefix is 'dbi::') the environment variable `DBI_DRIVER` is used. If neither variable is set then the connect dies.

Examples of `$data_source` values:

```
dbi:DriverName:database_name
dbi:DriverName:database_name@hostname:port
dbi:DriverName:database=database_name;host=hostname;port=port
```

There is \*no standard\* for the text following the driver name. Each driver is free to use whatever syntax it wants. The only requirement the DBI makes is that all the information is supplied in a single string. You must consult the documentation for the drivers you are using for a description of the syntax they require. (Where a driver author needs to define a syntax for the `data_source` it is recommended that they follow the ODBC style, the last example above.)

If the environment variable `DBI_AUTOPROXY` is defined (and the driver in `$data_source` is not 'Proxy') then the connect request will automatically be changed to:

```
dbi:Proxy:$ENV{DBI_AUTOPROXY};dsn=$data_source
```

and passed to the DBD::Proxy module. DBI\_AUTOPROXY would typically be "hostname=...;port=...". See the DBD::Proxy manpage for more details.

If \$username or \$password are \*undefined\* (rather than just empty) then the DBI will substitute the values of the DBI\_USER and DBI\_PASS environment variables respectively. The DBI will warn if the environment variables are not defined. The use of the environment for these values is not recommended for security reasons. The mechanism is primarily intended to simplify testing.

DBI->connect automatically installs the driver if it has not been installed yet. Driver installation \*always\* returns a valid driver handle or it \*dies\* with an error message which includes the string 'install\_driver' and the underlying problem. So, DBI->connect will die on a driver installation failure and will only return undef on a connect failure, for which \$DBI::errstr will hold the error.

The \$data\_source argument (with the 'dbi:...:' prefix removed) and the \$username and \$password arguments are then passed to the driver for processing. The DBI does not define \*any\* interpretation for the contents of these fields. The driver is free to interpret the data\_source, username and password fields in any way and supply whatever defaults are appropriate for the engine being accessed (Oracle, for example, uses the ORACLE\_SID and TWO\_TASK env vars if no data\_source is specified).

The AutoCommit and PrintError attributes for each connection default to \*on\* (see the AutoCommit and PrintError entries elsewhere in this document for more information). However, it is strongly recommended that AutoCommit is \*explicitly\* defined as required rather than rely on the default. Future versions of the DBI may issue a warning if AutoCommit is not explicitly defined.

The \%attr parameter can be used to alter the default settings of the PrintError, RaiseError, AutoCommit and other attributes. For example:

```
$dbh = DBI->connect($data_source, $user, $pass, {  
    PrintError => 0,  
    AutoCommit => 0  
});
```

You can also define connection attribute values within the \$data\_source parameter. For example

```
dbi:DriverName(PrintError=>0,Taint=>1):...
```

Individual attributes values specified in this way take precedence over any conflicting values specified via the \%attrib parameter to 'connect()'.

Portable applications should not assume that a single driver will be able to support multiple simultaneous sessions. Though most do.

Where possible each session (\$dbh) is independent from the transactions in other sessions. This is useful where you need to hold cursors open across transactions, e.g., use one session for your long lifespan cursors (typically read-only) and another for your short update transactions.

For compatibility with old DBI scripts the driver can be specified by passing its name as the fourth argument to connect (instead of \%attr):

```
$dbh = DBI->connect($data_source, $user, $pass, $driver);
```

In this 'old-style' form of connect the `$data_source` should not start with `'dbi:driver_name:'` and, even if it does, the embedded `driver_name` will be ignored. The `$dbh->{AutoCommit}` attribute is `*undefined*`. The `$dbh->{PrintError}` attribute is off. And the old `DBI_DBNAME` env var is checked if `DBI_DSN` is not defined. `*This 'old-style' connect will be withdrawn in a future version*`.

```
connect_cached *NEW*
    $dbh = DBI->connect_cached($data_source, $username, $password)
        || die $DBI::errstr;
    $dbh = DBI->connect_cached($data_source, $username, $password, \%attr)
        || die $DBI::errstr;
```

Like the `connect` entry elsewhere in this document except that the database handle returned will be stored in a hash associated with the given parameters. If another call is made to `connect_cached` with the `*same parameter values*` then the corresponding cached `$dbh` will be returned, without contacting the data source, so long as it is still `*ok*`. If the cached database handle has been disconnected or the `ping()` method fails then it will be replaced with a new connection.

Note that the behaviour of this method differs in several respects from the behaviour of the persistent connections implemented by `Apache::DBI`.

This caching can be useful in some applications but it can also cause problems and should be used with care. The exact behaviour of this method `*is liable to change*`. If you intend to use it in any production applications you should discuss your needs in the `dbi-users` mailing list.

The cache can be accessed (and cleared) via the `CachedKids` entry elsewhere in this document attribute.

```
available_drivers
    @ary = DBI->available_drivers;
    @ary = DBI->available_drivers($quiet);
```

Returns a list of all available drivers by searching for `DBD::*` modules through the directories in `@INC`. By default a warning will be given if some drivers are hidden by others of the same name in earlier directories. Passing a true value for `$quiet` will inhibit the warning.

```
data_sources
    @ary = DBI->data_sources($driver);
    @ary = DBI->data_sources($driver, \%attr);
```

Returns a list of all data sources (databases) available via the named driver. The driver will be loaded if not already. If `$driver` is empty or `undef` then the value of the `DBI_DRIVER` environment variable will be used.

Data sources will be returned in a form suitable for passing to the `connect` entry elsewhere in this document method, i.e., they will include the `"dbi:$driver:"` prefix.

Note that many drivers have no way of knowing what data sources might be available for it and thus, typically, return an empty or incomplete list.

```
trace
    DBI->trace($trace_level)
    DBI->trace($trace_level, $trace_filename)
```

DBI trace information can be enabled for all handles using this DBI class method. To enable trace information for a specific handle use the similar `$h->trace` method described elsewhere.

Trace Levels:

- 0 - trace disabled.
- 1 - trace DBI method calls returning with results.
- 2 - trace method entry with parameters and exit with results.
- 3 - as above, adding some high-level information from the driver also adds some internal information from the DBI.
- 4 - as above, adding more detailed information from the driver also includes DBI mutex information when using threaded perl.
- 5 and above - as above but with more and more obscure information.

Trace level 1 is best for a simple overview of what's happening. Trace level 2 is a good choice for general purpose tracing. Levels 3 and above (up to 9) are best reserved for use when investigating a specific problem and you need to see 'inside' the driver and DBI.

The trace output is detailed and typically *very* useful. Much of the trace output is formatted using the the neat entry elsewhere in this document function and thus strings in the trace output may be edited and truncated by it.

Initially trace output is written to `STDERR`. If `$trace_filename` is specified then the file is opened in append mode and *all* trace output (including that from other handles) is redirected to that file. Further calls to trace without a `$trace_filename` do not alter where the trace output is sent. If `$trace_filename` is undefined then trace output is sent to `STDERR` and the previous trace file closed.

See also the `$h->trace()` and `$h->trace_msg()` methodd and the `DEBUGGING` entry elsewhere in this document for information about the `DBI_TRACE` environment variable.

## DBI Utility Functions

neat

```
$str = DBI::neat($value, $maxlen);
```

Return a string containing a neat (and tidy) representation of the supplied value.

Strings will be quoted (but internal quotes will not be escaped). Values *known* to be numeric will be unquoted. Undefined (NULL) values will be shown as 'undef' (without quotes). Unprintable characters will be replaced by dot (.).

For result strings longer than `$maxlen` the result string will be truncated to `$maxlen-4` and `'...'` will be appended. If `$maxlen` is 0 or undef it defaults to `$DBI::neat_maxlen` which, in turn, defaults to 400.

This function is designed to format values for human consumption. It is used internally by the DBI for the trace entry elsewhere in this document output. It should typically *not* be used for formatting values for database use (see also the quote entry elsewhere in this document).

neat\_list

```
$str = DBI::neat_list(\@listref, $maxlen, $field_sep);
```

Calls `DBI::neat` on each element of the list and returns a string containing the results joined with `$field_sep`. `$field_sep` defaults

to `", "'.

```
looks_like_number
  @bool = DBI::looks_like_number(@array);
```

Returns true for each element that looks like a number. Returns false for each element that does not look like a number. Returns undef for each element that is undefined or empty.

## DBI Dynamic Attributes

These attributes are always associated with the last handle used.

Where an attribute is equivalent to a method call, then refer to the method call for all related documentation.

Warning: these attributes are provided as a convenience but they do have limitations. Specifically, because they are associated with the last handle used, they should only be used *\*immediately\** after calling the method which 'sets' them. They have a 'short lifespan'.

If in any doubt, use the corresponding method call.

```
$DBI::err
  Equivalent to $h->err.
```

```
$DBI::errstr
  Equivalent to $h->errstr.
```

```
$DBI::state
  Equivalent to $h->state.
```

```
$DBI::rows
  Equivalent to $h->rows. Please refer to the the rows entry elsewhere
  in this document method documentation.
```

## METHODS COMMON TO ALL HANDLES

```
err
  $rv = $h->err;
```

Returns the *\*native\** database engine error code from the last driver function called. The code is typically an integer but you should not assume that.

If you need to test for individual errors *\*and\** have your program be portable to different database engines, then you'll need to determine what the corresponding error codes are for all those engines and test for all of them.

```
errstr
  $str = $h->errstr;
```

Returns the native database engine error message from the last driver function called.

```
state
  $str = $h->state;
```

Returns an error code in the standard SQLSTATE five character format. Note that the specific success code '00000' is translated to '0' (false). If the driver does not support SQLSTATE, and most don't, then state will return 'S1000' (General Error) for all errors.

```
trace
  $h->trace($trace_level);
```

```
$h->trace($trace_level, $trace_filename);
```

DBI trace information can be enabled for a specific handle (and any future children of that handle) by setting the trace level using the trace method.

Trace level 1 is best for a simple overview of what's happening. Trace level 2 is a good choice for general purpose tracing. Levels 3 and above (up to 9) are best reserved for use when investigating a specific problem and you need to see 'inside' the driver and DBI. Use \$trace\_level 0 to disable the trace.

The trace output is detailed and typically *very* useful. Much of the trace output is formatted using the neat entry elsewhere in this document function and thus strings in the trace output may be edited and truncated by it.

Initially trace output is written to STDERR. If \$trace\_filename is specified then the file is opened in append mode and *all* trace output (including that from other handles) is redirected to that file. Further calls to trace without a \$trace\_filename do not alter where the trace output is sent. If \$trace\_filename is undefined then trace output is sent to STDERR and the previous trace file closed.

See also the DBI->trace() method and the DEBUGGING entry elsewhere in this document for information about the DBI\_TRACE environment variable.

trace\_msg

```
$h->trace_msg($message_text);  
$h->trace_msg($message_text, $min_level);
```

Writes \$message\_text to trace file *if* trace is enabled for \$h or for the DBI as a whole. Can also be called as DBI->trace\_msg(\$msg). See the trace entry elsewhere in this document.

If \$min\_level is defined then the message is output only if the trace level is equal to or greater than that level. \$min\_level defaults to 1.

func

```
$h->func(@func_arguments, $func_name);
```

The func method can be used to call private non-standard and non-portable methods implemented by the driver. Note that the function name is given as the *last* argument.

This method is not directly related to calling stored procedures. Calling stored procedures is currently not defined by the DBI. Some drivers, such as DBD::Oracle, support it in non-portable ways. See driver documentation for more details.

#### ATTRIBUTES COMMON TO ALL HANDLES

These attributes are common to all types of DBI handles.

Some attributes are inherited by *child* handles. That is, the value of an inherited attribute in a newly created statement handle is the same as the value in the parent database handle. Changes to attributes in the new statement handle do not affect the parent database handle and changes to the database handle do not affect *existing* statement handles, only future ones.

Attempting to set or get the value of an unknown attribute is fatal, except for private driver specific attributes (which all have names starting with a lowercase letter).

Example:

```
$h->{AttributeName} = ...; # set/write  
... = $h->{AttributeName}; # get/read
```

Warn (boolean, inherited)

Enables useful warnings for certain bad practices. Enabled by default. Some emulation layers, especially those for perl4 interfaces, disable warnings. Since warnings are generated using the perl warn() function they can be intercepted using the perl \$SIG{\_\_WARN\_\_} hook.

Active (boolean, read-only)

True if the handle object is 'active'. This is rarely used in applications. The exact meaning of active is somewhat vague at the moment. For a database handle it typically means that the handle is connected to a database (\$dbh->disconnect should set Active off). For a statement handle it *typically* means that the handle is a select that may have more data to fetch (\$dbh->finish or fetching all the data should set Active off).

Kids (integer, read-only)

For a driver handle, Kids is the number of currently existing database handles that were created from that driver handle. For a database handle, Kids is the number of currently existing statement handles that were created from that database handle.

ActiveKids (integer, read-only)

Like Kids (above), but only counting those that are Active (as above).

CachedKids (hash ref)

For a database handle, returns a reference to the cache (hash) of statement handles created by the the prepare\_cached entry elsewhere in this document method. For a driver handle, it would return a reference to the cache (hash) of statement handles created by the the connect\_cached entry elsewhere in this document method.

CompatMode (boolean, inherited)

Used by emulation layers (such as Oraperl) to enable compatible behaviour in the underlying driver (e.g., DBD::Oracle) for this handle. Not normally set by application code.

InactiveDestroy (boolean)

This attribute can be used to disable the database related effect of DESTROY'ing a handle (which would normally close a prepared statement or disconnect from the database etc). It is specifically designed for use in UNIX applications which 'fork' child processes. Either the parent or the child process, but not both, should set InactiveDestroy on all their handles. For a database handle, this attribute does not disable an *explicit* call to the disconnect method, only the implicit call from DESTROY.

PrintError (boolean, inherited)

This attribute can be used to force errors to generate warnings (using warn) in addition to returning error codes in the normal way. When set on, any method which results in an error occurring will cause the DBI to effectively do a warn("\$class \$method failed: \$DBI::errstr") where \$class is the driver class and \$method is the name of the method which failed. E.g.,

```
DBD::Oracle::db prepare failed: ... error text here ...
```

By default DBI->connect sets PrintError on (except for old-style connect usage, see the connect entry elsewhere in this document for more details).

If desired, the warnings can be caught and processed using a `$$SIG{__WARN__}` handler or modules like `CGI::Carp` and `CGI::ErrorWrap`.

`RaiseError` (boolean, inherited)

This attribute can be used to force errors to raise exceptions rather than simply return error codes in the normal way. It defaults to off. When set on, any method which results in an error occurring will cause the DBI to effectively do a `die("$class $method failed: $DBI::errstr")` where `$class` is the driver class and `$method` is the name of the method which failed. E.g.,

```
DBD::Oracle::db prepare failed: ... error text here ...
```

If `PrintError` is also on then the `PrintError` is done before the `RaiseError` unless no `__DIE__` handler has been defined, in which case `PrintError` is skipped since the die will print the message.

If you want to temporarily turn `RaiseError` off (inside a library function that is likely to fail for example), the recommended way is like this:

```
{
  local $h->{RaiseError} = 0 if $h->{RaiseError};
  ...
}
```

The original value will automatically and reliably be restored by perl regardless of how the block is exited. The `'... if $h->{RaiseError}'` is optional but makes the code slightly faster in the common case. The same logic applies to other attributes, including `RaiseError`.

Sadly this doesn't work for perl versions upto and including 5.004\_04. For backwards compatibility could just use `'eval { ... }'` instead.

`ChopBlanks` (boolean, inherited)

This attribute can be used to control the trimming of trailing space characters from *fixed width* character (CHAR) fields. No other field types are affected, even where field values have trailing spaces.

The default is false (it is possible that that may change). Applications that need specific behaviour should set the attribute as needed. Emulation interfaces should set the attribute to match the behaviour of the interface they are emulating.

Drivers are not required to support this attribute but any driver which does not must arrange to return undef as the attribute value.

`LongReadLen` (unsigned integer, inherited)

This attribute may be used to control the maximum length of 'long' ('blob', 'memo' etc.) fields which the driver will *read* from the database automatically when it fetches each row of data. The `LongReadLen` attribute only relates to fetching/reading long values it is *not* involved in inserting/updating them.

A value of 0 means don't automatically fetch any long data (fetch should return undef for long fields when `LongReadLen` is 0).

The default is typically 0 (zero) bytes but may vary between drivers. Applications fetching long fields should set this value to slightly larger than the longest long field value which will be fetched.

Some databases return some 'long' types encoded as pairs of hex digits. For these types LongReadLen relates to the underlying data and not the doubled-up length of the encoded string.

Changing the value of LongReadLen for a statement handle *after* it's been prepare()'d *will typically have no effect* so it's usual to set LongReadLen on the \$dbh before calling prepare.

Note that the value used here has a direct effect on the memory used by the application, so don't be too generous. It's also a good idea to use values which are just smaller than a power of 2, e.g.,  $2^{16-2}$  which is 65534 bytes.

See the LongTruncOk entry elsewhere in this document about truncation behaviour.

LongTruncOk (boolean, inherited)

This attribute may be used to control the effect of fetching a long field value which has been truncated (typically because it's longer than the value of the LongReadLen attribute).

By default LongTruncOk is false and fetching a truncated long value will cause the fetch to fail. (Applications should always take care to check for errors after a fetch loop in case an error, such as a divide by zero or long field truncation, caused the fetch to terminate prematurely.)

If a fetch fails due to a long field truncation when LongTruncOk is false, many drivers will allow you to continue fetching further rows.

See also the LongReadLen entry elsewhere in this document.

Taint (boolean, inherited)

If this attribute is set to some true value and Perl is running in taint mode (e.g., started with the '-T' option), then a) all data fetched from the database is tainted, and b) the arguments to most DBI method calls are checked for being tainted. *\*This may change.\**

The attribute defaults to off, even if perl is in taint mode. See the perlsec manpage for more about taint mode. If Perl is not running in taint mode, this attribute has no effect.

Currently only fetched data is tainted. It is likely that the results of other DBI method calls, and the value of fetched attributes, will also be tainted in future versions. That change may well break your applications unless you take great care now. If you use DBI Taint mode, please report your experience and any suggestions for changes.

private\_\*

The DBI provides a way to store extra information in a DBI handle as 'private' attributes. The DBI will allow you to store and retrieve any attribute which has a name starting with 'private\_'. It is *strongly* recommended that you use just *one* private attribute (e.g., use a hash ref) *and* give it a long and unambiguous name that includes the module or application that the attribute relates to (e.g., 'private\_YourFullModuleName\_thingy').

## DBI DATABASE HANDLE OBJECTS

### Database Handle Methods

selectrow\_array

```
@row_ary = $dbh->selectrow_array($statement);  
@row_ary = $dbh->selectrow_array($statement, \%attr);  
@row_ary = $dbh->selectrow_array($statement, \%attr, @bind_values);
```

This utility method combines the `prepare`, `execute`, and `fetchrow_array` entries elsewhere in this document into a single call. If called in a list context it returns the first row of data from the statement. If called in a scalar context it returns the first field of the first row. The `$statement` parameter can be a previously prepared statement handle in which case the `prepare` is skipped.

If any method fails, and the `RaiseError` entry elsewhere in this document is not set, `selectrow_array` will return an empty list (or `undef` in scalar context).

```
selectall_arrayref
    $ary_ref = $dbh->selectall_arrayref($statement);
    $ary_ref = $dbh->selectall_arrayref($statement, \%attr);
    $ary_ref = $dbh->selectall_arrayref($statement, \%attr, @bind_values);
```

This utility method combines the `prepare`, `execute`, and `fetchall_arrayref` entries elsewhere in this document into a single call. It returns a reference to an array containing references to arrays for each row of data fetched.

The `$statement` parameter can be a previously prepared statement handle in which case the `prepare` is skipped.

If any method except `fetch` fails, and the `RaiseError` entry elsewhere in this document is not set, `selectall_arrayref` will return `undef`. If `fetch` fails, and the `RaiseError` entry elsewhere in this document is not set, then it will return with whatever data it has fetched thus far.

```
selectcol_arrayref
    $ary_ref = $dbh->selectcol_arrayref($statement);
    $ary_ref = $dbh->selectcol_arrayref($statement, \%attr);
    $ary_ref = $dbh->selectcol_arrayref($statement, \%attr, @bind_values);
```

This utility method combines the `prepare` and `execute` entries elsewhere in this document and fetching one column from all the rows, into a single call. It returns a reference to an array containing the values of the first column from each row.

The `$statement` parameter can be a previously prepared statement handle in which case the `prepare` is skipped.

If any method except `fetch` fails, and the `RaiseError` entry elsewhere in this document is not set, `selectall_arrayref` will return `undef`. If `fetch` fails, and the `RaiseError` entry elsewhere in this document is not set, then it will return with whatever data it has fetched thus far.

```
prepare
    $sth = $dbh->prepare($statement)           || die $dbh->errstr;
    $sth = $dbh->prepare($statement, \%attr)   || die $dbh->errstr;
```

Prepares a *single* statement for later execution by the database engine and returns a reference to a statement handle object.

The returned statement handle can be used to get attributes of the statement and invoke the `execute` entry elsewhere in this document method. See the `Statement Handle Methods` entry elsewhere in this document.

Note that `prepare` should never execute a statement, even if it is not a `select` statement, it only *prepares* it for execution. (Having said that, some drivers, notably Oracle 7, will execute data

definition statements such as create/drop table when they are prepared. In practice this is rarely a problem.)

Drivers for engines which don't have the concept of preparing a statement will typically just store the statement in the returned handle and process it when `$sth->execute` is called. Such drivers are likely to be unable to give much useful information about the statement, such as `$sth->{NUM_OF_FIELDS}`, until after `$sth->execute` has been called. Portable applications should take this into account.

In general DBI drivers do *not* parse the contents of the statement (other than simply counting any the Placeholders entry elsewhere in this document). The statement is passed directly to the database engine (sometimes known as pass-thru mode). This has advantages and disadvantages. On the plus side, you can access all the functionality of the engine being used. On the downside, you're limited if using a simple engine and need to take extra care if attempting to write applications to be portable between engines.

Portable applications should not assume that a new statement can be prepared and/or executed while still fetching results from a previous statement.

Some command-line SQL tools use statement terminators, like a semicolon, to indicate the end of a statement. Such terminators should not be used with the DBI.

`prepare_cached`

```
$sth = $dbh->prepare_cached($statement)
$sth = $dbh->prepare_cached($statement, \%attr)
$sth = $dbh->prepare_cached($statement, \%attr, $allow_active)
```

Like the `prepare` entry elsewhere in this document except that the statement handle returned will be stored in a hash associated with the `$dbh`. If another call is made to `prepare_cached` with the *same* parameter values\* then the corresponding cached `$sth` will be returned without contacting the database server.

This caching can be useful in some applications but it can also cause problems and should be used with care. A warning will be generated if the cached `$sth` being returned is active (i.e., is a select that may still have data to be fetched) unless `$allow_active` is true.

The cache can be accessed (and cleared) via the `CachedKids` entry elsewhere in this document attribute.

`do`

```
$rc = $dbh->do($statement) || die $dbh->errstr;
$rc = $dbh->do($statement, \%attr) || die $dbh->errstr;
$rv = $dbh->do($statement, \%attr, @bind_values) || ...
```

Prepare and execute a single statement. Returns the number of rows affected (-1 if not known or not available) or undef on error.

This method is typically most useful for *non-select* statements which either cannot be prepared in advance (due to a limitation of the driver) or which do not need to be executed repeatedly. It should not be used for select statements because it does not return a statement handle so you can't fetch any data.

The default `do` method is logically similar to:

```
sub do {
    my($dbh, $statement, $attr, @bind_values) = @_;
```

```

    my $sth = $dbh->prepare($statement, $attr) or return undef;
    $sth->execute(@bind_values) or return undef;
    my $rows = $sth->rows;
    ($rows == 0) ? "0E0" : $rows; # always return true if no error
}

```

Example:

```

my $rows_deleted = $dbh->do(q{
    delete from table
    where status = ?
}, undef, 'DONE') || die $dbh->errstr;

```

Using placeholders and '@bind\_values' with the 'do' method can be useful because it avoids the need to correctly quote any variables in the \$statement.

The 'q{...}' style quoting used in this example avoids clashing with quotes that may be used in the SQL statement. Use the double-quote like 'qq{...}' operator if you want to interpolate variables into the string. See the section on "Quote and Quote-like Operators" in the perlop manpage for more details.

```

commit
    $rc = $dbh->commit    || die $dbh->errstr;

```

Commit (make permanent) the most recent series of database changes if the database supports transactions.

If the database supports transactions and AutoCommit is on then the commit should issue a "commit ineffective with AutoCommit" warning.

See also the Transactions entry elsewhere in this document.

```

rollback
    $rc = $dbh->rollback  || die $dbh->errstr;

```

Roll-back (undo) the most recent series of uncommitted database changes if the database supports transactions.

If the database supports transactions and AutoCommit is on then the rollback should issue a "rollback ineffective with AutoCommit" warning.

See also the Transactions entry elsewhere in this document.

```

disconnect
    $rc = $dbh->disconnect || warn $dbh->errstr;

```

Disconnects the database from the database handle. Typically only used before exiting the program. The handle is of little use after disconnecting.

The transaction behaviour of the disconnect method is, sadly, undefined. Some database systems (such as Oracle and Ingres) will automatically commit any outstanding changes, but others (such as Informix) will rollback any outstanding changes. Applications not using AutoCommit should explicitly call commit or rollback before calling disconnect.

The database is automatically disconnected (by the DESTROY method) if still connected when there are no longer any references to the handle. The DESTROY method for each driver *should* implicitly call rollback to undo any uncommitted changes. This is *vital* behaviour to ensure that incomplete transactions don't get committed simply because Perl calls DESTROY on every object before exiting. Also, do

not rely on the order of object destruction during 'global destruction', it is undefined.

Generally if you want your changes to be committed or rolled back when you disconnect then you should explicitly call the commit entry elsewhere in this document or the rollback entry elsewhere in this document before disconnecting.

If you disconnect from a database while you still have active statement handles you will get a warning. The statement handles should either be cleared (destroyed) before disconnecting or the finish method called on each one.

ping

```
$rc = $dbh->ping;
```

Attempts to determine, in a reasonably efficient way, if the database server is still running and the connection to it is still working. Individual drivers should implement this function in the most suitable manner for their database engine.

The default implementation currently always returns true without actually doing anything. Actually it returns "\0E0" which is true but zero. That way you can tell if the return value is genuine or just the default.

Very few applications would have any use for this method. See the specialist Apache::DBI module for one example usage.

table\_info \*NEW\*

Warning: This method is experimental and may change or disappear.

```
$sth = $dbh->table_info;
```

Returns an active statement handle that can be used to fetch information about tables and views that exist in the database.

The handle has at least the following fields in the order show below. Other fields, after these, may also be present.

TABLE\_CAT: Table catalogue identifier. NULL (undef) if not applicable to data source (usually the case). Empty if not applicable to the table.

TABLE\_SCHEM: The name of the schema containing the TABLE\_NAME value. NULL (undef) if not applicable to data source. Empty if not applicable to the table.

TABLE\_NAME: Name of the table (or view, synonym, etc).

TABLE\_TYPE: One of the following: "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM" or a data source specific type identifier.

REMARKS: A description of the table. May be NULL (undef).

Note that table\_info might not return records for all tables. Applications can use any valid table regardless of whether it's returned by table\_info. See also the tables entry elsewhere in this document.

For more detailed information about the fields and their meanings, you can refer to:

<http://msdn.microsoft.com/library/sdkdoc/dasdk/odbc/odbcsqltables.htm>

tables \*NEW\*

Warning: This method is experimental and may change or disappear.

```
@names = $dbh->tables;
```

Returns a list of table and view names. This list should include all tables which can be used in a select statement without further qualification. That typically means all the tables and views owned by the current user and all those accessible via public synonyms/aliases (excluding non-metadata system tables and views).

Note that table\_info might not return records for all tables. Applications can use any valid table regardless of whether it's returned by tables. See also the table\_info entry elsewhere in this document.

type\_info\_all \*NEW\*

Warning: This method is experimental and may change or disappear.

```
$type_info_all = $dbh->type_info_all;
```

Returns a reference to an array which holds information about each data type variant supported by the database and driver. The array and its contents should be treated as read-only.

The first item is a reference to a hash of Name => Index pairs. The following items are references to arrays, one per supported data type variant. The leading hash defines the names and order of the fields within the following list of arrays. For example:

```
$type_info_all = [  
  {  
    TYPE_NAME           => 0,  
    DATA_TYPE          => 1,  
    COLUMN_SIZE         => 2,      # was PRECISION originally  
    LITERAL_PREFIX      => 3,  
    LITERAL_SUFFIX      => 4,  
    CREATE_PARAMS       => 5,  
    NULLABLE            => 6,  
    CASE_SENSITIVE     => 7,  
    SEARCHABLE          => 8,  
    UNSIGNED_ATTRIBUTE => 9,  
    FIXED_PREC_SCALE   => 10,     # was MONEY originally  
    AUTO_UNIQUE_VALUE  => 11,     # was AUTO_INCREMENT originally  
    LOCAL_TYPE_NAME    => 12,  
    MINIMUM_SCALE      => 13,  
    MAXIMUM_SCALE      => 14,  
    NUM_PREC_RADIX     => 15,  
  },  
  [ 'VARCHAR', SQL_VARCHAR,  
    undef, "", "", undef, 0, 1, 1, 0, 0, 0, undef, 1, 255, undef  
  ],  
  [ 'INTEGER', SQL_INTEGER,  
    undef, "", "", undef, 0, 0, 1, 0, 0, 0, undef, 0, 0, 10  
  ],  
];
```

Note that more than one row may have the same value in the DATA\_TYPE field if there are different ways to spell the type name and/or there are variants of the type with different attributes (e.g., with and without AUTO\_UNIQUE\_VALUE set, with and without UNSIGNED\_ATTRIBUTE etc).

The rows are ordered by DATA\_TYPE first and then by how closely each type maps to the corresponding ODBC SQL data type, closest first.

The meaning of the fields is described in the documentation for the

the `type_info` entry elsewhere in this document method. The index values shown above (e.g., `NULLABLE => 6`) are for illustration only. Drivers may define the fields with a different order.

This method is not normally used directly. The `type_info` entry elsewhere in this document method provides a more useful interface to the data.

`type_info *NEW*`

Warning: This method is experimental and may change or disappear.

```
@type_info = $dbh->type_info($data_type);
```

Returns a list of hash references holding information about one or more variants of `$data_type` (or a type *reasonably compatible* with it).

If `$data_type` is `SQL_ALL_TYPES` then the list will contain hashes for all data type variants supported by the database and driver.

The list is ordered by `DATA_TYPE` first and then by how closely each type maps to the corresponding ODBC SQL data type, closest first.

The keys of the hash follow the same letter case conventions as the rest of the DBI (see the Naming Conventions and Name Space entry elsewhere in this document). The following items should exist:

`TYPE_NAME` (string)

Data type name for use in CREATE TABLE statements etc.

`DATA_TYPE` (integer)

SQL data type number.

`COLUMN_SIZE` (integer)

For numeric types this is either the total number of digits (if the `NUM_PREC_RADIX` value is 10) or the total number of bits allowed in the column (if `NUM_PREC_RADIX` is 2).

For string types this is the maximum size of the string in bytes.

For date and interval types this is the maximum number of characters needed to display the value.

`LITERAL_PREFIX` (string)

Characters used to prefix a literal. Typically `'` for characters, possibly `0x` for binary values passed as hex. `NULL` (undef) is returned for data types where this is not applicable.

`LITERAL_SUFFIX` (string)

Characters used to suffix a literal. Typically `'` for characters. `NULL` (undef) is returned for data types where this is not applicable.

`CREATE_PARAMS` (string)

Parameters for a data type definition. For example, `CREATE_PARAMS` for a `DECIMAL` would be `"precision,scale"`. For a `VARCHAR` it would be `"max length"`. `NULL` (undef) is returned for data types where this is not applicable.

`NULLABLE` (integer)

Indicates whether the data type accepts a `NULL` value: 0 = no, 1 = yes, 2 = unknown.

`CASE_SENSITIVE` (boolean)

Indicates whether the data type is case sensitive in collations

and comparisons.

SEARCHABLE (integer)

Indicates how the data type can be used in a WHERE clause:

- 0 - cannot be used in a WHERE clause
- 1 - only with a LIKE predicate
- 2 - all comparison operators except LIKE
- 3 - can be used in a WHERE clause with any comparison operator

UNSIGNED\_ATTRIBUTE (boolean)

Indicates whether the data type is unsigned. NULL (undef) is returned for data types where this is not applicable.

FIXED\_PREC\_SCALE (boolean)

Indicates whether the data type always has the same precision and scale (such as a money type). NULL (undef) is returned for data types where this is not applicable.

AUTO\_UNIQUE\_VALUE (boolean)

Indicates whether a column of this data type is automatically set to a unique value whenever a new row is inserted. NULL (undef) is returned for data types where this is not applicable.

LOCAL\_TYPE\_NAME (string)

Localised version of the TYPE\_NAME for use in dialogue with users. NULL (undef) is returned if a localised name is not available (in which case TYPE\_NAME should be used).

MINIMUM\_SCALE (integer)

The minimum scale of the data type. If a data type has a fixed scale then MAXIMUM\_SCALE holds the same value. NULL (undef) is returned for data types where this is not applicable.

MAXIMUM\_SCALE (integer)

The maximum scale of the data type. If a data type has a fixed scale then MINIMUM\_SCALE holds the same value. NULL (undef) is returned for data types where this is not applicable.

NUM\_PREC\_RADIX (integer)

The radix value of the data type. For approximate numeric types this contains the value 2 and COLUMN\_SIZE holds the number of bits. For exact numeric types this contains the value 10 and COLUMN\_SIZE holds the number of decimal digits. NULL (undef) is returned for data types where this is not applicable.

For more detailed information about these fields and their meanings, you can refer to:

<http://msdn.microsoft.com/library/sdkdoc/dasdk/odbc/odbcsqlgettypeinfo.htm>

The individual data types are described here:

[http://msdn.microsoft.com/library/sdkdoc/dasdk/odbc/odbcsql\\_data\\_types.htm](http://msdn.microsoft.com/library/sdkdoc/dasdk/odbc/odbcsql_data_types.htm)

quote

```
$sql = $dbh->quote($value);  
$sql = $dbh->quote($value, $data_type);
```

Quote a string literal for use as a literal value in an SQL statement by \*escaping\* any special characters (such as quotation marks) contained within the string \*and\* adding the required type of outer quotation marks.

```
$sql = sprintf "select foo from bar where baz = %s",  
             $dbh->quote("Don't\n");
```

For most database types quote would return `'Don't'` (including the outer quotation marks).

An undefined `$value` value will be returned as the string `NULL` (without quotation marks) to match how `NULLs` are represented in SQL.

If `$data_type` is supplied it is used to try to determine the required quoting behaviour by using the information returned by the `type_info` entry elsewhere in this document. As a special case, the standard numeric types are optimised to return `$value` without calling `type_info`.

Quote will probably *not* be able to deal with all possible input (such as binary data or data containing newlines) and is not related in any way with escaping or quoting shell meta-characters. There is no need to quote values being used with the section on "Placeholders and Bind Values".

## Database Handle Attributes

This section describes attributes specific to database handles.

Changes to these database handle attributes do not affect any other existing or future database handles.

Attempting to set or get the value of an unknown attribute is fatal, except for private driver specific attributes (which all have names starting with a lowercase letter).

Example:

```
$h->{AutoCommit} = ...;      # set/write
... = $h->{AutoCommit};     # get/read
```

### AutoCommit (boolean)

If true then database changes cannot be rolled-back (undone). If false then database changes automatically occur within a 'transaction' which must either be committed or rolled-back using the `commit` or `rollback` methods.

Drivers should always default to AutoCommit mode. (An unfortunate choice largely forced on the DBI by ODBC and JDBC conventions.)

Attempting to set AutoCommit to an unsupported value is a fatal error. This is an important feature of the DBI. Applications which need full transaction behaviour can set `$dbh->{AutoCommit} = 0` (or via the `connect` entry elsewhere in this document) without having to check the value was assigned okay.

For the purposes of this description we can divide databases into three categories:

- Database which don't support transactions at all.
- Database in which a transaction is always active.
- Database in which a transaction must be explicitly started ('BEGIN WORK').

\* Database which don't support transactions at all

For these databases attempting to turn AutoCommit off is a fatal error. Commit and rollback both issue warnings about being ineffective while AutoCommit is in effect.

\* Database in which a transaction is always active

These are typically mainstream commercial relational databases with

'ANSI standard' transaction behaviour.

If AutoCommit is off then changes to the database won't have any lasting effect unless the commit entry elsewhere in this document is called (but see also the disconnect entry elsewhere in this document). If the rollback entry elsewhere in this document is called then any changes since the last commit are undone.

If AutoCommit is on then the effect is the same as if the DBI were to have called commit automatically after every successful database operation. In other words, calling commit or rollback explicitly while AutoCommit is on would be ineffective because the changes would have already been committed.

Changing AutoCommit from off to on should issue a the commit entry elsewhere in this document in most drivers.

Changing AutoCommit from on to off should have no immediate effect.

For databases which don't support a specific auto-commit mode, the driver has to commit each statement automatically using an explicit COMMIT after it completes successfully (and roll it back using an explicit ROLLBACK if it fails). The error information reported to the application will correspond to the statement which was executed, unless it succeeded and the commit or rollback failed.

\* Database in which a transaction must be explicitly started

For these database the intention is to have them act like databases in which a transaction is always active (as described above).

To do this the DBI driver will automatically begin a transaction when AutoCommit is turned off (from the default on state) and will automatically begin another transaction after a the commit entry elsewhere in this document or the rollback entry elsewhere in this document.

In this way, the application does not have to treat these databases as a special case.

See the disconnect entry elsewhere in this document for other important notes about transactions.

Driver (handle)

Holds the handle of the parent Driver. The only recommended use for this is to find the name of the driver using

```
$dbh->{Driver}->{Name}
```

Name (string)

Holds the 'name' of the database. Usually (and recommended to be) the same as the "dbi:DriverName:..." string used to connect to the database but with the leading "dbi:DriverName:" removed.

RowCacheSize (integer) \*NEW\*

A hint to the driver indicating the size of local row cache the application would like the driver to use for future select statements. If a row cache is not implemented then setting RowCacheSize is ignored and getting the value returns undef.

Some RowCacheSize values have special meaning:

- 0 - Automatically determine a reasonable cache size for each select
- 1 - Disable the local row cache
- >1 - Cache this many rows
- <0 - Cache as many rows fit into this much memory for each select.

Note that large cache sizes may require very large amount of memory (cached rows \* maximum size of row) and that a large cache will cause a longer delay for the first fetch and when the cache needs refilling.

See also the RowsInCache entry elsewhere in this document statement handle attribute.

## DBI STATEMENT HANDLE OBJECTS

### Statement Handle Methods

#### bind\_param

```
$rc = $sth->bind_param($p_num, $bind_value) || die $sth->errstr;
$rv = $sth->bind_param($p_num, $bind_value, \%attr) || ...
$rv = $sth->bind_param($p_num, $bind_value, $bind_type) || ...
```

The `bind_param` method can be used to *bind* (assign/associate) a value with a *placeholder* embedded in the prepared statement. Placeholders are indicated with question mark character ('?'). For example:

```
$dbh->{RaiseError} = 1;          # save having to check each method call
$sth = $dbh->prepare("select name, age from people where name like ?");
$sth->bind_param(1, "John%");    # placeholders are numbered from 1
$sth->execute;
DBI::dump_results($sth);
```

Note that the '?' is not enclosed in quotation marks even when the placeholder represents a string. Some drivers also allow ':1', ':2' etc and ':name' style placeholders in addition to '?' but their use is not portable. Undefined bind values or 'undef' are be used to indicate null values.

Some drivers do not support placeholders.

With most drivers placeholders can't be used for any element of a statement that would prevent the database server validating the statement and creating a query execution plan for it. For example:

```
"select name, age from ?"          # wrong (will probably fail)
"select name, ? from people"      # wrong (but may not 'fail')
```

Also, placeholders can only represent single scalar values, so this statement, for example, won't work as expected for more than one value:

```
"select name, age from people where name in (?)"    # wrong
```

#### Data Types for Placeholders

The '\%attr' parameter can be used to hint at the data type the placeholder should have. Typically the driver is only interested in knowing if the placeholder should be bound as a number or a string.

```
$sth->bind_param(1, $value, { TYPE => SQL_INTEGER });
```

As a short-cut for this common case, the data type can be passed directly in place of the attr hash reference. This example is equivalent to the one above:

```
$sth->bind_param(1, $value, SQL_INTEGER);
```

The TYPE value indicates the *standard* (non-driver-specific) type for this parameter. To specify the driver-specific type the driver *may* support a driver-specific attribute, e.g., { ora\_type => 97 }.

The data type for a placeholder cannot be changed after the first `bind_param` call (but it can be left unspecified, in which case it defaults to the previous value).

Perl only has string and number scalar data types. All database types that aren't numbers are bound as strings and must be in a format the database will understand.

As an alternative to specifying the data type in the `bind_param` call, you can let the driver pass the value as the default type (VARCHAR) then use an SQL function to convert the type within the statement. E.g.,

```
insert into price(code, price) values (?, convert(money,?))
```

The `convert` function used here is just an example. The actual function and syntax will vary between different databases (and so is obviously non-portable).

See also the section on "Placeholders and Bind Values" for more information.

```
bind_param_inout
    $rc = $sth->bind_param_inout($p_num, \$bind_value, $max_len) || die $sth->errstr
r;
    $rv = $sth->bind_param_inout($p_num, \$bind_value, $max_len, \%attr) || ...
    $rv = $sth->bind_param_inout($p_num, \$bind_value, $max_len, $bind_type) || ...
```

This method acts like the `bind_param` entry elsewhere in this document but also enables values to be *output from* (updated by) the statement. The statement is typically a call to a stored procedure. The `$bind_value` must be passed as a *reference* to the actual value to be used.

Note that unlike the `bind_param` entry elsewhere in this document, the `$bind_value` variable is *not* read when `bind_param_inout` is called. Instead, the value in the variable is read at the time the `execute` entry elsewhere in this document is called.

The additional `$max_len` parameter specifies the minimum amount of memory to allocate to `$bind_value` for the new value. If the value is too big to fit then the `execute` should fail. If unsure what value to use, pick a generous length larger than the longest value that would ever be returned. The only cost of using a very large value is memory.

It is expected that few drivers will support this method. The only driver currently known to do so is `DBD::Oracle` (`DBD::ODBC` may support it in a future release). Therefore it should *not* be used for database independent applications.

Undefined values or `'undef'` are be used to indicate null values. See also the section on "Placeholders and Bind Values" for more information.

```
execute
    $rv = $sth->execute || die $sth->errstr;
    $rv = $sth->execute(@bind_values) || die $sth->errstr;
```

Perform whatever processing is necessary to execute the prepared statement. An `undef` is returned if an error occurs, a successful `execute` always returns true regardless of the number of rows affected (even if it's zero, see below). It is always important to check the return status of `execute` (and most other DBI methods) for errors.

For a *\*non-select\** statement, `execute` returns the number of rows affected (if known). If no rows were affected then `execute` returns "0E0" which Perl will treat as 0 but will regard as true. Note that it is *\*not\** an error for no rows to be affected by a statement. If the number of rows affected is not known then `execute` returns -1.

For *\*select\** statements `execute` simply 'starts' the query within the Engine. Use one of the fetch methods to retrieve the data after calling `execute`. The `execute` method does *\*not\** return the number of rows that will be returned by the query (because most Engines can't tell in advance), it simply returns a true value.

If any arguments are given then `execute` will effectively call the `bind_param` entry elsewhere in this document for each value before executing the statement. Values bound in this way are usually treated as `SQL_VARCHAR` types unless the driver can determine the correct type (which is rare) or `bind_param` (or `bind_param_inout`) has already been used to specify the type.

#### `fetchrow_arrayref`

```
$ary_ref = $sth->fetchrow_arrayref;  
$ary_ref = $sth->fetch;      # alias
```

Fetches the next row of data and returns a reference to an array holding the field values. Null field values are returned as `undef`. This is the fastest way to fetch data, particularly if used with `$sth->bind_columns`.

If there are no more rows *\*or\** an error occurs then `fetchrow_arrayref` returns an `undef`. You should check `$sth->err` afterwards (or use the `RaiseError` attribute) to discover if the `undef` returned was due to an error.

Note that *\*currently\** the *\*same\** array ref will be returned for each fetch so don't store the ref and then use it after a later fetch.

#### `fetchrow_array`

```
@ary = $sth->fetchrow_array;
```

An alternative to `'fetchrow_arrayref'`. Fetches the next row of data and returns it as an array holding the field values. Null values are returned as `undef`.

If there are no more rows *\*or\** an error occurs then `fetchrow_array` returns an empty list. You should check `$sth->err` afterwards (or use the `RaiseError` attribute) to discover if the empty list returned was due to an error.

#### `fetchrow_hashref`

```
$hash_ref = $sth->fetchrow_hashref;  
$hash_ref = $sth->fetchrow_hashref($name);
```

An alternative to `'fetchrow_arrayref'`. Fetches the next row of data and returns it as a reference to a hash containing field name and field value pairs. Null values are returned as `undef`.

If there are no more rows *\*or\** an error occurs then `fetchrow_hashref` returns an `undef`. You should check `$sth->err` afterwards (or use the `RaiseError` attribute) to discover if the `undef` returned was due to an error.

The optional `$name` parameter specifies the name of the statement handle attribute to use to get the field names. It defaults to `'the NAME entry elsewhere in this document'`.

The keys of the hash are the same names returned by `$sth->{$name}`.

If more than one field has the same name there will only be one entry in the returned hash for those fields.

Note that using `fetchrow_hashref` is currently *\*not portable\** between databases because different databases return fields names with different letter cases (some all uppercase, some all lower, and some return the letter case used to create the table). This will be addressed in a future version of the DBI.

Because of the extra work `fetchrow_hashref` and perl have to perform it is not as efficient as `fetchrow_arrayref` or `fetchrow_array` and is not recommended where performance is very important. Currently a new hash reference is returned for each row. This is likely to change in the future so don't rely on it.

`fetchall_arrayref`

```
$tbl_ary_ref = $sth->fetchall_arrayref;  
$tbl_ary_ref = $sth->fetchall_arrayref( $slice_array_ref );  
$tbl_ary_ref = $sth->fetchall_arrayref( $slice_hash_ref );
```

The `'fetchall_arrayref'` method can be used to fetch all the data to be returned from a prepared and executed statement handle. It returns a reference to an array which contains one reference per row.

If there are no rows to return, `fetchall_arrayref` returns a reference to an empty array. If an error occurs `fetchall_arrayref` returns the data fetched thus far, which may be none. You should check `$sth->err` afterwards (or use the `RaiseError` attribute) to discover if the data is complete or was truncated due to an error.

When passed an array reference, `fetchall_arrayref` uses the `fetchrow_arrayref` entry elsewhere in this document to fetch each row as an array ref. If the parameter array is not empty then it is used as a slice to select individual columns by index number.

With no parameters, `fetchall_arrayref` acts as if passed an empty array ref.

When passed a hash reference, `fetchall_arrayref` uses the `fetchrow_hashref` entry elsewhere in this document to fetch each row as a hash ref. If the parameter hash is not empty then it is used as a slice to select individual columns by name. The names should be lower case regardless of the letter case in `$sth->{NAME}`. The values of the hash should be set to 1.

For example, to fetch just the first column of every row you can use:

```
$tbl_ary_ref = $sth->fetchall_arrayref([0]);
```

To fetch the second to last and last column of every row you can use:

```
$tbl_ary_ref = $sth->fetchall_arrayref([-2,-1]);
```

To fetch only the fields called `foo` and `bar` of every row you can use:

```
$tbl_ary_ref = $sth->fetchall_arrayref({ foo=>1, bar=>1 });
```

The first two examples return a ref to an array of array refs. The last returns a ref to an array of hash refs.

`finish`

```
$rc = $sth->finish;
```

Indicates that no more data will be fetched from this statement handle before it is either executed again or destroyed. \*It is rarely needed\* but can sometimes be helpful in very specific situations in order to allow the server to free up resources currently being held (such as sort buffers).

When all the data has been fetched from a select statement the driver should automatically call finish for you. So you should not normally need to call it explicitly.

Consider a query like

```
SELECT foo FROM table WHERE bar=? ORDER BY foo
```

where you want to select just the first (smallest) foo value from a very large table. When executed the database server will have to use temporary buffer space to store the sorted rows. If, after executing the handle and selecting one row, the handle won't be re-executed for some time and won't be destroyed, the finish method can be used to tell the server that the buffer space can be freed.

Calling finish resets the the Active entry elsewhere in this document attribute for the statement. It may also make some statement handle attributes, like NAME and TYPE etc., unavailable if they have not already been accessed (and thus cached).

The finish method does not affect the transaction status of the session. It has nothing to do with transactions. It's mostly an internal 'housekeeping' method that is rarely needed. There's no need to call finish if you're about to destroy or re-execute the statement handle. See also the disconnect entry elsewhere in this document and the the Active entry elsewhere in this document attribute.

rows

```
$rv = $sth->rows;
```

Returns the number of rows affected by the last row affecting command, or -1 if not known or not available.

Generally you can only rely on a row count after a \*non\*-select 'execute' (for some specific operations like update and delete) or after fetching \*all\* the rows of a select statement.

For select statements it is generally not possible to know how many rows will be returned except by fetching them all. Some drivers will return the number of rows the application has fetched \*so far\* but others may return -1 until all rows have been fetched. So, use of the rows method, or \$DBI::rows, with select statements is \*not\* recommended.

bind\_col

```
$rc = $sth->bind_col($column_number, \$var_to_bind);
```

Binds an output column (field) of a select statement to a perl variable. You do not need to do this but it can be useful for some applications.

Whenever a row is fetched from the database the corresponding perl variable is automatically updated. There is no need to fetch and assign the values manually. This makes using bound variables very efficient. See bind\_columns below for an example. Note that column numbers count up from 1.

The binding is performed at a very low level using perl aliasing so

there is no extra copying taking place. So long as the driver uses the correct internal DBI call to get the array the fetch function returns, it will automatically support column binding.

For maximum portability between drivers, `bind_col` should be called after `execute`.

The `bind_param` entry elsewhere in this document method performs a similar function for input variables.

#### `bind_columns`

```
$rc = $sth->bind_columns(@list_of_refs_to_vars_to_bind);
```

Calls the `bind_col` entry elsewhere in this document for each column of the select statement. You do not need to bind columns but it can be useful for some applications. The `bind_columns` method will die if the number of references does not match the number of fields.

For maximum portability between drivers, `bind_columns` should be called after `execute`.

For example:

```
$dbh->{RaiseError} = 1; # do this, or check every call for errors
$sth = $dbh->prepare(q{ select region, sales from sales_by_region });
$sth->execute;
my ($region, $sales);

# Bind perl variables to columns:
$rv = $sth->bind_columns(\$region, \$sales);

# you can also use perl's \(...) syntax (see perlref docs):
#   $sth->bind_columns(\$region, $sales);

# Column binding is the most efficient way to fetch data
while ($sth->fetch) {
    print "$region: $sales\n";
}
```

For compatibility with old scripts, if the first parameter is undef or a hash reference it will be ignored.

#### `dump_results`

```
$rows = $sth->dump_results($maxlen, $lsep, $fsep, $fh);
```

Fetches all the rows from `$sth`, calls `DBI::neat_list` for each row and prints the results to `$fh` (defaults to `'STDOUT'`) separated by `$lsep` (default `''\n''`). `$fsep` defaults to `','` and `$maxlen` defaults to 35.

This method is designed as a handy utility for prototyping and testing queries. Since it uses the `neat_list` entry elsewhere in this document which uses the `neat` entry elsewhere in this document which formats and edits the string for reading by humans, it's not recommended for data transfer applications.

#### Statement Handle Attributes

This section describes attributes specific to statement handles. Most of these attributes are read-only.

Changes to these statement handle attributes do not affect any other existing or future statement handles.

Attempting to set or get the value of an unknown attribute is fatal, except for private driver specific attributes (which all have names

starting with a lowercase letter).

Example:

```
... = $h->{NUM_OF_FIELDS};    # get/read
```

Note that some drivers cannot provide valid values for some or all of these attributes until after `$sth->execute` has been called.

See also the `finish` entry elsewhere in this document for the effect it may have on some attributes.

**NUM\_OF\_FIELDS** (integer, read-only)

Number of fields (columns) the prepared statement will return.  
Non-select statements will have `NUM_OF_FIELDS == 0`.

**NUM\_OF\_PARAMS** (integer, read-only)

The number of parameters (placeholders) in the prepared statement.  
See **SUBSTITUTION VARIABLES** below for more details.

**NAME** (array-ref, read-only)

Returns a *reference* to an array of field names for each column. The names may contain spaces but should not be truncated or have any trailing space. Note that the names have the letter case (upper, lower or mixed) as returned by the driver being used. Portable applications should use the `NAME_lc` entry elsewhere in this document or the `NAME_uc` entry elsewhere in this document.

```
print "First column name: $sth->{NAME}->[0]\n";
```

**NAME\_lc** (array-ref, read-only)

Like the `NAME` entry elsewhere in this document but always returns lowercase names.

**NAME\_uc** (array-ref, read-only)

Like the `NAME` entry elsewhere in this document but always returns uppercase names.

**TYPE** (array-ref, read-only) *\*NEW\**

Returns a *reference* to an array of integer values for each column. The value indicates the data type of the corresponding column.

The values used correspond to the international standards (ANSI X3.135 and ISO/IEC 9075) which, in general terms, means ODBC. Driver specific types which don't exactly match standard types should generally return the same values as an ODBC driver supplied by the makers of the database. That might include private type numbers in ranges the vendor has officially registered. See:

```
ftp://jerry.ece.umassd.edu/isowg3/dbl/SQL_Registry
```

Where there's no vendor supplied ODBC driver to be compatible with, the DBI driver can use type numbers in the range now *\*officially\** reserved for use by the DBI: -9999 to -9000.

All possible values for `TYPE` should have at least one entry in the output of the `'type_info_all()'` method (see the `type_info_all` entry elsewhere in this document).

**PRECISION** (array-ref, read-only) *\*NEW\**

Returns a *reference* to an array of integer values for each column. For nonnumeric columns the value generally refers to either the maximum length or the defined length of the column. For numeric columns the value refers to the maximum number of significant digits used by the data type (without considering a sign character or decimal point). Note that for floating point types (`REAL`, `FLOAT`,

DOUBLE) the 'display size' can be up to 7 characters greater than the precision (for the sign + decimal point + the letter E + a sign + 2 or 3 digits).

SCALE (array-ref, read-only) \*NEW\*

Returns a \*reference\* to an array of integer values for each column. NULL (undef) values indicate columns where scale is not applicable.

NULLABLE (array-ref, read-only)

Returns a \*reference\* to an array indicating the possibility of each column returning a null: 0 = no, 1 = yes, 2 = unknown.

```
print "First column may return NULL\n" if $sth->{NULLABLE}->[0];
```

CursorName (string, read-only)

Returns the name of the cursor associated with the statement handle if available. If not available or the database driver does not support the '"where current of ..."' SQL syntax then it returns undef.

Statement (string, read-only) \*NEW\*

Returns the statement string passed to the the prepare entry elsewhere in this document method.

RowsInCache (integer, read-only) \*NEW\*

If the driver supports a local row cache for select statements then this attribute holds the number of un-fetched rows in the cache. If the driver doesn't, then it returns undef. Note that some drivers pre-fetch rows on execute, others wait till the first fetch.

See also the the RowCacheSize entry elsewhere in this document database handle attribute.

## FURTHER INFORMATION

### Transactions

Transactions are a fundamental part of any robust database system. They protect against errors and database corruption by ensuring that sets of related changes to the database take place in atomic (indivisible, all-or-nothing) units.

This section applies to databases which support transactions and where AutoCommit is \*off\*. See the AutoCommit entry elsewhere in this document for details of using AutoCommit with various types of database.

The recommended way to implement robust transactions in Perl applications is to make use of 'eval { ... }' (which is very fast, unlike 'eval "...").

```
eval {
    foo(...)    # do lots of work here
    bar(...)    # including inserts
    baz(...)    # and updates
};
if ($?) {
    # $? contains $DBI::errstr if DBI RaiseError caused die
    $dbh->rollback;
    # add other application on-error-clean-up code here
}
else {
    $dbh->commit;
}
```

The code in foo(), or any other code executed from within the curly braces, can be implemented in this way:

```
$h->method(@args) || die $h->errstr
```

or the `$h->{RaiseError}` attribute can be set on. With `RaiseError` set the DBI will automatically die if any DBI method call on that handle (or a child handle) fails, so you don't have to test the return value of each method call. See the `RaiseError` entry elsewhere in this document for more details.

A major advantage of the `eval` approach is that the transaction will be properly rolled back if *any* code in the inner application dies for *any* reason. The major advantage of using the `$h->{RaiseError}` attribute is that all DBI calls will be checked automatically. Both techniques are strongly recommended.

#### Handling BLOB / LONG / Memo Fields

Many databases support 'blob' (binary large objects), 'long' or similar datatypes for holding very long strings or large amounts of binary data in a single field. Some databases support variable length long values over 2,000,000,000 bytes in length.

Since values of that size can't usually be held in memory and because databases can't usually know in advance the length of the longest long that will be returned from a select statement (unlike other data types) some special handling is required.

In this situation the value of the `$h->{LongReadLen}` attribute is used to determine how much buffer space to allocate when *fetching* such fields. The `$h->{LongTruncOk}` attribute is used to determine how to behave if a fetched value can't fit into the buffer.

When trying to insert long or binary values, placeholders should be used since there are often limits on the maximum size of an (insert) statement and the the quote entry elsewhere in this document method generally can't cope with binary data. See the `Placeholders and Bind Values` entry elsewhere in this document.

#### Simple Examples

Here's a complete example program to select and fetch some data:

```
my $dbh = DBI->connect("dbi:DriverName:db_name", $user, $password)
    || die "Can't connect to $data_source: $DBI::errstr";

my $sth = $dbh->prepare( q{
    SELECT name, phone
    FROM mytelbook
}) || die "Can't prepare statement: $DBI::errstr";

my $rc = $sth->execute
    || die "Can't execute statement: $DBI::errstr";

print "Query will return $sth->{NUM_OF_FIELDS} fields.\n\n";
print "Field names: @{ $sth->{NAME} }\n";

while (($name, $phone) = $sth->fetchrow_array) {
    print "$name: $phone\n";
}
# check for problems which may have terminated the fetch early
die $sth->errstr if $sth->err;

$dbh->disconnect;
```

Here's a complete example program to insert some data from a file: (this example uses `RaiseError` to avoid needing to check each call)

```

my $dbh = DBI->connect("dbi:DriverName:db_name", $user, $password, {
    RaiseError => 1, AutoCommit => 0
});

my $sth = $dbh->prepare( q{
    INSERT INTO table (name, phone) VALUES (?, ?)
});

open FH, "<phone.csv" or die "Unable to open phone.csv: $!";
while (<FH>) {
    chop;
    my ($name, $phone) = split /,;/;
    $sth->execute($name, $phone);
}
close FH;

$dbh->commit;
$dbh->disconnect;

```

Converting fetched NULLs (undefined values) to empty strings:

```

while($row = $sth->fetchrow_arrayref) {
    # this is a fast and simple way to deal with nulls:
    foreach (@$row) { $_ = '' unless defined }
    print "@$row\n";
}

```

The 'q{...}' style quoting used in these example avoids clashing with quotes that may be used in the SQL statement. Use the double-quote like 'qq{...}' operator if you want to interpolate variables into the string. See the section on "Quote and Quote-like Operators" in the perlfaq manpage for more details.

### Threads and Thread Safety

Perl versions 5.004\_50 and later support threads on many platforms. The DBI should build on these platforms but currently has made \*no\* attempt to be thread safe.

### Signal Handling and Canceling Operations

The first thing to say is that signal handling in Perl is currently \*not\* safe. There is \*always\* a small risk of Perl crashing and/or core dumping when, or after, handling a signal. (The risk was reduced with 5.004\_04 but is still present.)

The two most common uses of signals in relation to the DBI are for canceling operations when the user types Ctrl-C (interrupt), and for implementing a timeout using alarm() and \$SIG{ALRM}.

To assist in implementing these operations the DBI provides a 'cancel' method for statement handles. The cancel method should abort the current operation and is designed to be called from a signal handler.

However, it must be stressed that: a) few drivers implement this at the moment (the DBI provides a default method that just returns undef), b) even if implemented there is still a possibility that the statement handle, and possibly the parent database handle, will not be usable afterwards.

If cancel returns true then it is implemented and has successfully invoked the database engine's own cancel function. If it returns false then cancel failed (undef if not implemented).

### DEBUGGING

In addition to the the trace entry elsewhere in this document method you

can enable the same trace information by setting the DBI\_TRACE environment variable before starting perl.

On unix-like systems using a bourne-like shell you can do this easily for a single command:

```
DBI_TRACE=2 perl your_test_script.pl
```

If DBI\_TRACE is set to a non-numeric value then it is assumed to be a file name and the trace level will be set to 2 with all trace output will be appended to that file. If the name begins with a number followed by an equals ('=') then they are stripped off from the name and the number is used to set the trace level. For example:

```
DBI_TRACE=3=dbitrace.log perl your_test_script.pl
```

See also the the trace entry elsewhere in this document method.

## WARNING AND ERROR MESSAGES

(This section needs more words about causes and remedies.)

### Fatal Errors

Can't call method "prepare" without a package or object reference  
The \$dbh handle you're using to call prepare is probably undefined because the preceding connect failed. You should always check the return status of DBI methods, or use the the RaiseError entry elsewhere in this document attribute.

Can't call method "execute" without a package or object reference  
The \$sth handle you're using to call execute is probably undefined because the preceding prepare failed. You should always check the return status of DBI methods, or use the the RaiseError entry elsewhere in this document attribute.

DBI/DBD internal version mismatch  
DBD driver has not implemented the AutoCommit attribute  
Can't [sg]let %s->{%s}: unrecognised attribute

### Warnings

Database handle destroyed without explicit disconnect  
DBI Handle cleared whilst still holding %d cached kids!  
DBI Handle cleared whilst still active!  
DBI Handle has uncleared implementors data  
DBI Handle has %d uncleared child handles

### SEE ALSO

Database Documentation

SQL Language Reference Manual.

### Books and Journals

Programming the Perl DBI, by Alligator Descartes and Tim Bunce.  
Due to be published by O'Reilly September/October 1999.

Programming Perl 2nd Ed. by Larry Wall, Tom Christiansen & Randal Schwartz.  
Learning Perl by Randal Schwartz.

Dr Dobb's Journal, November 1996.  
The Perl Journal, April 1997.

### Manual Pages

the perl(1) manpage, the perlmod(1) manpage, the perlbook(1) manpage

### Mailing List

The dbi-users mailing list is the primary means of communication among users of the DBI and its related modules. Subscribe and unsubscribe via:

<http://www.isc.org/dbi-lists.html>

Mailing list archives are held at:

<http://www.rosat.mpe-garching.mpg.de/mailling-lists/PerlDB-Interest/>  
<http://www.xray.mpe.mpg.de/mailling-lists/#dbi>  
<http://outside.organic.com/mail-archives/dbi-users/>  
<http://www.coe.missouri.edu/~faq/lists/dbi.html>

#### Assorted Related WWW Links

The DBI 'Home Page' (not maintained by me):

<http://www.symbolstone.org/technology/perl/DBI>

Other DBI related links:

<http://eskimo.tamu.edu/~jbaker/dbi-examples.html>  
<http://tegan.deltanet.com/~phlip/DBUIDoc.html>  
[http://dc.pm.org/perl\\_db.html](http://dc.pm.org/perl_db.html)  
<http://wdvl.com/Authoring/DB/Intro/toc.html>  
<http://www.hotwired.com/webmonkey/backend/tutorials/tutorial11.html>

Other database related links:

[http://www.jcc.com/sql\\_std.html](http://www.jcc.com/sql_std.html)  
<http://cuiwww.unige.ch/OSG/info/FreeDB/FreeDB.home.html>

#### Commercial and Data Warehouse Links

<http://www.datamining.org>  
<http://www.olapcouncil.org>  
<http://www.idwa.org>  
<http://www.knowledgecenters.org/dwcenter.asp>  
<http://pwp.starnetinc.com/larryg/>  
<http://www.data-warehouse.com>

#### Recommended Perl Programming Links

<http://language.perl.com/style/>

#### FAQ

Please also read the DBI FAQ which is installed as a DBI::FAQ module so you can use perldoc to read it by executing the 'perldoc DBI::FAQ' command.

#### AUTHORS

DBI by Tim Bunce. This pod text by Tim Bunce, J. Douglas Dunlop, Jonathan Leffler and others. Perl by Larry Wall and the perl5-porters.

#### COPYRIGHT

The DBI module is Copyright (c) 1995-1999 Tim Bunce. England. All rights reserved.

You may distribute under the terms of either the GNU General Public License or the Artistic License, as specified in the Perl README file.

#### ACKNOWLEDGEMENTS

I would like to acknowledge the valuable contributions of the many people I have worked with on the DBI project, especially in the early years (1992-1994). In no particular order: Kevin Stock, Buzz Moschetti,

Kurt Andersen, Ted Lemon, William Hails, Garth Kennedy, Michael Pepler, Neil S. Briscoe, Jeff Urlwin, David J. Hughes, Jeff Stander, Forrest D. Whitcher, Larry Wall, Jeff Fried, Roy Johnson, Paul Hudson, Georg Rehfeld, Steve Sizemore, Ron Pool, Jon Meek, Tom Christiansen, Steve Baumgarten, Randal Schwartz, and a whole lot more.

#### TRANSLATIONS

A German translation of this manual and other Perl module docs (all probably slightly out of date) is available, thanks to O'Reilly, at:

<http://www.oreilly.de/catalog/perlmodger/manpages.html>

Some other translations:

<http://cronopio.net/perl/> - Spanish  
<http://member.nifty.ne.jp/hippo2000/dbimemo.htm> - Japanese

#### SUPPORT / WARRANTY

The DBI is free software. IT COMES WITHOUT WARRANTY OF ANY KIND.

Commercial support for Perl and the DBI, DBD::Oracle and Oraperl modules can be arranged via The Perl Clinic. See <http://www.perlclinic.com> for more details.

#### TRAINING

References to DBI related training resources. No recommendation implied.

<http://www.treepax.co.uk>  
<http://www.keller.com/dbweb>

#### OUTSTANDING ISSUES TO DO

- data types (ISO type numbers and type name conversions)
- error handling
- data dictionary methods
- test harness support methods
- portability
- blob\_read
- etc

#### FREQUENTLY ASKED QUESTIONS

See the DBI FAQ for a more comprehensive list of FAQs. Use the 'perldoc DBI::FAQ' command to read it.

How fast is the DBI?

To measure the speed of the DBI and DBD::Oracle code I modified DBD::Oracle such that you can set an attribute which will cause the same row to be fetched from the row cache over and over again (without involving Oracle code but exercising \*all\* the DBI and DBD::Oracle code in the code path for a fetch).

The results (on my lightly loaded old Sparc 10) fetching 50000 rows using:

```
1 while $csr->fetch;
```

were: one field: 5300 fetches per cpu second (approx) ten fields: 4000 fetches per cpu second (approx)

Obviously results will vary between platforms (newer faster platforms can reach around 50000 fetches per second) but it does give a feel for the maximum performance: fast. By way of comparison, using the code:

```
1 while @row = $csr->fetchrow_array;
```

(fetchrow\_array is roughly the same as ora\_fetch) gives:

one field: 3100 fetches per cpu second (approx)  
ten fields: 1000 fetches per cpu second (approx)

Notice the slowdown and the more dramatic impact of extra fields. (The fields were all one char long. The impact would be even bigger for longer strings.)

Changing that slightly to represent actually `_doing_` something in perl with the fetched data:

```
while(@row = $csr->fetchrow_array) {  
    $hash{++$i} = [ @row ];  
}
```

gives: ten fields: 500 fetches per cpu second (approx)

That simple addition has *\*halved\** the performance.

I therefore conclude that DBI and DBD::Oracle overheads are small compared with Perl language overheads (and probably database overheads).

So, if you think the DBI or your driver is slow, try replacing your fetch loop with just:

```
1 while $csr->fetch;
```

and time that. If that helps then point the finger at your own code. If that doesn't help much then point the finger at the database, the platform, the network etc. But think carefully before pointing it at the DBI or your driver.

(Having said all that, if anyone can show me how to make the DBI or drivers even more efficient, I'm all ears.)

Why doesn't my CGI script work right?

Read the information in the references below. Please do *\*not\** post CGI related questions to the dbi-users mailing list (or to me).

<http://www.perl.com/perl/faq/idiots-guide.html>  
<http://www3.pair.com/webthing/docs/cgi/faqs/cgifaq.shtml>  
<http://www.perl.com/perl/faq/perl-cgi-faq.html>  
<http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html>  
<http://www.boutell.com/faq/>  
<http://www.perl.com/perl/faq/>

Using Apache and mod\_perl?

<http://perl.apache.org/guide/>

General problems and good ideas:

Use the CGI::ErrorWrap module.  
Remember that many env vars won't be set for CGI scripts

How can I maintain a WWW connection to a database?

For information on the Apache httpd server and the mod\_perl module see

<http://perl.apache.org/>

What about ODBC?

A DBD::ODBC module is available.

Does the DBI have a year 2000 problem?

No. The DBI has no knowledge or understanding of dates at all.

Individual drivers (DBD::\*) may have some date handling code but are unlikely to have year 2000 related problems within their code. However, your application code which \*uses\* the DBI and DBD drivers may have year 2000 related problems if it has not been designed and written well.

See also the "Does Perl have a year 2000 problem?" section of the Perl FAQ:

<http://www.perl.com/CPAN/doc/FAQs/FAQ/PerlFAQ.html>

#### OTHER RELATED WORK AND PERL MODULES

Apache::DBI by E.Mergl@bawue.de

To be used with the Apache daemon together with an embedded perl interpreter like mod\_perl. Establishes a database connection which remains open for the lifetime of the http daemon. This way the CGI connect and disconnect for every database access becomes superfluous.

JDBC Server by Stuart 'Zen' Bishop <zen@bf.rmit.edu.au>

The server is written in Perl. The client classes that talk to it are of course in Java. Thus, a Java applet or application will be able to communicate via the JDBC API with any database that has a DBI driver installed. The URL used is in the form jdbc:dbi://host.domain.etc:999/Driver/DBName. It seems to be very similar to some commercial products, such as jdbcKona.

Remote Proxy DBD support

As of DBI 1.02, a complete implementation of a DBD::Proxy driver and the DBI::ProxyServer are part of the DBI distribution.

SQL Parser

Hugo van der Sanden <hv@crypt.compulink.co.uk>  
Stephen Zander <stephen.zander@mckesson.com>

Based on the O'Reilly lex/yacc book examples and yacc.

See also the SQL::Statement module, a very simple SQL parser and engine, base of the DBD::CSV driver.