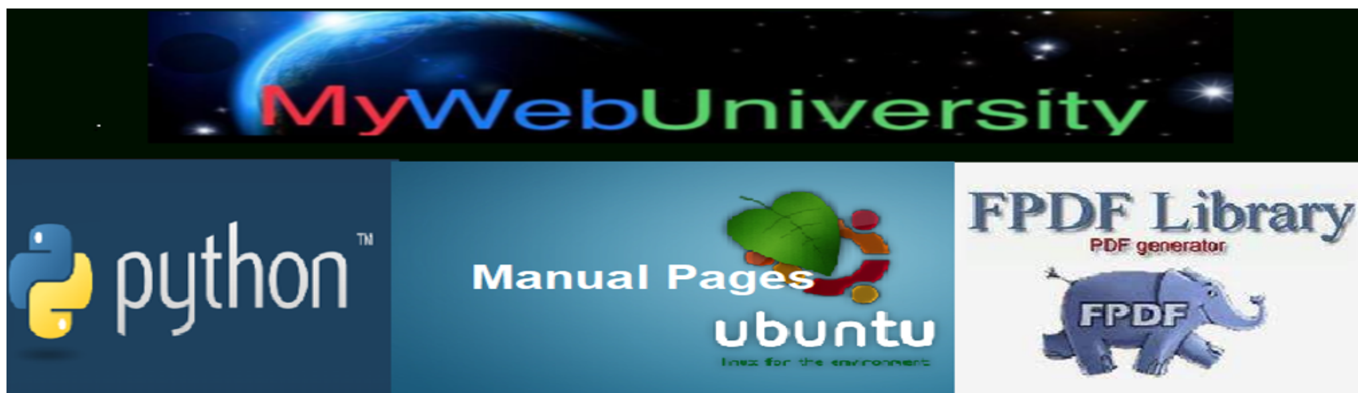


Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!

Rocky Enterprise Linux 9.2 Manual Pages on command 'DBI.3pm'

$ man DBI.3pm

DBI(3pm) User Contributed Perl Documentation DBI(3pm)

NAME

DBI - Database independent interface for Perl

SYNOPSIS

```
use DBI;

@driver_names = DBI->available_drivers;

%drivers      = DBI->installed_drivers;

@data_sources = DBI->data_sources($driver_name, \%attr);

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

$rv  = $dbh->do($statement);

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

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

$ary_ref  = $dbh->selectall_arrayref($statement);

$hash_ref = $dbh->selectall_hashref($statement, $key_field);

$ary_ref  = $dbh->selectcol_arrayref($statement);

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

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

$ary_ref  = $dbh->selectrow_arrayref($statement);

$hash_ref = $dbh->selectrow_hashref($statement);

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

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

$rc = $sth->bind_param($p_num, $bind_value);

$rc = $sth->bind_param($p_num, $bind_value, $bind_type);
```

```
$rc = $sth->bind_param($p_num, $bind_value, \%attr);
$rv = $sth->execute;
$rv = $sth->execute(@bind_values);
$rv = $sth->execute_array(\%attr, ...);
$rc = $sth->bind_col($col_num, \$col_variable);
$rc = $sth->bind_columns(@list_of_refs_to_vars_to_bind);
@row_ary  = $sth->fetchrow_array;
$ary_ref  = $sth->fetchrow_arrayref;
$hash_ref = $sth->fetchrow_hashref;
$ary_ref  = $sth->fetchall_arrayref;
$ary_ref  = $sth->fetchall_arrayref( $slice, $max_rows );
$hash_ref = $sth->fetchall_hashref( $key_field );
$rv  = $sth->rows;
$rc  = $dbh->begin_work;
$rc  = $dbh->commit;
$rc  = $dbh->rollback;
$quoted_string = $dbh->quote($string);
$rc  = $h->err;
$str = $h->errstr;
$rv  = $h->state;
$rc  = $dbh->disconnect;
```

The synopsis above only lists the major methods and parameters.

GETTING HELP

General

Before asking any questions, reread this document, consult the archives and read the DBI FAQ. The archives are listed at the end of this document and on the DBI home page <http://dbi.perl.org/support/>

You might also like to read the Advanced DBI Tutorial at <http://www.slideshare.net/Tim.Bunce/dbi-advanced-tutorial-2007>

To help you make the best use of the dbi-users mailing list, and any other lists or forums you may use, I recommend that you read "Getting Answers" by Mike Ash: <http://mikeash.com/getting_answers.html>.

Mailing Lists

If you have questions about DBI, or DBD driver modules, you can get help from the dbi-users@perl.org mailing list. This is the best way to get help. You don't have to subscribe to the list in order to post, though I'd recommend it. You can get help on subscribing and using the list by emailing dbi-users-help@perl.org.

Please note that Tim Bunce does not maintain the mailing lists or the web pages (generous volunteers do that). So please don't send mail directly to him; he just doesn't have the time to answer questions personally. The dbi-users mailing list has lots of experienced people who should be able to help you if you need it. If you do email Tim he is very likely to just forward it to the mailing list.

IRC

DBI IRC Channel: #dbi on irc.perl.org (<irc://irc.perl.org/#dbi>)

Online

StackOverflow has a DBI tag <http://stackoverflow.com/questions/tagged/dbi> with over 800 questions.

The DBI home page at <http://dbi.perl.org/> and the DBI FAQ at <http://faq.dbi-support.com/> may be worth a visit. They include links to other resources, but are rather out-dated.

Reporting a Bug

If you think you've found a bug then please read "How to Report Bugs Effectively" by Simon Tatham: <http://www.chiark.greenend.org.uk/~sgtatham/bugs.html>.

If you think you've found a memory leak then read "Memory Leaks".

Your problem is most likely related to the specific DBD driver module you're using. If that's the case then click on the 'Bugs' link on the <http://metacpan.org> page for your driver. Only submit a bug report against the DBI itself if you're sure that your issue isn't related to the driver you're using.

NOTES

This is the DBI specification that corresponds to DBI version 1.642 (see DBI::Changes for details).

The DBI is evolving at a steady pace, so it's good to check that you have the latest copy.

The significant user-visible changes in each release are documented in the DBI::Changes module so you can read them by executing "perldoc DBI::Changes".

Some DBI changes require changes in the drivers, but the drivers can take some time to catch up. Newer versions of the DBI have added features that may not yet be supported by

the drivers you use. Talk to the authors of your drivers if you need a new feature that is not yet supported.

Features added after DBI 1.21 (February 2002) are marked in the text with the version number of the DBI release they first appeared in.

Extensions to the DBI API often use the "DBIx::*" namespace. See "Naming Conventions and Name Space". DBI extension modules can be found at <https://metacpan.org/search?q=DBIx>. And all modules related to the DBI can be found at <https://metacpan.org/search?q=DBI>.

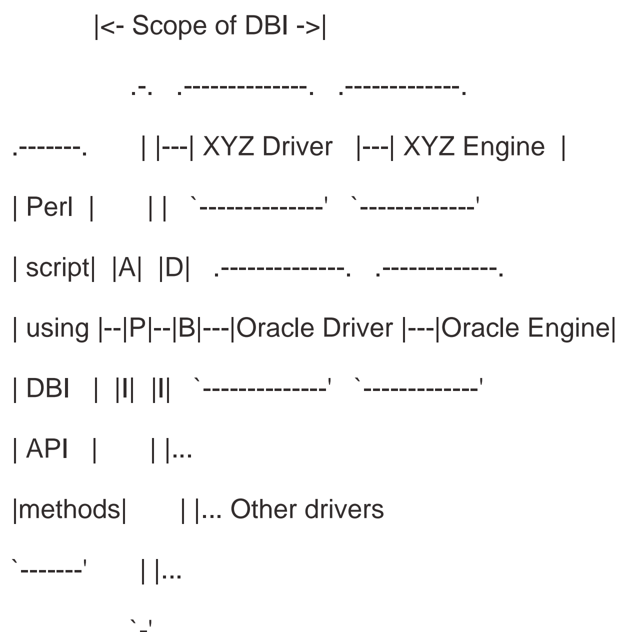
DESCRIPTION

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

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

This document often uses terms like references, objects, methods. If you're not familiar with those terms then it would be a good idea to read at least the following perl manuals first: perlreftut, perldsc, perllol, and perlboot.

Architecture of a DBI Application



```
             |<- Scope of DBI ->|
                  .-.   .--------------.   .-------------.
  .-------.       | |---| XYZ Driver   |---| XYZ Engine  |
  | Perl  |       | |   `--------------'   `-------------'
  | script|  |A|  |D|   .--------------.   .-------------.
  | using |--|P|--|B|---|Oracle Driver |---|Oracle Engine|
  | DBI   |  |I|  |I|   `--------------'   `-------------'
  | API   |       | |...
  |methods|       | |... Other drivers
  `-------'       | |...
                  `-'
```

The API, or Application Programming Interface, defines the call interface and variables for Perl scripts to use. The API is implemented by the Perl DBI 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 and handling, providing default implementations for methods, and many other non-database specific duties.

Each driver contains implementations of the DBI methods using the private interface functions of the corresponding database engine. Only authors of sophisticated/multi-database applications or generic library functions need be concerned with drivers.

Notation and Conventions

The following conventions are used in this document:

```
$dbh    Database handle object
$sth    Statement handle object
$drh    Driver handle object (rarely seen or used in applications)
$h      Any of the handle types above ($dbh, $sth, or $drh)
$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 handle objects if all references to them are deleted.

Outline Usage

To use DBI, first you need to load the DBI module:

```
use DBI;
use strict;
```

(The "use strict;" isn't required but is strongly recommended.)

Then you need to "connect" 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. This determines whether changes are automatically committed to the database when executed, or need to be explicitly committed later.

The DBI allows an application to "prepare" statements for later execution. A prepared statement is identified by a statement handle held in a Perl variable. We'll call the Perl variable $sth in our examples.

The typical method call sequence for a "SELECT" statement is:

```
  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";
  }
```

For queries that are not executed many times at once, it is often cleaner to use the higher level select wrappers:

```
  $row_hashref = $dbh->selectrow_hashref("SELECT foo, bar FROM table WHERE baz=?", undef, $baz);
  $arrayref_of_row_hashrefs = $dbh->selectall_arrayref(
    "SELECT foo, bar FROM table WHERE baz BETWEEN ? AND ?",
    { Slice => {} }, $baz_min, $baz_max);
```

The typical method call sequence for a non-"SELECT" statement is:

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

for example:

```
  $sth = $dbh->prepare("INSERT INTO table(foo,bar,baz) VALUES (?,?,?)");
  while(<CSV>) {
    chomp;
    my ($foo,$bar,$baz) = split /,/;
        $sth->execute( $foo, $bar, $baz );
```

```
}
```

The "do()" method is a wrapper of prepare and execute that can be simpler for non repeated non-"SELECT" statements (or with drivers that don't support placeholders):

```
$rows_affected = $dbh->do("UPDATE your_table SET foo = foo + 1");
$rows_affected = $dbh->do("DELETE FROM table WHERE foo = ?", undef, $foo);
```

To commit your changes to the database (when "AutoCommit" is off):

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

Finally, when you have finished working with the data source, you should "disconnect" 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. 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. Beware 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 current default format of the corresponding database engine. Time zone effects are database/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.

Perl supports two kinds of strings: Unicode (utf8 internally) and non-Unicode (defaults to iso-8859-1 if forced to assume an encoding). Drivers should accept both kinds of strings and, if required, convert them to the character set of the database being used. Similarly, when fetching from the database character data that isn't iso-8859-1 the driver should convert it into utf8.

Multiple SQL statements may not be combined in a single statement handle ($sth), although some databases and drivers do support this (notably Sybase and SQL Server).

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 (see <http://www.perl.com/CPAN/modules/by-module/DBIx/>). Package names beginning with "DBD::" are reserved for use by DBI database drivers. All environment variables used by the DBI or by individual DBDs 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, ISO SQL92 etc (portable)
  MixedCase     DBI API (portable), underscores are not used.
  lower_case    Driver or database 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).

SQL - A Query Language

Most DBI drivers require applications to use a dialect of SQL (Structured Query Language) to interact with the database engine. The "Standards Reference Information" section provides links to useful information about SQL.

The DBI itself does not mandate or require any particular language to be used; it is language independent. In ODBC terms, the DBI is in "pass-thru" mode, although individual

drivers might not be. The only requirement is that queries and other statements must be expressed as a single string of characters passed as the first argument to the "prepare" or "do" methods.

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

  http://www.mcjones.org/System_R/SQL_Reunion_95/sqlr95.html

Follow the "Full Contents" then "Intergalactic dataspeak" links for the SQL history.

Placeholders and Bind Values

Some drivers support placeholders and bind values. Placeholders, also called parameter markers, are used to indicate values in a database statement 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. Note that the "?" is not enclosed in quotation marks, even when the placeholder represents a string.

Some drivers also allow placeholders like ":"name and ":"N (e.g., ":1", ":2", and so on) in addition to "?", but their use is not portable.

If the ":"N form of placeholder is supported by the driver you're using, then you should be able to use either "bind_param" or "execute" to bind values. Check your driver documentation.

Some drivers allow you to prevent the recognition of a placeholder by placing a single backslash character ("\") immediately before it. The driver will remove the backslash character and ignore the placeholder, passing it unchanged to the backend. If the driver supports this then "get_info"(9000) will return true.

With most drivers, placeholders can't be used for any element of a statement that would prevent the database server from 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. For example, the following statement won't work as expected for more than one value:

```
"SELECT name, age FROM people WHERE name IN (?)"    # wrong
"SELECT name, age FROM people WHERE name IN (?,?)"  # two names
```

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", are used to indicate NULL values. You can insert and update columns with a NULL value as you would a non-NULL value. These examples insert and update the column "age" with a NULL value:

```
$sth = $dbh->prepare(qq{
  INSERT INTO people (fullname, age) VALUES (?, ?)
});
$sth->execute("Joe Bloggs", undef);
$sth = $dbh->prepare(qq{
  UPDATE people SET age = ? WHERE fullname = ?
});
$sth->execute(undef, "Joe Bloggs");
```

However, care must be taken when trying to use NULL values in a "WHERE" clause. Consider:

```
SELECT fullname FROM people WHERE age = ?
```

Binding an "undef" (NULL) to the placeholder will not select rows which have a NULL "age"! At least for database engines that conform to the SQL standard. 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 age IS NULL"".

A common issue is to have a code fragment handle a value that could be either "defined" or "undef" (non-NULL or NULL) at runtime. A simple technique is to prepare the appropriate statement as needed, and substitute the placeholder for non-NULL cases:

```
$sql_clause = defined $age? "age = ?" : "age IS NULL";
$sth = $dbh->prepare(qq{
  SELECT fullname FROM people WHERE $sql_clause
});
$sth->execute(defined $age ? $age : ());
```

The following technique illustrates qualifying a "WHERE" clause with several columns, whose associated values ("defined" or "undef") are in a hash %h:

```
for my $col ("age", "phone", "email") {
  if (defined $h{$col}) {
    push @sql_qual, "$col = ?";
    push @sql_bind, $h{$col};
  }
  else {
    push @sql_qual, "$col IS NULL";
  }
}
$sql_clause = join(" AND ", @sql_qual);
$sth = $dbh->prepare(qq{
    SELECT fullname FROM people WHERE $sql_clause
});
$sth->execute(@sql_bind);
```

The techniques above call prepare for the SQL statement with each call to execute. Because calls to prepare() can be expensive, performance can suffer when an application iterates many times over statements like the above.

A better solution is a single "WHERE" clause that supports both NULL and non-NULL comparisons. Its SQL statement would need to be prepared only once for all cases, thus improving performance. Several examples of "WHERE" clauses that support this are presented below. But each example lacks portability, robustness, or simplicity. Whether an example is supported on your database engine depends on what SQL extensions it provides, and where it supports the "?" placeholder in a statement.

```
0)  age = ?
1)  NVL(age, xx) = NVL(?, xx)
2)  ISNULL(age, xx) = ISNULL(?, xx)
3)  DECODE(age, ?, 1, 0) = 1
4)  age = ? OR (age IS NULL AND ? IS NULL)
5)  age = ? OR (age IS NULL AND SP_ISNULL(?) = 1)
6)  age = ? OR (age IS NULL AND ? = 1)
```

Statements formed with the above "WHERE" clauses require execute statements as follows.

The arguments are required, whether their values are "defined" or "undef".

```
0,1,2,3)  $sth->execute($age);
4,5)      $sth->execute($age, $age);
6)        $sth->execute($age, defined($age) ? 0 : 1);
```

Example 0 should not work (as mentioned earlier), but may work on a few database engines anyway (e.g. Sybase). Example 0 is part of examples 4, 5, and 6, so if example 0 works, these other examples may work, even if the engine does not properly support the right hand side of the "OR" expression.

Examples 1 and 2 are not robust: they require that you provide a valid column value xx (e.g. '~') which is not present in any row. That means you must have some notion of what data won't be stored in the column, and expect clients to adhere to that.

Example 5 requires that you provide a stored procedure (SP_ISNULL in this example) that acts as a function: it checks whether a value is null, and returns 1 if it is, or 0 if not.

Example 6, the least simple, is probably the most portable, i.e., it should work with most, if not all, database engines.

Here is a table that indicates which examples above are known to work on various database engines:

```
                -----Examples------
                0  1  2  3  4  5  6
                -  -  -  -  -  -  -
Oracle 9        N  Y  N  Y  Y  ?  Y
Informix IDS 9  N  N  N  Y  N  Y  Y
MS SQL          N  N  Y  N  Y  ?  Y
Sybase          Y  N  N  N  N  N  Y
AnyData,DBM,CSV Y  N  N  N  Y  Y* Y
SQLite 3.3      N  N  N  N  Y  N  N
MSAccess        N  N  N  N  Y  N  Y
```

* Works only because Example 0 works.

DBI provides a sample perl script that will test the examples above on your database engine and tell you which ones work. It is located in the ex/ subdirectory of the DBI source distribution, or here:

<https://github.com/perl5-dbi/dbi/blob/master/ex/perl_dbi_nulls_test.pl> Please use the

script to help us fill-in and maintain this table.

Performance

Without using placeholders, the insert statement shown previously would have to contain the literal values to be inserted and 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 (?, ?, ?)
}) or die $dbh->errstr;
while (<>) {
    chomp;
    my ($product_code, $qty, $price) = split /,/;
    $sth->execute($product_code, $qty, $price) or die $dbh->errstr;
}
$dbh->commit or die $dbh->errstr;
```

See "execute" and "bind_param" 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 "Quote and Quote-like Operators" in perlop for more details.

See also the "bind_columns" method, which is used to associate Perl variables with the output columns of a "SELECT" statement.

THE DBI PACKAGE AND CLASS

In this section, we cover the DBI class methods, utility functions, and the dynamic attributes associated with generic DBI handles.

DBI Constants

Constants representing the values of the SQL standard types can be imported individually by name, or all together by importing the special ":sql_types" tag.

The names and values of all the defined SQL standard types can be produced like this:

```
foreach (@{ $DBI::EXPORT_TAGS{sql_types} }) {
  printf "%s=%d\n", $_, &{"DBI::$_"};
```

}

These constants are defined by SQL/CLI, ODBC or both. "SQL_BIGINT" has conflicting codes in SQL/CLI and ODBC, DBI uses the ODBC one.

See the "type_info", "type_info_all", and "bind_param" methods for possible uses.

Note that just because the DBI defines a named constant for a given data type doesn't mean that drivers will support that data type.

DBI Class Methods

The following methods are provided by the DBI class:

"parse_dsn"

```
($scheme, $driver, $attr_string, $attr_hash, $driver_dsn) = DBI->parse_dsn($dsn)
    or die "Can't parse DBI DSN '$dsn'";
```

Breaks apart a DBI Data Source Name (DSN) and returns the individual parts. If $dsn doesn't contain a valid DSN then parse_dsn() returns an empty list.

$scheme is the first part of the DSN and is currently always 'dbi'. $driver is the driver name, possibly defaulted to $ENV{DBI_DRIVER}, and may be undefined. $attr_string is the contents of the optional attribute string, which may be undefined. If $attr_string is not empty then $attr_hash is a reference to a hash containing the parsed attribute names and values. $driver_dsn is the last part of the DBI DSN string. For example:

```
($scheme, $driver, $attr_string, $attr_hash, $driver_dsn)
    = DBI->parse_dsn("dbi:MyDriver(RaiseError=>1):db=test;port=42");
$scheme      = 'dbi';
$driver      = 'MyDriver';
$attr_string = 'RaiseError=>1';
$attr_hash   = { 'RaiseError' => '1' };
$driver_dsn  = 'db=test;port=42';
```

The parse_dsn() method was added in DBI 1.43.

"connect"

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

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

connection.

If the connect fails (see below), it returns "undef" and sets both $DBI::err and

$DBI::errstr. (It does not explicitly set $!.) You should generally 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 database and keep a copy of each

returned database handle.

The $data_source value must begin with ""dbi:"driver_name":"". The driver_name 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., the $data_source prefix is ""dbi::""), the environment variable

"DBI_DRIVER" is used. If neither variable is set, then "connect" dies.

Examples of $data_source values are:

```
  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.

It is recommended that drivers support the ODBC style, shown in the last example above. It

is also recommended that they support the three common names '"host"', '"port"', and

'"database"' (plus '"db"' as an alias for "database"). This simplifies automatic

construction of basic DSNs: "dbi:$driver:database=$db;host=$host;port=$port". Drivers

should aim to 'do something reasonable' when given a DSN in this form, but if any part is

meaningless for that driver (such as 'port' for Informix) it should generate an error if

that part is not empty.

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:

```
  $ENV{DBI_AUTOPROXY};dsn=$data_source
```

"DBI_AUTOPROXY" is typically set as ""dbi:Proxy:hostname=...;port=..."". If

$ENV{DBI_AUTOPROXY} doesn't begin with '"dbi:"' then "dbi:Proxy:" will be prepended to it

first. See the DBD::Proxy documentation 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. However, the everyday use of these environment variables is not recommended for security reasons. The mechanism is primarily intended to simplify testing. See below for alternative way to specify the username and password.

"DBI->connect" automatically installs the driver if it has not been installed yet. Driver installation either returns a valid driver handle, or it dies with an error message that 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, in which case $DBI::errstr will hold the error message. Use "eval" if you need to catch the ""install_driver"" 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 environment variables if no $data_source is specified.)

The "AutoCommit" and "PrintError" attributes for each connection default to "on". (See "AutoCommit" and "PrintError" for more information.) However, it is strongly recommended that you explicitly define "AutoCommit" rather than rely on the default. The "PrintWarn" attribute defaults to true. The "RaiseWarn" attribute defaults to false.

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

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

The username and password can also be specified using the attributes "Username" and "Password", in which case they take precedence over the $username and $password parameters.

You can also define connection attribute values within the $data_source parameter. For

example:

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

Individual attributes values specified in this way take precedence over any conflicting values specified via the "\%attr" parameter to "connect".

The "dbi_connect_method" attribute can be used to specify which driver method should be called to establish the connection. The only useful values are 'connect', 'connect_cached', or some specialized case like 'Apache::DBI::connect' (which is automatically the default when running within Apache).

Where possible, each session ($dbh) is independent from the transactions in other sessions. This is useful when you need to hold cursors open across transactions--for example, if you 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 $data_source should not start with ""dbi:driver_name:"". (If it does, the embedded driver_name will be ignored). Also note that in this older form of "connect", the "$dbh->{AutoCommit}" attribute is undefined, the "$dbh->{PrintError}" attribute is off, and the old "DBI_DBNAME" environment variable is checked if "DBI_DSN" is not defined. Beware that this "old-style" "connect" will soon be withdrawn in a future version of DBI.

"connect_cached"

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

"connect_cached" is like "connect", except that the database handle returned is also 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 if it is still valid. The cached database handle is replaced with a new connection if it has been disconnected or if the "ping" method fails.

Note that the behaviour of this method differs in several respects from the behaviour of persistent connections implemented by Apache::DBI. However, if Apache::DBI is loaded then

"connect_cached" will use it.

Caching connections can be useful in some applications, but it can also cause problems, such as too many connections, and so should be used with care. In particular, avoid changing the attributes of a database handle created via connect_cached() because it will affect other code that may be using the same handle. When connect_cached() returns a handle the attributes will be reset to their initial values. This can cause problems, especially with the "AutoCommit" attribute.

Also, to ensure that the attributes passed are always the same, avoid passing references inline. For example, the "Callbacks" attribute is specified as a hash reference. Be sure to declare it external to the call to connect_cached(), such that the hash reference is not re-created on every call. A package-level lexical works well:

```
package MyDBH;
my $cb = {
    'connect_cached.reused' => sub { delete $_[4]->{AutoCommit} },
};
sub dbh {
    DBI->connect_cached( $dsn, $username, $auth, { Callbacks => $cb });
}
```

Where multiple separate parts of a program are using connect_cached() to connect to the same database with the same (initial) attributes it is a good idea to add a private attribute to the connect_cached() call to effectively limit the scope of the caching. For example:

```
DBI->connect_cached(..., { private_foo_cachekey => "Bar", ... });
```

Handles returned from that connect_cached() call will only be returned by other connect_cached() call elsewhere in the code if those other calls also pass in the same attribute values, including the private one. (I've used "private_foo_cachekey" here as an example, you can use any attribute name with a "private_" prefix.)

Taking that one step further, you can limit a particular connect_cached() call to return handles unique to that one place in the code by setting the private attribute to a unique value for that place:

```
DBI->connect_cached(..., { private_foo_cachekey => __FILE__.__LINE__, ... });
```

By using a private attribute you still get connection caching for the individual calls to connect_cached() but, by making separate database connections for separate parts of the

code, the database handles are isolated from any attribute changes made to other handles.

The cache can be accessed (and cleared) via the "CachedKids" attribute:

```
my $CachedKids_hashref = $dbh->{Driver}->{CachedKids};
%$CachedKids_hashref = () if $CachedKids_hashref;
```

"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 is 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.

"installed_drivers"

```
%drivers = DBI->installed_drivers();
```

Returns a list of driver name and driver handle pairs for all drivers 'installed' (loaded) into the current process. The driver name does not include the 'DBD::' prefix.

To get a list of all drivers available in your perl installation you can use "available_drivers".

Added in DBI 1.49.

"installed_versions"

```
DBI->installed_versions;
@ary  = DBI->installed_versions;
$hash = DBI->installed_versions;
```

Calls available_drivers() and attempts to load each of them in turn using install_driver(). For each load that succeeds the driver name and version number are added to a hash. When running under DBI::PurePerl drivers which appear not be pure-perl are ignored.

When called in array context the list of successfully loaded drivers is returned (without the 'DBD::' prefix).

When called in scalar context an extra entry for the "DBI" is added (and "DBI::PurePerl" if appropriate) and a reference to the hash is returned.

When called in a void context the installed_versions() method will print out a formatted list of the hash contents, one per line, along with some other information about the DBI version and OS.

Due to the potentially high memory cost and unknown risks of loading in an unknown number of drivers that just happen to be installed on the system, this method is not recommended for general use. Use available_drivers() instead.

The installed_versions() method is primarily intended as a quick way to see from the command line what's installed. For example:

```
  perl -MDBI -e 'DBI->installed_versions'
```

The installed_versions() method was added in DBI 1.38.

"data_sources"

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

Returns a list of data sources (databases) available via the named driver. If $driver is empty or "undef", then the value of the "DBI_DRIVER" environment variable is used.

The driver will be loaded if it hasn't been already. Note that if the driver loading fails then data_sources() dies with an error message that includes the string ""install_driver"" and the underlying problem.

Data sources are returned in a form suitable for passing to the "connect" method (that is, they will include the ""dbi:$driver:"" prefix).

Note that many drivers have no way of knowing what data sources might be available for it. These drivers return an empty or incomplete list or may require driver-specific attributes.

There is also a data_sources() method defined for database handles.

"trace"

```
  DBI->trace($trace_setting)
  DBI->trace($trace_setting, $trace_filename)
  DBI->trace($trace_setting, $trace_filehandle)
  $trace_setting = DBI->trace;
```

The "DBI->trace" method sets the global default trace settings and returns the previous trace settings. It can also be used to change where the trace output is sent.

There's a similar method, "$h->trace", which sets the trace settings for the specific handle it's called on.

See the "TRACING" section for full details about the DBI's powerful tracing facilities.

"visit_handles"

```
  DBI->visit_handles( $coderef );
```

```
DBI->visit_handles( $coderef, $info );
```

Where $coderef is a reference to a subroutine and $info is an arbitrary value which, if undefined, defaults to a reference to an empty hash. Returns $info.

For each installed driver handle, if any, $coderef is invoked as:

```
$coderef->($driver_handle, $info);
```

If the execution of $coderef returns a true value then "visit_child_handles" is called on that child handle and passed the returned value as $info.

For example:

```
my $info = $dbh->{Driver}->visit_child_handles(sub {
    my ($h, $info) = @_;
    ++$info->{ $h->{Type} }; # count types of handles (dr/db/st)
    return $info; # visit kids
});
```

See also "visit_child_handles".

DBI Utility Functions

In addition to the DBI methods listed in the previous section, the DBI package also provides several utility functions.

These can be imported into your code by listing them in the "use" statement. For example:

```
use DBI qw(neat data_diff);
```

Alternatively, all these utility functions (except hash) can be imported using the ":utils" import tag. For example:

```
use DBI qw(:utils);
```

"data_string_desc"

```
$description = data_string_desc($string);
```

Returns an informal description of the string. For example:

```
UTF8 off, ASCII, 42 characters 42 bytes
UTF8 off, non-ASCII, 42 characters 42 bytes
UTF8 on, non-ASCII, 4 characters 6 bytes
UTF8 on but INVALID encoding, non-ASCII, 4 characters 6 bytes
UTF8 off, undef
```

The initial "UTF8" on/off refers to Perl's internal SvUTF8 flag. If $string has the SvUTF8 flag set but the sequence of bytes it contains are not a valid UTF-8 encoding then data_string_desc() will report "UTF8 on but INVALID encoding".

The "ASCII" vs "non-ASCII" portion shows "ASCII" if all the characters in the string are ASCII (have code points <= 127).

The data_string_desc() function was added in DBI 1.46.

"data_string_diff"

```
  $diff = data_string_diff($a, $b);
```

Returns an informal description of the first character difference between the strings. If both $a and $b contain the same sequence of characters then data_string_diff() returns an empty string. For example:

```
 Params a & b     Result
 ------------     ------
 'aaa', 'aaa'     ''
 'aaa', 'abc'     'Strings differ at index 2: a[2]=a, b[2]=b'
 'aaa', undef     'String b is undef, string a has 3 characters'
 'aaa', 'aa'      'String b truncated after 2 characters'
```

Unicode characters are reported in "\x{XXXX}" format. Unicode code points in the range U+0800 to U+08FF are unassigned and most likely to occur due to double-encoding. Characters in this range are reported as "\x{08XX}='C'" where "C" is the corresponding latin-1 character.

The data_string_diff() function only considers logical characters and not the underlying encoding. See "data_diff" for an alternative.

The data_string_diff() function was added in DBI 1.46.

"data_diff"

```
  $diff = data_diff($a, $b);
  $diff = data_diff($a, $b, $logical);
```

Returns an informal description of the difference between two strings. It calls "data_string_desc" and "data_string_diff" and returns the combined results as a multi-line string.

For example, "data_diff("abc", "ab\x{263a}")" will return:

```
  a: UTF8 off, ASCII, 3 characters 3 bytes
  b: UTF8 on, non-ASCII, 3 characters 5 bytes
  Strings differ at index 2: a[2]=c, b[2]=\x{263A}
```

If $a and $b are identical in both the characters they contain and their physical encoding then data_diff() returns an empty string. If $logical is true then physical encoding

differences are ignored (but are still reported if there is a difference in the

characters).

The data_diff() function was added in DBI 1.46.

"neat"

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

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

Strings will be quoted, although internal quotes will not be escaped. Values known to be numeric will be unquoted. Undefined (NULL) values will be shown as "undef" (without quotes).

If the string is flagged internally as utf8 then double quotes will be used, otherwise single quotes are used and 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 "trace" output. It should typically not be used for formatting values for database use. (See also "quote".)

"neat_list"

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

Calls "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 = 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.

"hash"

```
  $hash_value = DBI::hash($buffer, $type);
```

Return a 32-bit integer 'hash' value corresponding to the contents of $buffer. The $type parameter selects which kind of hash algorithm should be used.

For the technically curious, type 0 (which is the default if $type isn't specified) is based on the Perl 5.1 hash except that the value is forced to be negative (for obscure

historical reasons). Type 1 is the better "Fowler / Noll / Vo" (FNV) hash. See
<http://www.isthe.com/chongo/tech/comp/fnv/> for more information. Both types are
implemented in C and are very fast.

This function doesn't have much to do with databases, except that it can sometimes be
handy to store such values in a database. It also doesn't have much to do with perl
hashes, like %foo.

"sql_type_cast"

```
  $sts = DBI::sql_type_cast($sv, $sql_type, $flags);
```

sql_type_cast attempts to cast $sv to the SQL type (see DBI Constants) specified in
$sql_type. At present only the SQL types "SQL_INTEGER", "SQL_DOUBLE" and "SQL_NUMERIC" are
supported.

For "SQL_INTEGER" the effect is similar to using the value in an expression that requires
an integer. It gives the perl scalar an 'integer aspect'. (Technically the value gains an
IV, or possibly a UV or NV if the value is too large for an IV.)

For "SQL_DOUBLE" the effect is similar to using the value in an expression that requires a
general numeric value. It gives the perl scalar a 'numeric aspect'. (Technically the
value gains an NV.)

"SQL_NUMERIC" is similar to "SQL_INTEGER" or "SQL_DOUBLE" but more general and more
cautious. It will look at the string first and if it looks like an integer (that will fit
in an IV or UV) it will act like "SQL_INTEGER", if it looks like a floating point value it
will act like "SQL_DOUBLE", if it looks like neither then it will do nothing - and thereby
avoid the warnings that would be generated by "SQL_INTEGER" and "SQL_DOUBLE" when given
non-numeric data.

$flags may be:

"DBIstcf_DISCARD_STRING"

If this flag is specified then when the driver successfully casts the bound perl
scalar to a non-string type then the string portion of the scalar will be discarded.

"DBIstcf_STRICT"

If $sv cannot be cast to the requested $sql_type then by default it is left untouched
and no error is generated. If you specify "DBIstcf_STRICT" and the cast fails, this
will generate an error.

The returned $sts value is:

```
  -2 sql_type is not handled
```

-1 sv is undef so unchanged

0 sv could not be cast cleanly and DBIstcf_STRICT was used

1 sv could not be cast and DBIstcf_STRICT was not used

2 sv was cast successfully

This method is exported by the :utils tag and was introduced in DBI 1.611.

DBI Dynamic Attributes

Dynamic attributes are always associated with the last handle used (that handle is represented by $h in the descriptions below).

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, they have a short lifespan: because they are associated with the last handle used, they should only be used immediately after calling the method that "sets" them. 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 documentation for the "rows" method.

$DBI::lasth

Returns the DBI object handle used for the most recent DBI method call. If the last DBI method call was a DESTROY then $DBI::lasth will return the handle of the parent of the destroyed handle, if there is one.

METHODS COMMON TO ALL HANDLES

The following methods can be used by all types of DBI handles.

"err"

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

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

The DBI resets $h->err to undef before almost all DBI method calls, so the value only has

a short lifespan. Also, for most drivers, the statement handles share the same error variable as the parent database handle, so calling a method on one handle may reset the error on the related handles.

(Methods which don't reset err before being called include err() and errstr(), obviously, state(), rows(), func(), trace(), trace_msg(), ping(), and the tied hash attribute FETCH() and STORE() methods.)

If you need to test for specific error conditions 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.

The DBI uses the value of $DBI::stderr as the "err" value for internal errors. Drivers should also do likewise. The default value for $DBI::stderr is 2000000000.

A driver may return 0 from err() to indicate a warning condition after a method call. Similarly, a driver may return an empty string to indicate a 'success with information' condition. In both these cases the value is false but not undef. The errstr() and state() methods may be used to retrieve extra information in these cases.

See "set_err" for more information.

"errstr"

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

Returns the native database engine error message from the last DBI method called. This has the same lifespan issues as the "err" method described above.

The returned string may contain multiple messages separated by newline characters.

The errstr() method should not be used to test for errors, use err() for that, because drivers may return 'success with information' or warning messages via errstr() for methods that have not 'failed'.

See "set_err" for more information.

"state"

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

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

The driver is free to return any value via "state", e.g., warning codes, even if it has not declared an error by returning a true value via the "err" method described above.

The state() method should not be used to test for errors, use err() for that, because drivers may return a 'success with information' or warning state code via state() for methods that have not 'failed'.

"set_err"

```
$rv = $h->set_err($err, $errstr);
$rv = $h->set_err($err, $errstr, $state);
$rv = $h->set_err($err, $errstr, $state, $method);
$rv = $h->set_err($err, $errstr, $state, $method, $rv);
```

Set the "err", "errstr", and "state" values for the handle. This method is typically only used by DBI drivers and DBI subclasses.

If the "HandleSetErr" attribute holds a reference to a subroutine it is called first. The subroutine can alter the $err, $errstr, $state, and $method values. See "HandleSetErr" for full details. If the subroutine returns a true value then the handle "err", "errstr", and "state" values are not altered and set_err() returns an empty list (it normally returns $rv which defaults to undef, see below).

Setting "err" to a true value indicates an error and will trigger the normal DBI error handling mechanisms, such as "RaiseError" and "HandleError", if they are enabled, when execution returns from the DBI back to the application.

Setting "err" to "" indicates an 'information' state, and setting it to "0" indicates a 'warning' state. Setting "err" to "undef" also sets "errstr" to undef, and "state" to "", irrespective of the values of the $errstr and $state parameters.

The $method parameter provides an alternate method name for the "RaiseError"/"PrintError"/"RaiseWarn"/"PrintWarn" error string instead of the fairly unhelpful '"set_err"'.

The "set_err" method normally returns undef. The $rv parameter provides an alternate return value.

Some special rules apply if the "err" or "errstr" values for the handle are already set...

If "errstr" is true then: "" [err was %s now %s]"" is appended if $err is true and "err" is already true and the new err value differs from the original one. Similarly "" [state was %s now %s]"" is appended if $state is true and "state" is already true and the new state value differs from the original one. Finally ""\n"" and the new $errstr are appended if $errstr differs from the existing errstr value. Obviously the %s's above are replaced by the corresponding values.

The handle "err" value is set to $err if: $err is true; or handle "err" value is undef; or $err is defined and the length is greater than the handle "err" length. The effect is that an 'information' state only overrides undef; a 'warning' overrides undef or 'information', and an 'error' state overrides anything.

The handle "state" value is set to $state if $state is true and the handle "err" value was set (by the rules above).

Support for warning and information states was added in DBI 1.41.

"trace"

```
$h->trace($trace_settings);
$h->trace($trace_settings, $trace_filename);
$trace_settings = $h->trace;
```

The trace() method is used to alter the trace settings for a handle (and any future children of that handle). It can also be used to change where the trace output is sent.

There's a similar method, "DBI->trace", which sets the global default trace settings.

See the "TRACING" section for full details about the DBI's powerful tracing facilities.

"trace_msg"

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

Writes $message_text to the trace file if the trace level is greater than or equal to $min_level (which defaults to 1). Can also be called as "DBI->trace_msg($msg)".

See "TRACING" for more details.

"func"

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

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.

It's also important to note that the func() method does not clear a previous error ($DBI::err etc.) and it does not trigger automatic error detection (RaiseError etc.) so you must check the return status and/or $h->err to detect errors.

(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.)

See also install_method() in DBI::DBD for how you can avoid needing to use func() and gain direct access to driver-private methods.

"can"

```
$is_implemented = $h->can($method_name);
```

Returns true if $method_name is implemented by the driver or a default method is provided by the DBI's driver base class. It returns false where a driver hasn't implemented a method and the default method is provided by the DBI's driver base class is just an empty stub.

"parse_trace_flags"

```
$trace_settings_integer = $h->parse_trace_flags($trace_settings);
```

Parses a string containing trace settings and returns the corresponding integer value used internally by the DBI and drivers.

The $trace_settings argument is a string containing a trace level between 0 and 15 and/or trace flag names separated by vertical bar (""|"") or comma ("","") characters. For example: "SQL|3|foo".

It uses the parse_trace_flag() method, described below, to process the individual trace flag names.

The parse_trace_flags() method was added in DBI 1.42.

"parse_trace_flag"

```
$bit_flag = $h->parse_trace_flag($trace_flag_name);
```

Returns the bit flag corresponding to the trace flag name in $trace_flag_name. Drivers are expected to override this method and check if $trace_flag_name is a driver specific trace flags and, if not, then call the DBI's default parse_trace_flag().

The parse_trace_flag() method was added in DBI 1.42.

"private_attribute_info"

```
$hash_ref = $h->private_attribute_info();
```

Returns a reference to a hash whose keys are the names of driver-private handle attributes available for the kind of handle (driver, database, statement) that the method was called on.

For example, the return value when called with a DBD::Sybase $dbh could look like this:

```
{
    syb_dynamic_supported => undef,
    syb_oc_version => undef,
    syb_server_version => undef,
    syb_server_version_string => undef,
```

```
  }
```

and when called with a DBD::Sybase $sth they could look like this:

```
  {
      syb_types => undef,
      syb_proc_status => undef,
      syb_result_type => undef,
  }
```

The values should be undef. Meanings may be assigned to particular values in future.

"swap_inner_handle"

```
  $rc = $h1->swap_inner_handle( $h2 );
  $rc = $h1->swap_inner_handle( $h2, $allow_reparent );
```

Brain transplants for handles. You don't need to know about this unless you want to become a handle surgeon.

A DBI handle is a reference to a tied hash. A tied hash has an inner hash that actually holds the contents. The swap_inner_handle() method swaps the inner hashes between two handles. The $h1 and $h2 handles still point to the same tied hashes, but what those hashes are tied to has been swapped. In effect $h1 becomes $h2 and vice-versa. This is powerful stuff, expect problems. Use with care.

As a small safety measure, the two handles, $h1 and $h2, have to share the same parent unless $allow_reparent is true.

The swap_inner_handle() method was added in DBI 1.44.

Here's a quick kind of 'diagram' as a worked example to help think about what's happening:

```
    Original state:
            dbh1o -> dbh1i
            sthAo -> sthAi(dbh1i)
            dbh2o -> dbh2i
    swap_inner_handle dbh1o with dbh2o:
            dbh2o -> dbh1i
            sthAo -> sthAi(dbh1i)
            dbh1o -> dbh2i
    create new sth from dbh1o:
            dbh2o -> dbh1i
            sthAo -> sthAi(dbh1i)
```

```
        dbh1o -> dbh2i
        sthBo -> sthBi(dbh2i)
    swap_inner_handle sthAo with sthBo:
        dbh2o -> dbh1i
        sthBo -> sthAi(dbh1i)
        dbh1o -> dbh2i
        sthAo -> sthBi(dbh2i)
```

"visit_child_handles"

```
$h->visit_child_handles( $coderef );
$h->visit_child_handles( $coderef, $info );
```

Where $coderef is a reference to a subroutine and $info is an arbitrary value which, if undefined, defaults to a reference to an empty hash. Returns $info.

For each child handle of $h, if any, $coderef is invoked as:

```
$coderef->($child_handle, $info);
```

If the execution of $coderef returns a true value then "visit_child_handles" is called on that child handle and passed the returned value as $info.

For example:

```
# count database connections with names (DSN) matching a pattern
my $connections = 0;
$dbh->{Driver}->visit_child_handles(sub {
    my ($h, $info) = @_;
    ++$connections if $h->{Name} =~ /foo/;
    return 0; # don't visit kids
})
```

See also "visit_handles".

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 generates a warning, 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"

Type: boolean, inherited

The "Warn" attribute enables useful warnings for certain bad practices. It is enabled by default and should only be disabled in rare circumstances. Since warnings are generated using the Perl "warn" function, they can be intercepted using the Perl $SIG{__WARN__} hook.

The "Warn" attribute is not related to the "PrintWarn" attribute.

"Active"

Type: boolean, read-only

The "Active" attribute is 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 ("$dbh->disconnect" sets "Active" off). For a statement handle it typically means that the handle is a "SELECT" that may have more data to fetch. (Fetching all the data or calling "$sth->finish" sets "Active" off.)

"Executed"

Type: boolean

The "Executed" attribute is true if the handle object has been "executed". Currently only the $dbh do() method and the $sth execute(), execute_array(), and execute_for_fetch() methods set the "Executed" attribute.

When it's set on a handle it is also set on the parent handle at the same time. So calling execute() on a $sth also sets the "Executed" attribute on the parent $dbh.

The "Executed" attribute for a database handle is cleared by the commit() and rollback() methods (even if they fail). The "Executed" attribute of a statement handle is not cleared by the DBI under any circumstances and so acts as a permanent record of whether the statement handle was ever used.

The "Executed" attribute was added in DBI 1.41.

"Kids"

Type: 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. For a statement handle, the value is zero.

"ActiveKids"

Type: integer, read-only

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

"CachedKids"

Type: hash ref

For a database handle, "CachedKids" returns a reference to the cache (hash) of statement handles created by the "prepare_cached" method. For a driver handle, returns a reference to the cache (hash) of database handles created by the "connect_cached" method.

"Type"

Type: scalar, read-only

The "Type" attribute identifies the type of a DBI handle. Returns "dr" for driver handles, "db" for database handles and "st" for statement handles.

"ChildHandles"

Type: array ref

The ChildHandles attribute contains a reference to an array of all the handles created by this handle which are still accessible. The contents of the array are weak-refs and will become undef when the handle goes out of scope. (They're cleared out occasionally.) "ChildHandles" returns undef if your perl version does not support weak references (check the Scalar::Util module). The referenced array returned should be treated as read-only.

For example, to enumerate all driver handles, database handles and statement handles:

```
sub show_child_handles {
    my ($h, $level) = @_;
    printf "%sh %s %s\n", $h->{Type}, "\t" x $level, $h;
    show_child_handles($_, $level + 1)
        for (grep { defined } @{$h->{ChildHandles}});
}
my %drivers = DBI->installed_drivers();
show_child_handles($_, 0) for (values %drivers);
```

"CompatMode"

Type: boolean, inherited

The "CompatMode" attribute is 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.

It also has the effect of disabling the 'quick FETCH' of attribute values from the handles attribute cache. So all attribute values are handled by the drivers own FETCH method. This makes them slightly slower but is useful for special-purpose drivers like DBD::Multiplex.

"InactiveDestroy"

Type: boolean

The default value, false, means a handle will be fully destroyed as normal when the last reference to it is removed, just as you'd expect.

If set true then the handle will be treated by the DESTROY as if it was no longer Active, and so the database engine related effects of DESTROYing a handle will be skipped. Think of the name as meaning 'treat the handle as not-Active in the DESTROY method'.

For a database handle, this attribute does not disable an explicit call to the disconnect method, only the implicit call from DESTROY that happens if the handle is still marked as "Active".

This attribute is specifically designed for use in Unix applications that "fork" child processes. For some drivers, when the child process exits the destruction of inherited handles cause the corresponding handles in the parent process to cease working.

Either the parent or the child process, but not both, should set "InactiveDestroy" true on all their shared handles. Alternatively, and preferably, the "AutoInactiveDestroy" can be set in the parent on connect.

To help tracing applications using fork the process id is shown in the trace log whenever a DBI or handle trace() method is called. The process id also shown for every method call if the DBI trace level (not handle trace level) is set high enough to show the trace from the DBI's method dispatcher, e.g. >= 9.

"AutoInactiveDestroy"

Type: boolean, inherited

The "InactiveDestroy" attribute, described above, needs to be explicitly set in the child process after a fork(), on every active database and statement handle. This is a problem if the code that performs the fork() is not under your control, perhaps in a third-party

module. Use "AutoInactiveDestroy" to get around this situation.

If set true, the DESTROY method will check the process id of the handle and, if different from the current process id, it will set the InactiveDestroy attribute. It is strongly recommended that "AutoInactiveDestroy" is enabled on all new code (it's only not enabled by default to avoid backwards compatibility problems).

This is the example it's designed to deal with:

```
my $dbh = DBI->connect(...);
some_code_that_forks(); # Perhaps without your knowledge
# Child process dies, destroying the inherited dbh
$dbh->do(...); # Breaks because parent $dbh is now broken
```

The "AutoInactiveDestroy" attribute was added in DBI 1.614.

"PrintWarn"

Type: boolean, inherited

The "PrintWarn" attribute controls the printing of warnings recorded by the driver. When set to a true value (the default) the DBI will check method calls to see if a warning condition has been set. If so, the DBI will effectively do a "warn("$class $method warning: $DBI::errstr")" where $class is the driver class and $method is the name of the method which failed. E.g.,

```
DBD::Oracle::db execute warning: ... warning text here ...
```

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

See also "set_err" for how warnings are recorded and "HandleSetErr" for how to influence it.

Fetching the full details of warnings can require an extra round-trip to the database server for some drivers. In which case the driver may opt to only fetch the full details of warnings if the "PrintWarn" attribute is true. If "PrintWarn" is false then these drivers should still indicate the fact that there were warnings by setting the warning string to, for example: "3 warnings".

"PrintError"

Type: boolean, inherited

The "PrintError" 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".

If desired, the warnings can be caught and processed using a $SIG{__WARN__} handler or

modules like CGI::Carp and CGI::ErrorWrap.

"RaiseWarn"

Type: boolean, inherited

The "RaiseWarn" attribute can be used to force warnings to raise exceptions rather then

simply printing them. It is "off" by default. When set "on", any method which sets

warning condition will cause the DBI to effectively do a "die("$class $method warning:

$DBI::errstr")", where $class is the driver class and $method is the name of the method

that sets warning condition. E.g.,

```
  DBD::Oracle::db execute warning: ... warning text here ...
```

If you turn "RaiseWarn" on then you'd normally turn "PrintWarn" off. If "PrintWarn" is

also on, then the "PrintWarn" is done first (naturally).

This attribute was added in DBI 1.643.

"RaiseError"

Type: boolean, inherited

The "RaiseError" attribute can be used to force errors to raise exceptions rather than

simply return error codes in the normal way. It is "off" by default. When set "on", any

method which results in an error 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 that failed. E.g.,

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

If you turn "RaiseError" on then you'd normally turn "PrintError" off. If "PrintError" is

also on, then the "PrintError" is done first (naturally).

Typically "RaiseError" is used in conjunction with "eval", or a module like Try::Tiny or

TryCatch, to catch the exception that's been thrown and handle it. For example:

```
  use Try::Tiny;
  try {
    ...
    $sth->execute();
```

```
    ...
  } catch {
    # $sth->err and $DBI::err will be true if error was from DBI
    warn $_; # print the error (which Try::Tiny puts into $_)
    ... # do whatever you need to deal with the error
  };
```

In the catch block the $DBI::lasth variable can be useful for diagnosis and reporting if you can't be sure which handle triggered the error. For example, $DBI::lasth->{Type} and $DBI::lasth->{Statement}.

See also "Transactions".

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};  # localize and turn off for this block
    ...
  }
```

The original value will automatically and reliably be restored by Perl, regardless of how the block is exited. The same logic applies to other attributes, including "PrintError".

"HandleError"

Type: code ref, inherited

The "HandleError" attribute can be used to provide your own alternative behaviour in case of errors. If set to a reference to a subroutine then that subroutine is called when an error is detected (at the same point that "RaiseError" and "PrintError" are handled). It is called also when "RaiseWarn" is enabled and a warning is detected.

The subroutine is called with three parameters: the error message string that "RaiseError", "RaiseWarn" or "PrintError" would use, the DBI handle being used, and the first value being returned by the method that failed (typically undef).

If the subroutine returns a false value then the "RaiseError", "RaiseWarn" and/or "PrintError" attributes are checked and acted upon as normal.

For example, to "die" with a full stack trace for any error:

```
  use Carp;
  $h->{HandleError} = sub { confess(shift) };
```

Or to turn errors into exceptions:

```
  use Exception; # or your own favourite exception module
  $h->{HandleError} = sub { Exception->new('DBI')->raise($_[0]) };
```

It is possible to 'stack' multiple HandleError handlers by using closures:

```
  sub your_subroutine {
    my $previous_handler = $h->{HandleError};
    $h->{HandleError} = sub {
      return 1 if $previous_handler and &$previous_handler(@_);
      ... your code here ...
    };
  }
```

Using a "my" inside a subroutine to store the previous "HandleError" value is important.

See perlsub and perlref for more information about closures.

It is possible for "HandleError" to alter the error message that will be used by "RaiseError", "RaiseWarn" and "PrintError" if it returns false. It can do that by altering the value of $_[0]. This example appends a stack trace to all errors and, unlike the previous example using Carp::confess, this will work "PrintError" as well as "RaiseError":

```
  $h->{HandleError} = sub { $_[0]=Carp::longmess($_[0]); 0; };
```

It is also possible for "HandleError" to hide an error, to a limited degree, by using "set_err" to reset $DBI::err and $DBI::errstr, and altering the return value of the failed method. For example:

```
  $h->{HandleError} = sub {
    return 0 unless $_[0] =~ /^\S+ fetchrow_arrayref failed:/;
    return 0 unless $_[1]->err == 1234; # the error to 'hide'
    $h->set_err(undef,undef);   # turn off the error
    $_[2] = [ ... ];    # supply alternative return value
    return 1;
  };
```

This only works for methods which return a single value and is hard to make reliable (avoiding infinite loops, for example) and so isn't recommended for general use! If you find a good use for it then please let me know.

"HandleSetErr"

Type: code ref, inherited

The "HandleSetErr" attribute can be used to intercept the setting of handle "err", "errstr", and "state" values. If set to a reference to a subroutine then that subroutine is called whenever set_err() is called, typically by the driver or a subclass.

The subroutine is called with five arguments, the first five that were passed to set_err(): the handle, the "err", "errstr", and "state" values being set, and the method name. These can be altered by changing the values in the @_ array. The return value affects set_err() behaviour, see "set_err" for details.

It is possible to 'stack' multiple HandleSetErr handlers by using closures. See "HandleError" for an example.

The "HandleSetErr" and "HandleError" subroutines differ in subtle but significant ways. HandleError is only invoked at the point where the DBI is about to return to the application with "err" set true. It's not invoked by the failure of a method that's been called by another DBI method. HandleSetErr, on the other hand, is called whenever set_err() is called with a defined "err" value, even if false. So it's not just for errors, despite the name, but also warn and info states. The set_err() method, and thus HandleSetErr, may be called multiple times within a method and is usually invoked from deep within driver code.

In theory a driver can use the return value from HandleSetErr via set_err() to decide whether to continue or not. If set_err() returns an empty list, indicating that the HandleSetErr code has 'handled' the 'error', the driver could then continue instead of failing (if that's a reasonable thing to do). This isn't excepted to be common and any such cases should be clearly marked in the driver documentation and discussed on the dbi-dev mailing list.

The "HandleSetErr" attribute was added in DBI 1.41.

"ErrCount"

Type: unsigned integer

The "ErrCount" attribute is incremented whenever the set_err() method records an error. It isn't incremented by warnings or information states. It is not reset by the DBI at any time.

The "ErrCount" attribute was added in DBI 1.41. Older drivers may not have been updated to use set_err() to record errors and so this attribute may not be incremented when using them.

"ShowErrorStatement"

Type: boolean, inherited

The "ShowErrorStatement" attribute can be used to cause the relevant Statement text to be appended to the error messages generated by the "RaiseError", "PrintError", "RaiseWarn" and "PrintWarn" attributes. Only applies to errors on statement handles plus the prepare(), do(), and the various "select*()" database handle methods. (The exact format of the appended text is subject to change.)

If "$h->{ParamValues}" returns a hash reference of parameter (placeholder) values then those are formatted and appended to the end of the Statement text in the error message.

"TraceLevel"

Type: integer, inherited

The "TraceLevel" attribute can be used as an alternative to the "trace" method to set the DBI trace level and trace flags for a specific handle. See "TRACING" for more details.

The "TraceLevel" attribute is especially useful combined with "local" to alter the trace settings for just a single block of code.

"FetchHashKeyName"

Type: string, inherited

The "FetchHashKeyName" attribute is used to specify whether the fetchrow_hashref() method should perform case conversion on the field names used for the hash keys. For historical reasons it defaults to '"NAME"' but it is recommended to set it to '"NAME_lc"' (convert to lower case) or '"NAME_uc"' (convert to upper case) according to your preference. It can only be set for driver and database handles. For statement handles the value is frozen when prepare() is called.

"ChopBlanks"

Type: boolean, inherited

The "ChopBlanks" 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 (although it is possible that the default may change). Applications that need specific behaviour should set the attribute as needed.

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

"LongReadLen"

Type: unsigned integer, inherited

The "LongReadLen" attribute may be used to control the maximum length of 'long' type fields (LONG, BLOB, CLOB, MEMO, etc.) which the driver will read from the database automatically when it fetches each row of data.

The "LongReadLen" attribute only relates to fetching and reading long values; it is not involved in inserting or updating them.

A value of 0 means not to automatically fetch any long data. Drivers may return undef or an empty string for long fields when "LongReadLen" is 0.

The default is typically 0 (zero) or 80 bytes but may vary between drivers. Applications fetching long fields should set this value to slightly larger than the longest long field value to be fetched.

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

Changing the value of "LongReadLen" for a statement handle after it has been "prepare"'d will typically have no effect, so it's common to set "LongReadLen" on the $dbh before calling "prepare".

For most drivers the value used here has a direct effect on the memory used by the statement handle while it's active, so don't be too generous. If you can't be sure what value to use you could execute an extra select statement to determine the longest value.

For example:

```
  $dbh->{LongReadLen} = $dbh->selectrow_array(qq{
      SELECT MAX(OCTET_LENGTH(long_column_name))
      FROM table WHERE ...
  });
  $sth = $dbh->prepare(qq{
      SELECT long_column_name, ... FROM table WHERE ...
  });
```

You may need to take extra care if the table can be modified between the first select and the second being executed. You may also need to use a different function if OCTET_LENGTH() does not work for long types in your database. For example, for Sybase use DATALENGTH() and for Oracle use LENGTHB().

See also "LongTruncOk" for information on truncation of long types.

"LongTruncOk"

Type: boolean, inherited

The "LongTruncOk" 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 so fetching a long value that needs to be truncated will cause the fetch to fail. (Applications should always be sure 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 "LongReadLen".

"TaintIn"

Type: boolean, inherited

If the "TaintIn" attribute is set to a true value and Perl is running in taint mode (e.g., started with the "-T" option), then all 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 perlsec for more about taint mode. If Perl is not running in taint mode, this attribute has no effect.

When fetching data that you trust you can turn off the TaintIn attribute, for that statement handle, for the duration of the fetch loop.

The "TaintIn" attribute was added in DBI 1.31.

"TaintOut"

Type: boolean, inherited

If the "TaintOut" attribute is set to a true value and Perl is running in taint mode (e.g., started with the "-T" option), then most data fetched from the database is considered tainted. This may change.

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

When fetching data that you trust you can turn off the TaintOut attribute, for that statement handle, for the duration of the fetch loop.

Currently only fetched data is tainted. It is possible that the results of other DBI method calls, and the value of fetched attributes, may 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.

The "TaintOut" attribute was added in DBI 1.31.

"Taint"

Type: boolean, inherited

The "Taint" attribute is a shortcut for "TaintIn" and "TaintOut" (it is also present for backwards compatibility).

Setting this attribute sets both "TaintIn" and "TaintOut", and retrieving it returns a true value if and only if "TaintIn" and "TaintOut" are both set to true values.

"Profile"

Type: inherited

The "Profile" attribute enables the collection and reporting of method call timing statistics. See the DBI::Profile module documentation for much more detail.

The "Profile" attribute was added in DBI 1.24.

"ReadOnly"

Type: boolean, inherited

An application can set the "ReadOnly" attribute of a handle to a true value to indicate that it will not be attempting to make any changes using that handle or any children of it.

Note that the exact definition of 'read only' is rather fuzzy. For more details see the documentation for the driver you're using.

If the driver can make the handle truly read-only then it should (unless doing so would have unpleasant side effect, like changing the consistency level from per-statement to per-session). Otherwise the attribute is simply advisory.

A driver can set the "ReadOnly" attribute itself to indicate that the data it is connected to cannot be changed for some reason.

If the driver cannot ensure the "ReadOnly" attribute is adhered to it will record a warning. In this case reading the "ReadOnly" attribute back after it is set true will return true even if the underlying driver cannot ensure this (so any application knows the application declared itself ReadOnly).

Library modules and proxy drivers can use the attribute to influence their behavior. For example, the DBD::Gofer driver considers the "ReadOnly" attribute when making a decision about whether to retry an operation that failed.

The attribute should be set to 1 or 0 (or undef). Other values are reserved.

"Callbacks"

Type: hash ref

The DBI callback mechanism lets you intercept, and optionally replace, any method call on a DBI handle. At the extreme, it lets you become a puppet master, deceiving the application in any way you want.

The "Callbacks" attribute is a hash reference where the keys are DBI method names and the values are code references. For each key naming a method, the DBI will execute the associated code reference before executing the method.

The arguments to the code reference will be the same as to the method, including the invocant (a database handle or statement handle). For example, say that to callback to some code on a call to "prepare()":

```
$dbh->{Callbacks} = {
    prepare => sub {
        my ($dbh, $query, $attrs) = @_;
        print "Preparing q{$query}\n"
    },
};
```

The callback would then be executed when you called the "prepare()" method:

```
$dbh->prepare('SELECT 1');
```

And the output of course would be:

```
Preparing q{SELECT 1}
```

Because callbacks are executed before the methods they're associated with, you can modify the arguments before they're passed on to the method call. For example, to make sure that all calls to "prepare()" are immediately prepared by DBD::Pg, add a callback that makes sure that the "pg_prepare_now" attribute is always set:

```
my $dbh = DBI->connect($dsn, $username, $auth, {
    Callbacks => {
        prepare => sub {
            $_[2] ||= {};
            $_[2]->{pg_prepare_now} = 1;
            return; # must return nothing
        },
    }
```

```
  });
```

Note that we are editing the contents of @_ directly. In this case we've created the attributes hash if it's not passed to the "prepare" call.

You can also prevent the associated method from ever executing. While a callback executes, $_ holds the method name. (This allows multiple callbacks to share the same code reference and still know what method was called.) To prevent the method from executing, simply "undef $_". For example, if you wanted to disable calls to "ping()", you could do this:

```
  $dbh->{Callbacks} = {
      ping => sub {
          # tell dispatch to not call the method:
          undef $_;
          # return this value instead:
          return "42 bells";
      }
  };
```

As with other attributes, Callbacks can be specified on a handle or via the attributes to "connect()". Callbacks can also be applied to a statement methods on a statement handle. For example:

```
  $sth->{Callbacks} = {
      execute => sub {
          print "Executing ", shift->{Statement}, "\n";
      }
  };
```

The "Callbacks" attribute of a database handle isn't copied to any statement handles it creates. So setting callbacks for a statement handle requires you to set the "Callbacks" attribute on the statement handle yourself, as in the example above, or use the special "ChildCallbacks" key described below.

Special Keys in Callbacks Attribute

In addition to DBI handle method names, the "Callbacks" hash reference supports four additional keys.

The first is the "ChildCallbacks" key. When a statement handle is created from a database handle the "ChildCallbacks" key of the database handle's "Callbacks" attribute, if any, becomes the new "Callbacks" attribute of the statement handle. This allows you to define

callbacks for all statement handles created from a database handle. For example, if you wanted to count how many times "execute" was called in your application, you could write:

```
my $exec_count = 0;
my $dbh = DBI->connect( $dsn, $username, $auth, {
    Callbacks => {
        ChildCallbacks => {
            execute => sub { $exec_count++; return; }
        }
    }
});
END {
    print "The execute method was called $exec_count times\n";
}
```

The other three special keys are "connect_cached.new", "connect_cached.connected", and "connect_cached.reused". These keys define callbacks that are called when "connect_cached()" is called, but allow different behaviors depending on whether a new handle is created or a handle is returned. The callback is invoked with these arguments: "$dbh, $dsn, $user, $auth, $attr".

For example, some applications uses "connect_cached()" to connect with "AutoCommit" enabled and then disable "AutoCommit" temporarily for transactions. If "connect_cached()" is called during a transaction, perhaps in a utility method, then it might select the same cached handle and then force "AutoCommit" on, forcing a commit of the transaction. See the "connect_cached" documentation for one way to deal with that. Here we'll describe an alternative approach using a callback.

Because the "connect_cached.new" and "connect_cached.reused" callbacks are invoked before "connect_cached()" has applied the connect attributes, you can use them to edit the attributes that will be applied. To prevent a cached handle from having its transactions committed before it's returned, you can eliminate the "AutoCommit" attribute in a "connect_cached.reused" callback, like so:

```
my $cb = {
    'connect_cached.reused' => sub { delete $_[4]->{AutoCommit} },
};
sub dbh {
```

```
      my $self = shift;
      DBI->connect_cached( $dsn, $username, $auth, {
          PrintError => 0,
          RaiseError => 1,
          AutoCommit => 1,
          Callbacks  => $cb,
      });
  }
```

The upshot is that new database handles are created with "AutoCommit" enabled, while cached database handles are left in whatever transaction state they happened to be in when retrieved from the cache.

Note that we've also used a lexical for the callbacks hash reference. This is because "connect_cached()" returns a new database handle if any of the attributes passed to is have changed. If we used an inline hash reference, "connect_cached()" would return a new database handle every time. Which would rather defeat the purpose.

A more common application for callbacks is setting connection state only when a new connection is made (by connect() or connect_cached()). Adding a callback to the connected method (when using "connect") or via "connect_cached.connected" (when useing connect_cached()>) makes this easy. The connected() method is a no-op by default (unless you subclass the DBI and change it). The DBI calls it to indicate that a new connection has been made and the connection attributes have all been set. You can give it a bit of added functionality by applying a callback to it. For example, to make sure that MySQL understands your application's ANSI-compliant SQL, set it up like so:

```
  my $dbh = DBI->connect($dsn, $username, $auth, {
      Callbacks => {
          connected => sub {
              shift->do(q{
                  SET SESSION sql_mode='ansi,strict_trans_tables,no_auto_value_on_zero';
              });
              return;
          },
      }
  });
```

If you're using "connect_cached()", use the "connect_cached.connected" callback, instead. This is because "connected()" is called for both new and reused database handles, but you want to execute a callback only the when a new database handle is returned. For example, to set the time zone on connection to a PostgreSQL database, try this:

```
my $cb = {
    'connect_cached.connected' => sub {
        shift->do('SET timezone = UTC');
    }
};
sub dbh {
    my $self = shift;
    DBI->connect_cached( $dsn, $username, $auth, { Callbacks => $cb });
}
```

One significant limitation with callbacks is that there can only be one per method per handle. This means it's easy for one use of callbacks to interfere with, or typically simply overwrite, another use of callbacks. For this reason modules using callbacks should document the fact clearly so application authors can tell if use of callbacks by the module will clash with use of callbacks by the application.

You might be able to work around this issue by taking a copy of the original callback and calling it within your own. For example:

```
my $prev_cb = $h->{Callbacks}{method_name};
$h->{Callbacks}{method_name} = sub {
  if ($prev_cb) {
      my @result = $prev_cb->(@_);
      return @result if not $_; # $prev_cb vetoed call
  }
  ... your callback logic here ...
};
```

"private_your_module_name_*"

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 name that the attribute relates to (e.g., ""private_YourFullModuleName_thingy"").

Because of the way the Perl tie mechanism works you cannot reliably use the "||=" operator directly to initialise the attribute, like this:

```
my $foo = $dbh->{private_yourmodname_foo} ||= { ... }; # WRONG
```

you should use a two step approach like this:

```
my $foo = $dbh->{private_yourmodname_foo};
$foo ||= $dbh->{private_yourmodname_foo} = { ... };
```

This attribute is primarily of interest to people sub-classing DBI, or for applications to piggy-back extra information onto DBI handles.

DBI DATABASE HANDLE OBJECTS

This section covers the methods and attributes associated with database handles.

Database Handle Methods

The following methods are specified for DBI database handles:

"clone"

```
$new_dbh = $dbh->clone(\%attr);
```

The "clone" method duplicates the $dbh connection by connecting with the same parameters ($dsn, $user, $password) as originally used.

The attributes for the cloned connect are the same as those used for the original connect, with any other attributes in "\%attr" merged over them. Effectively the same as doing:

```
%attributes_used = ( %original_attributes, %attr );
```

If \%attr is not given then it defaults to a hash containing all the attributes in the attribute cache of $dbh excluding any non-code references, plus the main boolean attributes (RaiseError, PrintError, AutoCommit, etc.). This behaviour is unreliable and so use of clone without an argument is deprecated and may cause a warning in a future release.

The clone method can be used even if the database handle is disconnected.

The "clone" method was added in DBI 1.33.

"data_sources"

```
@ary = $dbh->data_sources();
@ary = $dbh->data_sources(\%attr);
```

Returns a list of data sources (databases) available via the $dbh driver's data_sources() method, plus any extra data sources that the driver can discover via the connected $dbh.

Typically the extra data sources are other databases managed by the same server process that the $dbh is connected to.

Data sources are returned in a form suitable for passing to the "connect" method (that is, they will include the ""dbi:$driver:"" prefix).

The data_sources() method, for a $dbh, was added in DBI 1.38.

"do"

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

Prepare and execute a single statement. Returns the number of rows affected or "undef" on error. A return value of "-1" means the number of rows is not known, not applicable, or not available.

This method is typically most useful for non-"SELECT" statements that either cannot be prepared in advance (due to a limitation of the driver) or 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
}
```

For example:

```
my $rows_deleted = $dbh->do(q{
    DELETE FROM table
    WHERE status = ?
}, undef, 'DONE') or 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. But if you'll be executing the statement many times then it's more efficient to "prepare" it once and call "execute" many times instead.

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 "Quote and Quote-like Operators" in perlop for more details.

Note drivers are free to avoid the overhead of creating an DBI statement handle for do(), especially if there are no parameters. In this case error handlers, if invoked during do(), will be passed the database handle.

"last_insert_id"

```
  $rv = $dbh->last_insert_id();
  $rv = $dbh->last_insert_id($catalog, $schema, $table, $field);
  $rv = $dbh->last_insert_id($catalog, $schema, $table, $field, \%attr);
```

Returns a value 'identifying' the row just inserted, if possible. Typically this would be a value assigned by the database server to a column with an auto_increment or serial type. Returns undef if the driver does not support the method or can't determine the value.

The $catalog, $schema, $table, and $field parameters may be required for some drivers (see below). If you don't know the parameter values and your driver does not need them, then use "undef" for each.

There are several caveats to be aware of with this method if you want to use it for portable applications:

* For some drivers the value may only available immediately after the insert statement has executed (e.g., mysql, Informix).

* For some drivers the $catalog, $schema, $table, and $field parameters are required, for others they are ignored (e.g., mysql).

* Drivers may return an indeterminate value if no insert has been performed yet.

* For some drivers the value may only be available if placeholders have not been used (e.g., Sybase, MS SQL). In this case the value returned would be from the last non-placeholder insert statement.

* Some drivers may need driver-specific hints about how to get the value. For example, being told the name of the database 'sequence' object that holds the value. Any such hints are passed as driver-specific attributes in the \%attr parameter.

* If the underlying database offers nothing better, then some drivers may attempt to implement this method by executing ""select max($field) from $table"". Drivers using any approach like this should issue a warning if "AutoCommit" is true because it is generally

unsafe - another process may have modified the table between your insert and the select.

For situations where you know it is safe, such as when you have locked the table, you can silence the warning by passing "Warn" => 0 in \%attr.

* If no insert has been performed yet, or the last insert failed, then the value is implementation defined.

Given all the caveats above, it's clear that this method must be used with care.

The "last_insert_id" method was added in DBI 1.38.

"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 "prepare", "execute" and "fetchrow_array" into a single call. If called in a list context, it returns the first row of data from the statement. The $statement parameter can be a previously prepared statement handle, in which case the "prepare" is skipped.

If any method fails, and "RaiseError" is not set, "selectrow_array" will return an empty list.

If called in a scalar context for a statement handle that has more than one column, it is undefined whether the driver will return the value of the first column or the last. So don't do that. Also, in a scalar context, an "undef" is returned if there are no more rows or if an error occurred. That "undef" can't be distinguished from an "undef" returned because the first field value was NULL. For these reasons you should exercise some caution if you use "selectrow_array" in a scalar context, or just don't do that.

"selectrow_arrayref"

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

This utility method combines "prepare", "execute" and "fetchrow_arrayref" into a single call. It returns the first row of data from the statement. The $statement parameter can be a previously prepared statement handle, in which case the "prepare" is skipped.

If any method fails, and "RaiseError" is not set, "selectrow_arrayref" will return undef.

"selectrow_hashref"

```
  $hash_ref = $dbh->selectrow_hashref($statement);
```

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

This utility method combines "prepare", "execute" and "fetchrow_hashref" into a single call. It returns the first row of data from the statement. The $statement parameter can be a previously prepared statement handle, in which case the "prepare" is skipped.

If any method fails, and "RaiseError" is not set, "selectrow_hashref" will return undef.

"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 "prepare", "execute" and "fetchall_arrayref" into a single call. It returns a reference to an array containing a reference to an array (or hash, see below) for each row of data fetched.

The $statement parameter can be a previously prepared statement handle, in which case the "prepare" is skipped. This is recommended if the statement is going to be executed many times.

If "RaiseError" is not set and any method except "fetchall_arrayref" fails then "selectall_arrayref" will return "undef"; if "fetchall_arrayref" fails then it will return with whatever data has been fetched thus far. You should check "$dbh->err" afterwards (or use the "RaiseError" attribute) to discover if the data is complete or was truncated due to an error.

The "fetchall_arrayref" method called by "selectall_arrayref" supports a $max_rows parameter. You can specify a value for $max_rows by including a '"MaxRows"' attribute in \%attr. In which case finish() is called for you after fetchall_arrayref() returns.

The "fetchall_arrayref" method called by "selectall_arrayref" also supports a $slice parameter. You can specify a value for $slice by including a '"Slice"' or '"Columns"' attribute in \%attr. The only difference between the two is that if "Slice" is not defined and "Columns" is an array ref, then the array is assumed to contain column index values (which count from 1), rather than perl array index values. In which case the array is copied and each value decremented before passing to "/fetchall_arrayref".

You may often want to fetch an array of rows where each row is stored as a hash. That can be done simply using:

```
  my $emps = $dbh->selectall_arrayref(
```

```
    "SELECT ename FROM emp ORDER BY ename",
    { Slice => {} }
);
foreach my $emp ( @$emps ) {
    print "Employee: $emp->{ename}\n";
}
```

Or, to fetch into an array instead of an array ref:

```
@result = @{ $dbh->selectall_arrayref($sql, { Slice => {} }) };
```

See "fetchall_arrayref" method for more details.

"selectall_array"

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

This is a convenience wrapper around selectall_arrayref that returns the rows directly as a list, rather than a reference to an array of rows.

Note that if "RaiseError" is not set then you can't tell the difference between returning no rows and an error. Using RaiseError is best practice.

The "selectall_array" method was added in DBI 1.635.

"selectall_hashref"

```
$hash_ref = $dbh->selectall_hashref($statement, $key_field);
$hash_ref = $dbh->selectall_hashref($statement, $key_field, \%attr);
$hash_ref = $dbh->selectall_hashref($statement, $key_field, \%attr, @bind_values);
```

This utility method combines "prepare", "execute" and "fetchall_hashref" into a single call. It returns a reference to a hash containing one entry, at most, for each row, as returned by fetchall_hashref().

The $statement parameter can be a previously prepared statement handle, in which case the "prepare" is skipped. This is recommended if the statement is going to be executed many times.

The $key_field parameter defines which column, or columns, are used as keys in the returned hash. It can either be the name of a single field, or a reference to an array containing multiple field names. Using multiple names yields a tree of nested hashes.

If a row has the same key as an earlier row then it replaces the earlier row.

If any method except "fetchall_hashref" fails, and "RaiseError" is not set,

"selectall_hashref" will return "undef". If "fetchall_hashref" fails and "RaiseError" is not set, then it will return with whatever data it has fetched thus far. $DBI::err should be checked to catch that.

See fetchall_hashref() for more details.

"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 "prepare", "execute", 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. This is recommended if the statement is going to be executed many times.

If any method except "fetch" fails, and "RaiseError" is not set, "selectcol_arrayref" will return "undef". If "fetch" fails and "RaiseError" is not set, then it will return with whatever data it has fetched thus far. $DBI::err should be checked to catch that.

The "selectcol_arrayref" method defaults to pushing a single column value (the first) from each row into the result array. However, it can also push another column, or even multiple columns per row, into the result array. This behaviour can be specified via a '"Columns"' attribute which must be a ref to an array containing the column number or numbers to use.

For example:

```
  # get array of id and name pairs:
  my $ary_ref = $dbh->selectcol_arrayref("select id, name from table", { Columns=>[1,2] });
  my %hash = @$ary_ref; # build hash from key-value pairs so $hash{$id} => name
```

You can specify a maximum number of rows to fetch by including a '"MaxRows"' attribute in \%attr.

"prepare"

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

Prepares a 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" method. See "Statement Handle Methods".

Drivers for engines without the concept of preparing a statement will typically just store the statement in the returned handle and process it when "$sth->execute" is called. Such drivers are unlikely to give much useful information about the statement, such as "$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 Placeholders). 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 you're using a simple engine, and you need to take extra care if writing applications intended 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 normally be used with the DBI.

"prepare_cached"

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

Like "prepare" 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 $statement and %attr parameter values, then the corresponding cached $sth will be returned without contacting the database server. Be sure to understand the cautions and caveats noted below.

The $if_active parameter lets you adjust the behaviour if an already cached statement handle is still Active. There are several alternatives:

0: A warning will be generated, and finish() will be called on the statement handle before it is returned. This is the default behaviour if $if_active is not passed.

1: finish() will be called on the statement handle, but the warning is suppressed.

2: Disables any checking.

3: The existing active statement handle will be removed from the cache and a new statement handle prepared and cached in its place. This is the safest option because it doesn't

affect the state of the old handle, it just removes it from the cache. [Added in DBI 1.40]

Here are some examples of "prepare_cached":

```
sub insert_hash {
  my ($table, $field_values) = @_;
  # sort to keep field order, and thus sql, stable for prepare_cached
  my @fields = sort keys %$field_values;
  my @values = @{$field_values}{@fields};
  my $sql = sprintf "insert into %s (%s) values (%s)",
      $table, join(",", @fields), join(",", ("?")x@fields);
  my $sth = $dbh->prepare_cached($sql);
  return $sth->execute(@values);
}
sub search_hash {
  my ($table, $field_values) = @_;
  # sort to keep field order, and thus sql, stable for prepare_cached
  my @fields = sort keys %$field_values;
  my @values = @{$field_values}{@fields};
  my $qualifier = "";
  $qualifier = "where ".join(" and ", map { "$_=?" } @fields) if @fields;
  $sth = $dbh->prepare_cached("SELECT * FROM $table $qualifier");
  return $dbh->selectall_arrayref($sth, {}, @values);
}
```

Caveat emptor: This caching can be useful in some applications, but it can also cause problems and should be used with care. Here is a contrived case where caching would cause a significant problem:

```
my $sth = $dbh->prepare_cached('SELECT * FROM foo WHERE bar=?');
$sth->execute(...);
while (my $data = $sth->fetchrow_hashref) {
  # later, in some other code called within the loop...
  my $sth2 = $dbh->prepare_cached('SELECT * FROM foo WHERE bar=?');
  $sth2->execute(...);
  while (my $data2 = $sth2->fetchrow_arrayref) {
    do_stuff(...);
```

```
    }
  }
```

In this example, since both handles are preparing the exact same statement, $sth2 will not be its own statement handle, but a duplicate of $sth returned from the cache. The results will certainly not be what you expect. Typically the inner fetch loop will work normally, fetching all the records and terminating when there are no more, but now that $sth is the same as $sth2 the outer fetch loop will also terminate.

You'll know if you run into this problem because prepare_cached() will generate a warning by default (when $if_active is false).

The cache used by prepare_cached() is keyed by both the statement and any attributes so you can also avoid this issue by doing something like:

```
  $sth = $dbh->prepare_cached("...", { dbi_dummy => __FILE__.__LINE__ });
```

which will ensure that prepare_cached only returns statements cached by that line of code in that source file.

Also, to ensure the attributes passed are always the same, avoid passing references inline. For example, the Slice attribute is specified as a reference. Be sure to declare it external to the call to prepare_cached(), such that a new hash reference is not created on every call. See "connect_cached" for more details and examples.

If you'd like the cache to managed intelligently, you can tie the hashref returned by "CachedKids" to an appropriate caching module, such as Tie::Cache::LRU:

```
  my $cache;
  tie %$cache, 'Tie::Cache::LRU', 500;
  $dbh->{CachedKids} = $cache;
```

"commit"

```
  $rc  = $dbh->commit     or die $dbh->errstr;
```

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

If "AutoCommit" is on, then calling "commit" will issue a "commit ineffective with AutoCommit" warning.

See also "Transactions" in the "FURTHER INFORMATION" section below.

"rollback"

```
  $rc  = $dbh->rollback   or die $dbh->errstr;
```

Rollback (undo) the most recent series of uncommitted database changes if the database

supports transactions and AutoCommit is off.

If "AutoCommit" is on, then calling "rollback" will issue a "rollback ineffective with AutoCommit" warning.

See also "Transactions" in the "FURTHER INFORMATION" section below.

"begin_work"

```
$rc  = $dbh->begin_work   or die $dbh->errstr;
```

Enable transactions (by turning "AutoCommit" off) until the next call to "commit" or "rollback". After the next "commit" or "rollback", "AutoCommit" will automatically be turned on again.

If "AutoCommit" is already off when "begin_work" is called then it does nothing except return an error. If the driver does not support transactions then when "begin_work" attempts to set "AutoCommit" off the driver will trigger a fatal error.

See also "Transactions" in the "FURTHER INFORMATION" section below.

"disconnect"

```
$rc = $dbh->disconnect  or warn $dbh->errstr;
```

Disconnects the database from the database handle. "disconnect" is 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", as it is undefined.

Generally, if you want your changes to be committed or rolled back when you disconnect, then you should explicitly call "commit" or "rollback" before disconnecting.

If you disconnect from a database while you still have active statement handles (e.g., SELECT statement handles that may have more data to fetch), you will get a warning. The warning may indicate that a fetch loop terminated early, perhaps due to an uncaught error. To avoid the warning call the "finish" method on the active handles.

"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 current default implementation always returns true without actually doing anything. Actually, it returns ""0 but true"" which is true but zero. That way you can tell if the return value is genuine or just the default. Drivers should override this method with one that does the right thing for their type of database.

Few applications would have direct use for this method. See the specialized Apache::DBI module for one example usage.

"get_info"

```
  $value = $dbh->get_info( $info_type );
```

Returns information about the implementation, i.e. driver and data source capabilities, restrictions etc. It returns "undef" for unknown or unimplemented information types. For example:

```
  $database_version  = $dbh->get_info(  18 ); # SQL_DBMS_VER
  $max_select_tables = $dbh->get_info( 106 ); # SQL_MAXIMUM_TABLES_IN_SELECT
```

See "Standards Reference Information" for more detailed information about the information types and their meanings and possible return values.

The DBI::Const::GetInfoType module exports a %GetInfoType hash that can be used to map info type names to numbers. For example:

```
  $database_version = $dbh->get_info( $GetInfoType{SQL_DBMS_VER} );
```

The names are a merging of the ANSI and ODBC standards (which differ in some cases). See DBI::Const::GetInfoType for more details.

Because some DBI methods make use of get_info(), drivers are strongly encouraged to support at least the following very minimal set of information types to ensure the DBI itself works properly:

```
 Type  Name                        Example A     Example B
 ----  --------------------------  ------------  ----------------
   17  SQL_DBMS_NAME               'ACCESS'      'Oracle'
   18  SQL_DBMS_VER                '03.50.0000'  '08.01.0721 ...'
   29  SQL_IDENTIFIER_QUOTE_CHAR   '`'           '"'
```

41 SQL_CATALOG_NAME_SEPARATOR '.'   '@'

114 SQL_CATALOG_LOCATION   1   2

Values from 9000 to 9999 for get_info are officially reserved for use by Perl DBI. Values in that range which have been assigned a meaning are defined here:

9000: true if a backslash character ("\") before placeholder-like text (e.g. "?", ":foo") will prevent it being treated as a placeholder by the driver. The backslash will be removed before the text is passed to the backend.

"table_info"

```
$sth = $dbh->table_info( $catalog, $schema, $table, $type );
$sth = $dbh->table_info( $catalog, $schema, $table, $type, \%attr );
# then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

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

The arguments $catalog, $schema and $table may accept search patterns according to the database/driver, for example: $table = '%FOO%'; Remember that the underscore character ('"_"') is a search pattern that means match any character, so 'FOO_%' is the same as 'FOO%' and 'FOO_BAR%' will match names like 'FOO1BAR'.

The value of $type is a comma-separated list of one or more types of tables to be returned in the result set. Each value may optionally be quoted, e.g.:

```
$type = "TABLE";
$type = "'TABLE','VIEW'";
```

In addition the following special cases may also be supported by some drivers:

? If the value of $catalog is '%' and $schema and $table name are empty strings, the result set contains a list of catalog names. For example:

```
$sth = $dbh->table_info('%', '', '');
```

? If the value of $schema is '%' and $catalog and $table are empty strings, the result set contains a list of schema names.

? If the value of $type is '%' and $catalog, $schema, and $table are all empty strings, the result set contains a list of table types.

If your driver doesn't support one or more of the selection filter parameters then you may get back more than you asked for and can do the filtering yourself.

This method can be expensive, and can return a large amount of data. (For example, small Oracle installation returns over 2000 rows.) So it's a good idea to use the filters to

limit the data as much as possible.

The statement handle returned has at least the following fields in the order show below.

Other fields, after these, may also be present.

TABLE_CAT: Table catalog identifier. This field is NULL ("undef") if not applicable to the

data source, which is usually the case. This field is empty if not applicable to the

table.

TABLE_SCHEM: The name of the schema containing the TABLE_NAME value. This field is NULL

("undef") if not applicable to data source, and 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 type identifier that is specific to the data

source.

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 "tables", "Catalog Methods" and "Standards Reference Information".

"column_info"

```
  $sth = $dbh->column_info( $catalog, $schema, $table, $column );

  # then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

Returns an active statement handle that can be used to fetch information about columns in

specified tables.

The arguments $schema, $table and $column may accept search patterns according to the

database/driver, for example: $table = '%FOO%';

Note: The support for the selection criteria is driver specific. If the driver doesn't

support one or more of them then you may get back more than you asked for and can do the

filtering yourself.

Note: If your driver does not support column_info an undef is returned. This is distinct

from asking for something which does not exist in a driver which supports column_info as a

valid statement handle to an empty result-set will be returned in this case.

If the arguments don't match any tables then you'll still get a statement handle, it'll

just return no rows.

The statement handle returned has at least the following fields in the order shown below.

Other fields, after these, may also be present.

TABLE_CAT: The catalog identifier. This field is NULL ("undef") if not applicable to the data source, which is often the case. This field is empty if not applicable to the table.

TABLE_SCHEM: The schema identifier. This field is NULL ("undef") if not applicable to the data source, and empty if not applicable to the table.

TABLE_NAME: The table identifier. Note: A driver may provide column metadata not only for base tables, but also for derived objects like SYNONYMS etc.

COLUMN_NAME: The column identifier.

DATA_TYPE: The concise data type code.

TYPE_NAME: A data source dependent data type name.

COLUMN_SIZE: The column size. This is the maximum length in characters for character data types, the number of digits or bits for numeric data types or the length in the representation of temporal types. See the relevant specifications for detailed information.

BUFFER_LENGTH: The length in bytes of transferred data.

DECIMAL_DIGITS: The total number of significant digits to the right of the decimal point.

NUM_PREC_RADIX: The radix for numeric precision. The value is 10 or 2 for numeric data types and NULL ("undef") if not applicable.

NULLABLE: Indicates if a column can accept NULLs. The following values are defined:

```
SQL_NO_NULLS          0
SQL_NULLABLE          1
SQL_NULLABLE_UNKNOWN  2
```

REMARKS: A description of the column.

COLUMN_DEF: The default value of the column, in a format that can be used directly in an SQL statement.

Note that this may be an expression and not simply the text used for the default value in the original CREATE TABLE statement. For example, given:

```
  col1 char(30) default current_user    -- a 'function'
  col2 char(30) default 'string'        -- a string literal
```

where "current_user" is the name of a function, the corresponding "COLUMN_DEF" values would be:

```
  Database        col1                     col2
  --------        ----                     ----
  Oracle:         current_user             'string'
```

Postgres: "current_user"() 'string'::text

MS SQL: (user_name()) ('string')

SQL_DATA_TYPE: The SQL data type.

SQL_DATETIME_SUB: The subtype code for datetime and interval data types.

CHAR_OCTET_LENGTH: The maximum length in bytes of a character or binary data type column.

ORDINAL_POSITION: The column sequence number (starting with 1).

IS_NULLABLE: Indicates if the column can accept NULLs. Possible values are: 'NO', 'YES' and ''.

SQL/CLI defines the following additional columns:

CHAR_SET_CAT

CHAR_SET_SCHEM

CHAR_SET_NAME

COLLATION_CAT

COLLATION_SCHEM

COLLATION_NAME

UDT_CAT

UDT_SCHEM

UDT_NAME

DOMAIN_CAT

DOMAIN_SCHEM

DOMAIN_NAME

SCOPE_CAT

SCOPE_SCHEM

SCOPE_NAME

MAX_CARDINALITY

DTD_IDENTIFIER

IS_SELF_REF

Drivers capable of supplying any of those values should do so in the corresponding column and supply undef values for the others.

Drivers wishing to provide extra database/driver specific information should do so in extra columns beyond all those listed above, and use lowercase field names with the driver-specific prefix (i.e., 'ora_...'). Applications accessing such fields should do so by name and not by column number.

The result set is ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME and ORDINAL_POSITION.

Note: There is some overlap with statement handle attributes (in perl) and SQLDescribeCol (in ODBC). However, SQLColumns provides more metadata.

See also "Catalog Methods" and "Standards Reference Information".

"primary_key_info"

```
  $sth = $dbh->primary_key_info( $catalog, $schema, $table );
  # then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

Returns an active statement handle that can be used to fetch information about columns that make up the primary key for a table. The arguments don't accept search patterns (unlike table_info()).

The statement handle will return one row per column, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and KEY_SEQ. If there is no primary key then the statement handle will fetch no rows.

Note: The support for the selection criteria, such as $catalog, is driver specific. If the driver doesn't support catalogs and/or schemas, it may ignore these criteria.

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

TABLE_CAT: The catalog identifier. This field is NULL ("undef") if not applicable to the data source, which is often the case. This field is empty if not applicable to the table.

TABLE_SCHEM: The schema identifier. This field is NULL ("undef") if not applicable to the data source, and empty if not applicable to the table.

TABLE_NAME: The table identifier.

COLUMN_NAME: The column identifier.

KEY_SEQ: The column sequence number (starting with 1). Note: This field is named ORDINAL_POSITION in SQL/CLI.

PK_NAME: The primary key constraint identifier. This field is NULL ("undef") if not applicable to the data source.

See also "Catalog Methods" and "Standards Reference Information".

"primary_key"

```
  @key_column_names = $dbh->primary_key( $catalog, $schema, $table );
```

Simple interface to the primary_key_info() method. Returns a list of the column names that comprise the primary key of the specified table. The list is in primary key column sequence order. If there is no primary key then an empty list is returned.

"foreign_key_info"

```
  $sth = $dbh->foreign_key_info( $pk_catalog, $pk_schema, $pk_table
                               , $fk_catalog, $fk_schema, $fk_table );
  $sth = $dbh->foreign_key_info( $pk_catalog, $pk_schema, $pk_table
                               , $fk_catalog, $fk_schema, $fk_table
                               , \%attr );
  # then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

Returns an active statement handle that can be used to fetch information about foreign keys in and/or referencing the specified table(s). The arguments don't accept search patterns (unlike table_info()).

$pk_catalog, $pk_schema, $pk_table identify the primary (unique) key table (PKT).

$fk_catalog, $fk_schema, $fk_table identify the foreign key table (FKT).

If both PKT and FKT are given, the function returns the foreign key, if any, in table FKT that refers to the primary (unique) key of table PKT. (Note: In SQL/CLI, the result is implementation-defined.)

If only PKT is given, then the result set contains the primary key of that table and all foreign keys that refer to it.

If only FKT is given, then the result set contains all foreign keys in that table and the primary keys to which they refer. (Note: In SQL/CLI, the result includes unique keys too.)

For example:

```
  $sth = $dbh->foreign_key_info( undef, $user, 'master');
  $sth = $dbh->foreign_key_info( undef, undef,   undef , undef, $user, 'detail');
  $sth = $dbh->foreign_key_info( undef, $user, 'master', undef, $user, 'detail');
  # then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

Note: The support for the selection criteria, such as $catalog, is driver specific. If the driver doesn't support catalogs and/or schemas, it may ignore these criteria.

The statement handle returned has the following fields in the order shown below. Because ODBC never includes unique keys, they define different columns in the result set than SQL/CLI. SQL/CLI column names are shown in parentheses.

PKTABLE_CAT   (UK_TABLE_CAT   ): The primary (unique) key table catalog identifier. This field is NULL ("undef") if not applicable to the data source, which is often the case. This field is empty if not applicable to the table.

PKTABLE_SCHEM (UK_TABLE_SCHEM): The primary (unique) key table schema identifier. This field is NULL ("undef") if not applicable to the data source, and empty if not applicable to the table.

PKTABLE_NAME (UK_TABLE_NAME): The primary (unique) key table identifier.

PKCOLUMN_NAME (UK_COLUMN_NAME): The primary (unique) key column identifier.

FKTABLE_CAT (FK_TABLE_CAT): The foreign key table catalog identifier. This field is NULL ("undef") if not applicable to the data source, which is often the case. This field is empty if not applicable to the table.

FKTABLE_SCHEM (FK_TABLE_SCHEM): The foreign key table schema identifier. This field is NULL ("undef") if not applicable to the data source, and empty if not applicable to the table.

FKTABLE_NAME (FK_TABLE_NAME): The foreign key table identifier.

FKCOLUMN_NAME (FK_COLUMN_NAME): The foreign key column identifier.

KEY_SEQ (ORDINAL_POSITION): The column sequence number (starting with 1).

UPDATE_RULE (UPDATE_RULE): The referential action for the UPDATE rule. The following codes are defined:

CASCADE 0

RESTRICT 1

SET NULL 2

NO ACTION 3

SET DEFAULT 4

DELETE_RULE (DELETE_RULE): The referential action for the DELETE rule. The codes are the same as for UPDATE_RULE.

FK_NAME (FK_NAME): The foreign key name.

PK_NAME (UK_NAME): The primary (unique) key name.

DEFERRABILITY (DEFERABILITY): The deferrability of the foreign key constraint. The following codes are defined:

INITIALLY DEFERRED 5

INITIALLY IMMEDIATE 6

NOT DEFERRABLE 7

(UNIQUE_OR_PRIMARY): This column is necessary if a driver includes all candidate (i.e. primary and alternate) keys in the result set (as specified by SQL/CLI). The value of this column is UNIQUE if the foreign key references an alternate key and

PRIMARY if the foreign key references a primary key, or it may be undefined if the driver doesn't have access to the information.

See also "Catalog Methods" and "Standards Reference Information".

"statistics_info"

Warning: This method is experimental and may change.

```
  $sth = $dbh->statistics_info( $catalog, $schema, $table, $unique_only, $quick );
  # then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

Returns an active statement handle that can be used to fetch statistical information about a table and its indexes.

The arguments don't accept search patterns (unlike "table_info").

If the boolean argument $unique_only is true, only UNIQUE indexes will be returned in the result set, otherwise all indexes will be returned.

If the boolean argument $quick is set, the actual statistical information columns (CARDINALITY and PAGES) will only be returned if they are readily available from the server, and might not be current. Some databases may return stale statistics or no statistics at all with this flag set.

The statement handle will return at most one row per column name per index, plus at most one row for the entire table itself, ordered by NON_UNIQUE, TYPE, INDEX_QUALIFIER, INDEX_NAME, and ORDINAL_POSITION.

Note: The support for the selection criteria, such as $catalog, is driver specific. If the driver doesn't support catalogs and/or schemas, it may ignore these criteria.

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

TABLE_CAT: The catalog identifier. This field is NULL ("undef") if not applicable to the data source, which is often the case. This field is empty if not applicable to the table.

TABLE_SCHEM: The schema identifier. This field is NULL ("undef") if not applicable to the data source, and empty if not applicable to the table.

TABLE_NAME: The table identifier.

NON_UNIQUE: Unique index indicator. Returns 0 for unique indexes, 1 for non-unique indexes

INDEX_QUALIFIER: Index qualifier identifier. The identifier that is used to qualify the index name when doing a "DROP INDEX"; NULL ("undef") is returned if an index qualifier is not supported by the data source. If a non-NULL (defined) value is returned in this

column, it must be used to qualify the index name on a "DROP INDEX" statement; otherwise, the TABLE_SCHEM should be used to qualify the index name.

INDEX_NAME: The index identifier.

TYPE: The type of information being returned. Can be any of the following values: 'table', 'btree', 'clustered', 'content', 'hashed', or 'other'.

In the case that this field is 'table', all fields other than TABLE_CAT, TABLE_SCHEM, TABLE_NAME, TYPE, CARDINALITY, and PAGES will be NULL ("undef").

ORDINAL_POSITION: Column sequence number (starting with 1).

COLUMN_NAME: The column identifier.

ASC_OR_DESC: Column sort sequence. "A" for Ascending, "D" for Descending, or NULL ("undef") if not supported for this index.

CARDINALITY: Cardinality of the table or index. For indexes, this is the number of unique values in the index. For tables, this is the number of rows in the table. If not supported, the value will be NULL ("undef").

PAGES: Number of storage pages used by this table or index. If not supported, the value will be NULL ("undef").

FILTER_CONDITION: The index filter condition as a string. If the index is not a filtered index, or it cannot be determined whether the index is a filtered index, this value is NULL ("undef"). If the index is a filtered index, but the filter condition cannot be determined, this value is the empty string ''. Otherwise it will be the literal filter condition as a string, such as "SALARY <= 4500".

See also "Catalog Methods" and "Standards Reference Information".

"tables"

```
  @names = $dbh->tables( $catalog, $schema, $table, $type );
  @names = $dbh->tables;        # deprecated
```

Simple interface to table_info(). Returns a list of matching table names, possibly including a catalog/schema prefix.

See "table_info" for a description of the parameters.

If "$dbh->get_info(29)" returns true (29 is SQL_IDENTIFIER_QUOTE_CHAR) then the table names are constructed and quoted by "quote_identifier" to ensure they are usable even if they contain whitespace or reserved words etc. This means that the table names returned will include quote characters.

"type_info_all"

```
  $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 an 'index' hash of "Name ="> "Index" pairs. The items following that are references to arrays, one per supported data type variant. The leading index hash defines the names and order of the fields within the arrays that follow it.

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,
        SQL_DATA_TYPE     => 15,
        SQL_DATETIME_SUB  => 16,
        NUM_PREC_RADIX    => 17,
        INTERVAL_PRECISION=> 18,
    },
    [ '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
    ],
  ];
```

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 "type_info" method.

An 'index' hash is provided so you don't need to rely on index values defined above. However, using DBD::ODBC with some old ODBC drivers may return older names, shown as comments in the example above. Another issue with the index hash is that the lettercase of the keys is not defined. It is usually uppercase, as show here, but drivers may return names with any lettercase.

Drivers are also free to return extra driver-specific columns of information - though it's recommended that they start at column index 50 to leave room for expansion of the DBI/ODBC specification.

The type_info_all() method is not normally used directly. The "type_info" method provides a more usable and useful interface to the data.

"type_info"

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

Returns a list of hash references holding information about one or more variants of $data_type. 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. If called in a scalar context then only the first (best) element is returned.

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

If $data_type is an array reference then "type_info" returns the information for the first type in the array that has any matches.

The keys of the hash follow the same letter case conventions as the rest of the DBI (see "Naming Conventions and Name Space"). The following uppercase items should always exist, though may be undef:

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 characters.

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. A typical prefix is ""'"" for characters, or possibly ""0x"" for binary values passed as hexadecimal. NULL ("undef") is returned for data types for which 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)

Parameter names for data type definition. For example, "CREATE_PARAMS" for a "DECIMAL" would be ""precision,scale"" if the DECIMAL type should be declared as "DECIMAL("precision,scale")" where precision and scale are integer values. For a "VARCHAR" it would be ""max length"". NULL ("undef") is returned for data types for which this is not applicable.

NULLABLE (integer)

Indicates whether the data type accepts a NULL value: 0 or an empty string = 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, as follows:

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 for which 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 for which 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 for which this is not applicable.

LOCAL_TYPE_NAME (string)

Localized version of the "TYPE_NAME" for use in dialog with users. NULL ("undef") is returned if a localized 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 for which 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 for which this is not applicable.

SQL_DATA_TYPE (integer)

This column is the same as the "DATA_TYPE" column, except for interval and datetime data types. For interval and datetime data types, the "SQL_DATA_TYPE" field will return "SQL_INTERVAL" or "SQL_DATETIME", and the "SQL_DATETIME_SUB" field below will return the subcode for the specific interval or datetime data type. If this field is NULL, then the driver does not support or report on interval or datetime subtypes.

SQL_DATETIME_SUB (integer)

For interval or datetime data types, where the "SQL_DATA_TYPE" field above is

"SQL_INTERVAL" or "SQL_DATETIME", this field will hold the subcode for the specific interval or datetime data type. Otherwise it will be NULL ("undef").

Although not mentioned explicitly in the standards, it seems there is a simple relationship between these values:

```
DATA_TYPE == (10 * SQL_DATA_TYPE) + SQL_DATETIME_SUB
```

NUM_PREC_RADIX (integer)

The radix value of the data type. For approximate numeric types, "NUM_PREC_RADIX" contains the value 2 and "COLUMN_SIZE" holds the number of bits. For exact numeric types, "NUM_PREC_RADIX" contains the value 10 and "COLUMN_SIZE" holds the number of decimal digits. NULL ("undef") is returned either for data types for which this is not applicable or if the driver cannot report this information.

INTERVAL_PRECISION (integer)

The interval leading precision for interval types. NULL is returned either for data types for which this is not applicable or if the driver cannot report this information.

For example, to find the type name for the fields in a select statement you can do:

```
@names = map { scalar $dbh->type_info($_)->{TYPE_NAME} } @{ $sth->{TYPE} }
```

Since DBI and ODBC drivers vary in how they map their types into the ISO standard types you may need to search for more than one type. Here's an example looking for a usable type to store a date:

```
$my_date_type = $dbh->type_info( [ SQL_DATE, SQL_TIMESTAMP ] );
```

Similarly, to more reliably find a type to store small integers, you could use a list starting with "SQL_SMALLINT", "SQL_INTEGER", "SQL_DECIMAL", etc.

See also "Standards Reference Information".

"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");
```

For most database types, at least those that conform to SQL standards, quote would return

'Don''t' (including the outer quotation marks). For others it may return something like

'Don\'t'

An undefined $value value will be returned as the string "NULL" (without single 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 "type_info". As a special case, the standard numeric

types are optimized 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.

It is valid for the quote() method to return an SQL expression that evaluates to the

desired string. For example:

```
  $quoted = $dbh->quote("one\ntwo\0three")
```

may return something like:

```
  CONCAT('one', CHAR(12), 'two', CHAR(0), 'three')
```

The quote() method should not be used with "Placeholders and Bind Values".

"quote_identifier"

```
  $sql = $dbh->quote_identifier( $name );
  $sql = $dbh->quote_identifier( $catalog, $schema, $table, \%attr );
```

Quote an identifier (table name etc.) for use in an SQL statement, by escaping any special

characters (such as double quotation marks) it contains and adding the required type of

outer quotation marks.

Undefined names are ignored and the remainder are quoted and then joined together,

typically with a dot (".") character. For example:

```
  $id = $dbh->quote_identifier( undef, 'Her schema', 'My table' );
```

would, for most database types, return "Her schema"."My table" (including all the double

quotation marks).

If three names are supplied then the first is assumed to be a catalog name and special

rules may be applied based on what "get_info" returns for SQL_CATALOG_NAME_SEPARATOR (41)

and SQL_CATALOG_LOCATION (114). For example, for Oracle:

```
  $id = $dbh->quote_identifier( 'link', 'schema', 'table' );
```

would return "schema"."table"@"link".

"take_imp_data"

```
  $imp_data = $dbh->take_imp_data;
```

Leaves the $dbh in an almost dead, zombie-like, state and returns a binary string of raw implementation data from the driver which describes the current database connection. Effectively it detaches the underlying database API connection data from the DBI handle. After calling take_imp_data(), all other methods except "DESTROY" will generate a warning and return undef.

Why would you want to do this? You don't, forget I even mentioned it. Unless, that is, you're implementing something advanced like a multi-threaded connection pool. See DBI::Pool.

The returned $imp_data can be passed as a "dbi_imp_data" attribute to a later connect() call, even in a separate thread in the same process, where the driver can use it to 'adopt' the existing connection that the implementation data was taken from.

Some things to keep in mind...

* the $imp_data holds the only reference to the underlying database API connection data. That connection is still 'live' and won't be cleaned up properly unless the $imp_data is used to create a new $dbh which is then allowed to disconnect() normally.

* using the same $imp_data to create more than one other new $dbh at a time may well lead to unpleasant problems. Don't do that.

Any child statement handles are effectively destroyed when take_imp_data() is called.

The "take_imp_data" method was added in DBI 1.36 but wasn't useful till 1.49.

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 generates a warning, 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"

Type: 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 that need full transaction behaviour can set "$dbh->{AutoCommit} = 0" (or set "AutoCommit" to 0 via "connect") without having to check that the value was assigned successfully.

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

Databases which don't support transactions at all.

Databases in which a transaction is always active.

Databases in which a transaction must be explicitly started (C<'BEGIN WORK'>).

* Databases 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.

* Databases 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 "commit" is called (but see also "disconnect"). If "rollback" 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 called "commit" automatically after every successful database operation. So 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 will trigger a "commit".

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.

* Databases in which a transaction must be explicitly started

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

To do this, the driver will automatically begin an explicit transaction when "AutoCommit" is turned off, or after a "commit" or "rollback" (or when the application issues the next database operation after one of those events).

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

See "commit", "disconnect" and "Transactions" for other important notes about transactions.

"Driver"

Type: 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"

Type: 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.

"Statement"

Type: string, read-only

Returns the statement string passed to the most recent "prepare" or "do" method called in this database handle, even if that method failed. This is especially useful where "RaiseError" is enabled and the exception handler checks $@ and sees that a 'prepare' method call failed.

"RowCacheSize"

Type: integer

A hint to the driver indicating the size of the local row cache that 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, as follows:

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

Note that large cache sizes may require a very large amount of memory (cached rows *

maximum size of row). Also, a large cache will cause a longer delay not only for the first fetch, but also whenever the cache needs refilling.

See also the "RowsInCache" statement handle attribute.

"Username"

Type: string

Returns the username used to connect to the database.

DBI STATEMENT HANDLE OBJECTS

This section lists the methods and attributes associated with DBI statement handles.

Statement Handle Methods

The DBI defines the following methods for use on DBI statement handles:

"bind_param"

```
$sth->bind_param($p_num, $bind_value)
$sth->bind_param($p_num, $bind_value, \%attr)
$sth->bind_param($p_num, $bind_value, $bind_type)
```

The "bind_param" method takes a copy of $bind_value and associates it (binds it) with a placeholder, identified by $p_num, 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);
```

See "Placeholders and Bind Values" for more information.

Data Types for Placeholders

The "\%attr" parameter can be used to hint at the data type the placeholder should have. This is rarely needed. 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 the 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, such

as "{ ora_type => 97 }".

The SQL_INTEGER and other related constants can be imported using

```
use DBI qw(:sql_types);
```

See "DBI Constants" for more information.

The data type is 'sticky' in that bind values passed to execute() are bound with the data type specified by earlier bind_param() calls, if any. Portable applications should not rely on being able to change the data type after the first "bind_param" call.

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 except where the bind_param() TYPE attribute specifies a type that implies a particular format. For example, given:

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

the driver should expect $value to be in the ODBC standard SQL_DATETIME format, which is 'YYYY-MM-DD HH:MM:SS'. Similarly for SQL_DATE, SQL_TIME etc.

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"). You can then use an SQL function to convert the type within the statement. For example:

```
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 is non-portable.

See also "Placeholders and Bind Values" for more information.

"bind_param_inout"

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

This method acts like "bind_param", but also enables values to be 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 "bind_param", the $bind_value variable is not copied when "bind_param_inout" is called. Instead, the value in the variable is read at the time "execute" 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 returned from the database is too big to fit,

then the execution should fail. If unsure what value to use, pick a generous length, i.e., a length larger than the longest value that would ever be returned. The only cost of using a larger value than needed is wasted memory.

Undefined values or "undef" are used to indicate null values. See also "Placeholders and Bind Values" for more information.

"bind_param_array"

```
$rc = $sth->bind_param_array($p_num, $array_ref_or_value)
$rc = $sth->bind_param_array($p_num, $array_ref_or_value, \%attr)
$rc = $sth->bind_param_array($p_num, $array_ref_or_value, $bind_type)
```

The "bind_param_array" method is used to bind an array of values to a placeholder embedded in the prepared statement which is to be executed with "execute_array". For example:

```
$dbh->{RaiseError} = 1;        # save having to check each method call
$sth = $dbh->prepare("INSERT INTO staff (first_name, last_name, dept) VALUES(?, ?, ?)");
$sth->bind_param_array(1, [ 'John', 'Mary', 'Tim' ]);
$sth->bind_param_array(2, [ 'Booth', 'Todd', 'Robinson' ]);
$sth->bind_param_array(3, "SALES"); # scalar will be reused for each row
$sth->execute_array( { ArrayTupleStatus => \my @tuple_status } );
```

The %attr ($bind_type) argument is the same as defined for "bind_param". Refer to "bind_param" for general details on using placeholders.

(Note that bind_param_array() can not be used to expand a placeholder into a list of values for a statement like "SELECT foo WHERE bar IN (?)". A placeholder can only ever represent one value per execution.)

Scalar values, including "undef", may also be bound by "bind_param_array". In which case the same value will be used for each "execute" call. Driver-specific implementations may behave differently, e.g., when binding to a stored procedure call, some databases may permit mixing scalars and arrays as arguments.

The default implementation provided by DBI (for drivers that have not implemented array binding) is to iteratively call "execute" for each parameter tuple provided in the bound arrays. Drivers may provide more optimized implementations using whatever bulk operation support the database API provides. The default driver behaviour should match the default DBI behaviour, but always consult your driver documentation as there may be driver specific issues to consider.

Note that the default implementation currently only supports non-data returning statements

(INSERT, UPDATE, but not SELECT). Also, "bind_param_array" and "bind_param" cannot be mixed in the same statement execution, and "bind_param_array" must be used with "execute_array"; using "bind_param_array" will have no effect for "execute".

The "bind_param_array" method was added in DBI 1.22.

"execute"

```
  $rv = $sth->execute                or die $sth->errstr;
  $rv = $sth->execute(@bind_values)  or 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 if you're not using "RaiseError".

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 database 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 databases can't tell in advance), it simply returns a true value.

You can tell if the statement was a "SELECT" statement by checking if "$sth->{NUM_OF_FIELDS}" is greater than zero after calling "execute".

If any arguments are given, then "execute" will effectively call "bind_param" 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 unless "bind_param" (or "bind_param_inout") has already been used to specify the type.

Note that passing "execute" an empty array is the same as passing no arguments at all, which will execute the statement with previously bound values. That's probably not what you want.

If execute() is called on a statement handle that's still active ($sth->{Active} is true) then it should effectively call finish() to tidy up the previous execution results before starting this new execution.

"execute_array"

```
$tuples = $sth->execute_array(\%attr) or die $sth->errstr;
$tuples = $sth->execute_array(\%attr, @bind_values) or die $sth->errstr;
($tuples, $rows) = $sth->execute_array(\%attr) or die $sth->errstr;
($tuples, $rows) = $sth->execute_array(\%attr, @bind_values) or die $sth->errstr;
```

Execute the prepared statement once for each parameter tuple (group of values) provided either in the @bind_values, or by prior calls to "bind_param_array", or via a reference passed in \%attr.

When called in scalar context the execute_array() method returns the number of tuples executed, or "undef" if an error occurred. Like execute(), a successful execute_array() always returns true regardless of the number of tuples executed, even if it's zero. If there were any errors the ArrayTupleStatus array can be used to discover which tuples failed and with what errors.

When called in list context the execute_array() method returns two scalars; $tuples is the same as calling execute_array() in scalar context and $rows is the number of rows affected for each tuple, if available or -1 if the driver cannot determine this. NOTE, some drivers cannot determine the number of rows affected per tuple but can provide the number of rows affected for the batch. If you are doing an update operation the returned rows affected may not be what you expect if, for instance, one or more of the tuples affected the same row multiple times. Some drivers may not yet support list context, in which case $rows will be undef, or may not be able to provide the number of rows affected when performing this batch operation, in which case $rows will be -1.

Bind values for the tuples to be executed may be supplied row-wise by an "ArrayTupleFetch" attribute, or else column-wise in the @bind_values argument, or else column-wise by prior calls to "bind_param_array".

Where column-wise binding is used (via the @bind_values argument or calls to bind_param_array()) the maximum number of elements in any one of the bound value arrays determines the number of tuples executed. Placeholders with fewer values in their parameter arrays are treated as if padded with undef (NULL) values.

If a scalar value is bound, instead of an array reference, it is treated as a variable length array with all elements having the same value. It does not influence the number of tuples executed, so if all bound arrays have zero elements then zero tuples will be executed. If all bound values are scalars then one tuple will be executed, making execute_array() act just like execute().

The "ArrayTupleFetch" attribute can be used to specify a reference to a subroutine that will be called to provide the bind values for each tuple execution. The subroutine should return an reference to an array which contains the appropriate number of bind values, or return an undef if there is no more data to execute.

As a convenience, the "ArrayTupleFetch" attribute can also be used to specify a statement handle. In which case the fetchrow_arrayref() method will be called on the given statement handle in order to provide the bind values for each tuple execution.

The values specified via bind_param_array() or the @bind_values parameter may be either scalars, or arrayrefs. If any @bind_values are given, then "execute_array" will effectively call "bind_param_array" 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 unless "bind_param", "bind_param_inout", "bind_param_array", or "bind_param_inout_array" has already been used to specify the type. See "bind_param_array" for details.

The "ArrayTupleStatus" attribute can be used to specify a reference to an array which will receive the execute status of each executed parameter tuple. Note the "ArrayTupleStatus" attribute was mandatory until DBI 1.38.

For tuples which are successfully executed, the element at the same ordinal position in the status array is the resulting rowcount (or -1 if unknown). If the execution of a tuple causes an error, then the corresponding status array element will be set to a reference to an array containing "err", "errstr" and "state" set by the failed execution.

If any tuple execution returns an error, "execute_array" will return "undef". In that case, the application should inspect the status array to determine which parameter tuples failed. Some databases may not continue executing tuples beyond the first failure. In this case the status array will either hold fewer elements, or the elements beyond the failure will be undef.

If all parameter tuples are successfully executed, "execute_array" returns the number tuples executed. If no tuples were executed, then execute_array() returns "0E0", just like execute() does, which Perl will treat as 0 but will regard as true.

For example:

```
  $sth = $dbh->prepare("INSERT INTO staff (first_name, last_name) VALUES (?, ?)");
  my $tuples = $sth->execute_array(
      { ArrayTupleStatus => \my @tuple_status },
```

```
    \@first_names,
    \@last_names,
  );
  if ($tuples) {
      print "Successfully inserted $tuples records\n";
  }
  else {
      for my $tuple (0..@last_names-1) {
          my $status = $tuple_status[$tuple];
          $status = [0, "Skipped"] unless defined $status;
          next unless ref $status;
          printf "Failed to insert (%s, %s): %s\n",
              $first_names[$tuple], $last_names[$tuple], $status->[1];
      }
  }
```

Support for data returning statements such as SELECT is driver-specific and subject to change. At present, the default implementation provided by DBI only supports non-data returning statements.

Transaction semantics when using array binding are driver and database specific. If "AutoCommit" is on, the default DBI implementation will cause each parameter tuple to be individually committed (or rolled back in the event of an error). If "AutoCommit" is off, the application is responsible for explicitly committing the entire set of bound parameter tuples. Note that different drivers and databases may have different behaviours when some parameter tuples cause failures. In some cases, the driver or database may automatically rollback the effect of all prior parameter tuples that succeeded in the transaction; other drivers or databases may retain the effect of prior successfully executed parameter tuples. Be sure to check your driver and database for its specific behaviour.

Note that, in general, performance will usually be better with "AutoCommit" turned off, and using explicit "commit" after each "execute_array" call.

The "execute_array" method was added in DBI 1.22, and ArrayTupleFetch was added in 1.36.

"execute_for_fetch"

```
  $tuples = $sth->execute_for_fetch($fetch_tuple_sub);
  $tuples = $sth->execute_for_fetch($fetch_tuple_sub, \@tuple_status);
```

```
($tuples, $rows) = $sth->execute_for_fetch($fetch_tuple_sub);
($tuples, $rows) = $sth->execute_for_fetch($fetch_tuple_sub, \@tuple_status);
```

The execute_for_fetch() method is used to perform bulk operations and although it is most often used via the execute_array() method you can use it directly. The main difference between execute_array and execute_for_fetch is the former does column or row-wise binding and the latter uses row-wise binding.

The fetch subroutine, referenced by $fetch_tuple_sub, is expected to return a reference to an array (known as a 'tuple') or undef.

The execute_for_fetch() method calls $fetch_tuple_sub, without any parameters, until it returns a false value. Each tuple returned is used to provide bind values for an $sth->execute(@$tuple) call.

In scalar context execute_for_fetch() returns "undef" if there were any errors and the number of tuples executed otherwise. Like execute() and execute_array() a zero is returned as "0E0" so execute_for_fetch() is only false on error. If there were any errors the @tuple_status array can be used to discover which tuples failed and with what errors.

When called in list context execute_for_fetch() returns two scalars; $tuples is the same as calling execute_for_fetch() in scalar context and $rows is the sum of the number of rows affected for each tuple, if available or -1 if the driver cannot determine this. If you are doing an update operation the returned rows affected may not be what you expect if, for instance, one or more of the tuples affected the same row multiple times. Some drivers may not yet support list context, in which case $rows will be undef, or may not be able to provide the number of rows affected when performing this batch operation, in which case $rows will be -1.

If \@tuple_status is passed then the execute_for_fetch method uses it to return status information. The tuple_status array holds one element per tuple. If the corresponding execute() did not fail then the element holds the return value from execute(), which is typically a row count. If the execute() did fail then the element holds a reference to an array containing ($sth->err, $sth->errstr, $sth->state).

If the driver detects an error that it knows means no further tuples can be executed then it may return, with an error status, even though $fetch_tuple_sub may still have more tuples to be executed.

Although each tuple returned by $fetch_tuple_sub is effectively used to call $sth->execute(@$tuple_array_ref) the exact timing may vary. Drivers are free to

accumulate sets of tuples to pass to the database server in bulk group operations for more efficient execution. However, the $fetch_tuple_sub is specifically allowed to return the same array reference each time (which is what fetchrow_arrayref() usually does).

For example:

```
my $sel = $dbh1->prepare("select foo, bar from table1");
$sel->execute;
my $ins = $dbh2->prepare("insert into table2 (foo, bar) values (?,?)");
my $fetch_tuple_sub = sub { $sel->fetchrow_arrayref };
my @tuple_status;
$rc = $ins->execute_for_fetch($fetch_tuple_sub, \@tuple_status);
my @errors = grep { ref $_ } @tuple_status;
```

Similarly, if you already have an array containing the data rows to be processed you'd use a subroutine to shift off and return each array ref in turn:

```
$ins->execute_for_fetch( sub { shift @array_of_arrays }, \@tuple_status);
```

The "execute_for_fetch" method was added in DBI 1.38.

"last_insert_id"

```
$rv = $sth->last_insert_id();
$rv = $sth->last_insert_id($catalog, $schema, $table, $field);
$rv = $sth->last_insert_id($catalog, $schema, $table, $field, \%attr);
```

Returns a value 'identifying' the row inserted by last execution of the statement $sth, if possible.

For some drivers the value may be 'identifying' the row inserted by the last executed statement, not by $sth.

See database handle method last_insert_id for all details.

The "last_insert_id" statement method was added in DBI 1.642.

"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 fields are returned as "undef" values in the array. This is the fastest way to fetch data, particularly if used with "$sth->bind_columns".

If there are no more rows or if 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 the same array reference is returned for each fetch, so don't store the reference and then use it after a later fetch. Also, the elements of the array are also reused for each row, so take care if you want to take a reference to an element. See also "bind_columns".

"fetchrow_array"

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

An alternative to "fetchrow_arrayref". Fetches the next row of data and returns it as a list containing the field values. Null fields are returned as "undef" values in the list.

If there are no more rows or if 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.

If called in a scalar context for a statement handle that has more than one column, it is undefined whether the driver will return the value of the first column or the last. So don't do that. Also, in a scalar context, an "undef" is returned if there are no more rows or if an error occurred. That "undef" can't be distinguished from an "undef" returned because the first field value was NULL. For these reasons you should exercise some caution if you use "fetchrow_array" in a scalar context.

"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 fields are returned as "undef" values in the hash.

If there are no more rows or if 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. For historical reasons it defaults to ""NAME"", however using either ""NAME_lc"" or ""NAME_uc"" is recommended for portability.

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, so statements like ""select foo, foo from bar"" will return only a single key from

"fetchrow_hashref". In these cases use column aliases or "fetchrow_arrayref". Note that it is the database server (and not the DBD implementation) which provides the name for fields containing functions like "count(*)" or ""max(c_foo)"" and they may clash with existing column names (most databases don't care about duplicate column names in a result-set). If you want these to return as unique names that are the same across databases, use aliases, as in ""select count(*) as cnt"" or ""select max(c_foo) mx_foo, ..."" depending on the syntax your database supports.

Because of the extra work "fetchrow_hashref" and Perl have to perform, it is not as efficient as "fetchrow_arrayref" or "fetchrow_array".

By default a reference to a new hash is returned for each row. It is likely that a future version of the DBI will support an attribute which will enable the same hash to be reused for each row. This will give a significant performance boost, but it won't be enabled by default because of the risk of breaking old code.

"fetchall_arrayref"

```
  $tbl_ary_ref = $sth->fetchall_arrayref;
  $tbl_ary_ref = $sth->fetchall_arrayref( $slice );
  $tbl_ary_ref = $sth->fetchall_arrayref( $slice, $max_rows  );
```

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 that contains one reference per row.

If called on an inactive statement handle, "fetchall_arrayref" returns undef.

If there are no rows left to return from an active statement handle, "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.

If $slice is an array reference, "fetchall_arrayref" uses "fetchrow_arrayref" to fetch each row as an array ref. If the $slice array is not empty then it is used as a slice to select individual columns by perl array index number (starting at 0, unlike column and parameter numbers which start at 1).

With no parameters, or if $slice is undefined, "fetchall_arrayref" acts as if passed an empty array ref.

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

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

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

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

Those two examples both return a reference to an array of array refs.

If $slice is a hash reference, "fetchall_arrayref" fetches each row as a hash reference.

If the $slice hash is empty then the keys in the hashes have whatever name lettercase is returned by default. (See "FetchHashKeyName" attribute.) If the $slice hash is not empty, then it is used as a slice to select individual columns by name. The values of the hash should be set to 1. The key names of the returned hashes match the letter case of the names in the parameter hash, regardless of the "FetchHashKeyName" attribute.

For example, to fetch all fields of every row as a hash ref:

```
$tbl_ary_ref = $sth->fetchall_arrayref({});
```

To fetch only the fields called "foo" and "bar" of every row as a hash ref (with keys named "foo" and "BAR", regardless of the original capitalization):

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

Those two examples both return a reference to an array of hash refs.

If $slice is a reference to a hash reference, that hash is used to select and rename columns. The keys are 0-based column index numbers and the values are the corresponding keys for the returned row hashes.

For example, to fetch only the first and second columns of every row as a hash ref (with keys named "k" and "v" regardless of their original names):

```
$tbl_ary_ref = $sth->fetchall_arrayref( \{ 0 => 'k', 1 => 'v' } );
```

If $max_rows is defined and greater than or equal to zero then it is used to limit the number of rows fetched before returning. fetchall_arrayref() can then be called again to fetch more rows. This is especially useful when you need the better performance of fetchall_arrayref() but don't have enough memory to fetch and return all the rows in one go.

Here's an example (assumes RaiseError is enabled):

```
my $rows = []; # cache for batches of rows
while( my $row = ( shift(@$rows) || # get row from cache, or reload cache:
                   shift(@{$rows=$sth->fetchall_arrayref(undef,10_000)||[]}) )
) {
  ...
```

```
}
```

That might be the fastest way to fetch and process lots of rows using the DBI, but it depends on the relative cost of method calls vs memory allocation.

A standard "while" loop with column binding is often faster because the cost of allocating memory for the batch of rows is greater than the saving by reducing method calls. It's possible that the DBI may provide a way to reuse the memory of a previous batch in future, which would then shift the balance back towards fetchall_arrayref().

"fetchall_hashref"

```
$hash_ref = $sth->fetchall_hashref($key_field);
```

The "fetchall_hashref" method can be used to fetch all the data to be returned from a prepared and executed statement handle. It returns a reference to a hash containing a key for each distinct value of the $key_field column that was fetched. For each key the corresponding value is a reference to a hash containing all the selected columns and their values, as returned by "fetchrow_hashref()".

If there are no rows to return, "fetchall_hashref" returns a reference to an empty hash.

If an error occurs, "fetchall_hashref" 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.

The $key_field parameter provides the name of the field that holds the value to be used for the key for the returned hash. For example:

```
$dbh->{FetchHashKeyName} = 'NAME_lc';
$sth = $dbh->prepare("SELECT FOO, BAR, ID, NAME, BAZ FROM TABLE");
$sth->execute;
$hash_ref = $sth->fetchall_hashref('id');
print "Name for id 42 is $hash_ref->{42}->{name}\n";
```

The $key_field parameter can also be specified as an integer column number (counting from 1). If $key_field doesn't match any column in the statement, as a name first then as a number, then an error is returned.

For queries returning more than one 'key' column, you can specify multiple column names by passing $key_field as a reference to an array containing one or more key column names (or index numbers). For example:

```
$sth = $dbh->prepare("SELECT foo, bar, baz FROM table");
$sth->execute;
```

```
  $hash_ref = $sth->fetchall_hashref( [ qw(foo bar) ] );
  print "For foo 42 and bar 38, baz is $hash_ref->{42}->{38}->{baz}\n";
```

The fetchall_hashref() method is normally used only where the key fields values for each row are unique. If multiple rows are returned with the same values for the key fields then later rows overwrite earlier ones.

"finish"

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

Indicate that no more data will be fetched from this statement handle before it is either executed again or destroyed. You almost certainly do not need to call this method.

Adding calls to "finish" after loop that fetches all rows is a common mistake, don't do it, it can mask genuine problems like uncaught fetch errors.

When all the data has been fetched from a "SELECT" statement, the driver will automatically call "finish" for you. So you should not call it explicitly except when you know that you've not fetched all the data from a statement handle and the handle won't be destroyed soon.

The most common example is when you only want to fetch just one row, but in that case the "selectrow_*" methods are usually better anyway.

Consider a query like:

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

on 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 just a few rows, 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 "Active" attribute for the statement. It may also make some statement handle attributes (such as "NAME" and "TYPE") unavailable if they have not already been accessed (and thus cached).

The "finish" method does not affect the transaction status of the database connection. It has nothing to do with transactions. It's mostly an internal "housekeeping" method that is rarely needed. See also "disconnect" and the "Active" attribute.

The "finish" method should have been called "discard_pending_rows".

"rows"

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

Returns the number of rows affected by the last row affecting command, or -1 if the number

of rows is 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.

One alternative method to get a row count for a "SELECT" is to execute a "SELECT COUNT(*) FROM ..." SQL statement with the same "..." as your query and then fetch the row count from that.

"bind_col"

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

Binds a Perl variable and/or some attributes to an output column (field) of a "SELECT" statement. Column numbers count up from 1. You do not need to bind output columns in order to fetch data. For maximum portability between drivers, bind_col() should be called after execute() and not before. See also "bind_columns" for an example.

The binding is performed at a low level using Perl aliasing. Whenever a row is fetched from the database $var_to_bind appears to be automatically updated simply because it now refers to the same memory location as the corresponding column value. This makes using bound variables very efficient. Binding a tied variable doesn't work, currently.

The "bind_param" method performs a similar, but opposite, function for input variables.

Data Types for Column Binding

The "\%attr" parameter can be used to hint at the data type formatting the column should have. For example, you can use:

```
  $sth->bind_col(1, undef, { TYPE => SQL_DATETIME });
```

to specify that you'd like the column (which presumably is some kind of datetime type) to be returned in the standard format for SQL_DATETIME, which is 'YYYY-MM-DD HH:MM:SS', rather than the native formatting the database would normally use.

There's no $var_to_bind in that example to emphasize the point that bind_col() works on the underlying column and not just a particular bound variable.

As a short-cut for the 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_col(1, undef, SQL_DATETIME);
```

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, such as "{ ora_type => 97 }".

The SQL_DATETIME and other related constants can be imported using

```
use DBI qw(:sql_types);
```

See "DBI Constants" for more information.

Few drivers support specifying a data type via a "bind_col" call (most will simply ignore the data type). Fewer still allow the data type to be altered once set. If you do set a column type the type should remain sticky through further calls to bind_col for the same column if the type is not overridden (this is important for instance when you are using a slice in fetchall_arrayref).

The TYPE attribute for bind_col() was first specified in DBI 1.41.

From DBI 1.611, drivers can use the "TYPE" attribute to attempt to cast the bound scalar to a perl type which more closely matches "TYPE". At present DBI supports "SQL_INTEGER", "SQL_DOUBLE" and "SQL_NUMERIC". See "sql_type_cast" for details of how types are cast.

Other attributes for Column Binding

The "\%attr" parameter may also contain the following attributes:

"StrictlyTyped"

If a "TYPE" attribute is passed to bind_col, then the driver will attempt to change the bound perl scalar to match the type more closely. If the bound value cannot be cast to the requested "TYPE" then by default it is left untouched and no error is generated. If you specify "StrictlyTyped" as 1 and the cast fails, this will generate an error.

This attribute was first added in DBI 1.611. When 1.611 was released few drivers actually supported this attribute but DBD::Oracle and DBD::ODBC should from versions 1.24.

"DiscardString"

When the "TYPE" attribute is passed to "bind_col" and the driver successfully casts the bound perl scalar to a non-string type then if "DiscardString" is set to 1, the string portion of the scalar will be discarded. By default, "DiscardString" is not

set.

This attribute was first added in DBI 1.611. When 1.611 was released few drivers actually supported this attribute but DBD::Oracle and DBD::ODBC should from versions 1.24.

"bind_columns"

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

Calls "bind_col" for each column of the "SELECT" statement.

The list of references should have the same number of elements as the number of columns in the "SELECT" statement. If it doesn't then "bind_columns" will bind the elements given, up to the number of columns, and then return an error.

For maximum portability between drivers, bind_columns() should be called after execute() and not before.

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, the first parameter will be ignored if it is "undef" or a hash reference.

Here's a more fancy example that binds columns to the values inside a hash (thanks to H.Merijn Brand):

```
$sth->execute;
my %row;
$sth->bind_columns( \( @row{ @{$sth->{NAME_lc} } } ));
while ($sth->fetch) {
```

```
    print "$row{region}: $row{sales}\n";
  }
```

"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 "neat_list" to format and edit the string for reading by humans, it is 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 generates a warning, except for private driver specific attributes (which all have names starting with a lowercase letter).

Example:

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

Some drivers cannot provide valid values for some or all of these attributes until after "$sth->execute" has been successfully called. Typically the attribute will be "undef" in these situations.

Some attributes, like NAME, are not appropriate to some types of statement, like SELECT. Typically the attribute will be "undef" in these situations.

For drivers which support stored procedures and multiple result sets (see "more_results") these attributes relate to the current result set.

See also "finish" to learn more about the effect it may have on some attributes.

"NUM_OF_FIELDS"

Type: integer, read-only

Number of fields (columns) in the data the prepared statement may return. Statements that don't return rows of data, like "DELETE" and "CREATE" set "NUM_OF_FIELDS" to 0 (though it may be undef in some drivers).

"NUM_OF_PARAMS"

Type: integer, read-only

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

"NAME"

Type: 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 "NAME_lc" or "NAME_uc".

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

Also note that the name returned for (aggregate) functions like count(*) or "max(c_foo)" is determined by the database server and not by "DBI" or the "DBD" backend.

"NAME_lc"

Type: array-ref, read-only

Like "/NAME" but always returns lowercase names.

"NAME_uc"

Type: array-ref, read-only

Like "/NAME" but always returns uppercase names.

"NAME_hash"

Type: hash-ref, read-only

"NAME_lc_hash"

Type: hash-ref, read-only

"NAME_uc_hash"

Type: hash-ref, read-only

The "NAME_hash", "NAME_lc_hash", and "NAME_uc_hash" attributes return column name information as a reference to a hash.

The keys of the hash are the names of the columns. The letter case of the keys corresponds to the letter case returned by the "NAME", "NAME_lc", and "NAME_uc" attributes respectively (as described above).

The value of each hash entry is the perl index number of the corresponding column (counting from 0). For example:

```
  $sth = $dbh->prepare("select Id, Name from table");
```

```
$sth->execute;
@row = $sth->fetchrow_array;
print "Name $row[ $sth->{NAME_lc_hash}{name} ]\n";
```

"TYPE"

Type: array-ref, read-only

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

The values correspond to the international standards (ANSI X3.135 and ISO/IEC 9075) which, in general terms, means ODBC. Driver-specific types that 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 with the ISO working group:

```
ftp://sqlstandards.org/SC32/SQL_Registry/
```

Where there's no vendor-supplied ODBC driver to be compatible with, the DBI driver can use type numbers in the range that is 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 "type_info_all").

"PRECISION"

Type: array-ref, read-only

Returns a reference to an array of integer values for each column.

For numeric columns, the value is the maximum number of digits (without considering a sign character or decimal point). Note that the "display size" for floating point types (REAL, FLOAT, DOUBLE) can be up to 7 characters greater than the precision (for the sign + decimal point + the letter E + a sign + 2 or 3 digits).

For any character type column the value is the OCTET_LENGTH, in other words the number of bytes, not characters.

(More recent standards refer to this as COLUMN_SIZE but we stick with PRECISION for backwards compatibility.)

"SCALE"

Type: array-ref, read-only

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

"NULLABLE"

Type: array-ref, read-only

Returns a reference to an array indicating the possibility of each column returning a

null. Possible values are 0 (or an empty string) = no, 1 = yes, 2 = unknown.

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

"CursorName"

Type: string, read-only

Returns the name of the cursor associated with the statement handle, if available. If not

available or if the database driver does not support the "where current of ..." SQL

syntax, then it returns "undef".

"Database"

Type: dbh, read-only

Returns the parent $dbh of the statement handle.

"Statement"

Type: string, read-only

Returns the statement string passed to the "prepare" method.

"ParamValues"

Type: hash ref, read-only

Returns a reference to a hash containing the values currently bound to placeholders. The

keys of the hash are the 'names' of the placeholders, typically integers starting at 1.

Returns undef if not supported by the driver.

See "ShowErrorStatement" for an example of how this is used.

* Keys:

If the driver supports "ParamValues" but no values have been bound yet then the driver

should return a hash with placeholders names in the keys but all the values undef, but

some drivers may return a ref to an empty hash because they can't pre-determine the names.

It is possible that the keys in the hash returned by "ParamValues" are not exactly the

same as those implied by the prepared statement. For example, DBD::Oracle translates

'"?"' placeholders into '":pN"' where N is a sequence number starting at 1.

* Values:

It is possible that the values in the hash returned by "ParamValues" are not exactly the

same as those passed to bind_param() or execute(). The driver may have slightly modified

values in some way based on the TYPE the value was bound with. For example a floating

point value bound as an SQL_INTEGER type may be returned as an integer. The values returned by "ParamValues" can be passed to another bind_param() method with the same TYPE and will be seen by the database as the same value. See also "ParamTypes" below.

The "ParamValues" attribute was added in DBI 1.28.

"ParamTypes"

Type: hash ref, read-only

Returns a reference to a hash containing the type information currently bound to placeholders. Returns undef if not supported by the driver.

* Keys:

See "ParamValues" above.

* Values:

The hash values are hashrefs of type information in the same form as that passed to the various bind_param() methods (See "bind_param" for the format and values).

It is possible that the values in the hash returned by "ParamTypes" are not exactly the same as those passed to bind_param() or execute(). Param attributes specified using the abbreviated form, like this:

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

are returned in the expanded form, as if called like this:

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

The driver may have modified the type information in some way based on the bound values, other hints provided by the prepare()'d SQL statement, or alternate type mappings required by the driver or target database system. The driver may also add private keys (with names beginning with the drivers reserved prefix, e.g., odbc_xxx).

* Example:

The keys and values in the returned hash can be passed to the various bind_param() methods to effectively reproduce a previous param binding. For example:

```
 # assuming $sth1 is a previously prepared statement handle
 my $sth2 = $dbh->prepare( $sth1->{Statement} );
 my $ParamValues = $sth1->{ParamValues} || {};
 my $ParamTypes  = $sth1->{ParamTypes}  || {};
 $sth2->bind_param($_, $ParamValues->{$_}, $ParamTypes->{$_})
   for keys %{ {%$ParamValues, %$ParamTypes} };
 $sth2->execute();
```

The "ParamTypes" attribute was added in DBI 1.49. Implementation is the responsibility of individual drivers; the DBI layer default implementation simply returns undef.

"ParamArrays"

Type: hash ref, read-only

Returns a reference to a hash containing the values currently bound to placeholders with "execute_array" or "bind_param_array". The keys of the hash are the 'names' of the placeholders, typically integers starting at 1. Returns undef if not supported by the driver or no arrays of parameters are bound.

Each key value is an array reference containing a list of the bound parameters for that column.

For example:

```
 $sth = $dbh->prepare("INSERT INTO staff (id, name) values (?,?)");
 $sth->execute_array({},[1,2], ['fred','dave']);
 if ($sth->{ParamArrays}) {
     foreach $param (keys %{$sth->{ParamArrays}}) {
         printf "Parameters for %s : %s\n", $param,
         join(",", @{$sth->{ParamArrays}->{$param}});
     }
 }
```

It is possible that the values in the hash returned by "ParamArrays" are not exactly the same as those passed to "bind_param_array" or "execute_array". The driver may have slightly modified values in some way based on the TYPE the value was bound with. For example a floating point value bound as an SQL_INTEGER type may be returned as an integer.

It is also possible that the keys in the hash returned by "ParamArrays" are not exactly the same as those implied by the prepared statement. For example, DBD::Oracle translates '"?"' placeholders into '":pN"' where N is a sequence number starting at 1.

"RowsInCache"

Type: integer, read-only

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, whereas others wait till the first fetch.

See also the "RowCacheSize" database handle attribute.

FURTHER INFORMATION

Catalog Methods

An application can retrieve metadata information from the DBMS by issuing appropriate queries on the views of the Information Schema. Unfortunately, "INFORMATION_SCHEMA" views are seldom supported by the DBMS. Special methods (catalog methods) are available to return result sets for a small but important portion of that metadata:

```
column_info
foreign_key_info
primary_key_info
table_info
statistics_info
```

All catalog methods accept arguments in order to restrict the result sets. Passing "undef" to an optional argument does not constrain the search for that argument. However, an empty string ('') is treated as a regular search criteria and will only match an empty value.

Note: SQL/CLI and ODBC differ in the handling of empty strings. An empty string will not restrict the result set in SQL/CLI.

Most arguments in the catalog methods accept only ordinary values, e.g. the arguments of "primary_key_info()". Such arguments are treated as a literal string, i.e. the case is significant and quote characters are taken literally.

Some arguments in the catalog methods accept search patterns (strings containing '_' and/or '%'), e.g. the $table argument of "column_info()". Passing '%' is equivalent to leaving the argument "undef".

Caveat: The underscore ('_') is valid and often used in SQL identifiers. Passing such a value to a search pattern argument may return more rows than expected! To include pattern characters as literals, they must be preceded by an escape character which can be achieved with

```
$esc = $dbh->get_info( 14 );  # SQL_SEARCH_PATTERN_ESCAPE
$search_pattern =~ s/([_%])/$esc$1/g;
```

The ODBC and SQL/CLI specifications define a way to change the default behaviour described above: All arguments (except list value arguments) are treated as identifier if the "SQL_ATTR_METADATA_ID" attribute is set to "SQL_TRUE". Quoted identifiers are very similar to ordinary values, i.e. their body (the string within the quotes) is interpreted

literally. Unquoted identifiers are compared in UPPERCASE.

The DBI (currently) does not support the "SQL_ATTR_METADATA_ID" attribute, i.e. it behaves like an ODBC driver where "SQL_ATTR_METADATA_ID" is set to "SQL_FALSE".

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 that support transactions and where "AutoCommit" is off. See "AutoCommit" for details of using "AutoCommit" with various types of databases.

The recommended way to implement robust transactions in Perl applications is to enable "RaiseError" and catch the error that's 'thrown' as an exception. For example, using Try::Tiny:

```
use Try::Tiny;
$dbh->{AutoCommit} = 0;  # enable transactions, if possible
$dbh->{RaiseError} = 1;
try {
    foo(...)        # do lots of work here
    bar(...)        # including inserts
    baz(...)        # and updates
    $dbh->commit;   # commit the changes if we get this far
} catch {
    warn "Transaction aborted because $_"; # Try::Tiny copies $@ into $_
    # now rollback to undo the incomplete changes
    # but do it in an eval{} as it may also fail
    eval { $dbh->rollback };
    # add other application on-error-clean-up code here
};
```

If the "RaiseError" attribute is not set, then DBI calls would need to be manually checked for errors, typically like this:

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

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 "RaiseError" for more details.

A major advantage of the "eval" approach is that the transaction will be properly rolled back if any code (not just DBI calls) 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.

After calling "commit" or "rollback" many drivers will not let you fetch from a previously active "SELECT" statement handle that's a child of the same database handle. A typical way round this is to connect the the database twice and use one connection for "SELECT" statements.

See "AutoCommit" and "disconnect" for other important information about transactions.

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.

See the description of "LongReadLen" for more information.

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 "quote" method generally can't cope with binary data. See "Placeholders and Bind Values".

Simple Examples

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

```
my $data_source = "dbi::DriverName:db_name";
my $dbh = DBI->connect($data_source, $user, $password)
    or die "Can't connect to $data_source: $DBI::errstr";
my $sth = $dbh->prepare( q{
        SELECT name, phone
        FROM mytelbook
}) or die "Can't prepare statement: $DBI::errstr";
my $rc = $sth->execute
```

```
    or 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: $!";
  while (<FH>) {
    chomp;
    my ($name, $phone) = split /,/;
    $sth->execute($name, $phone);
  }
  close FH;
  $dbh->commit;
  $dbh->disconnect;
```

Here's how to convert fetched NULLs (undefined values) into 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 examples 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 "Quote and Quote-like Operators" in perlop for more details.

Threads and Thread Safety

Perl 5.7 and later support a new threading model called iThreads. (The old "5.005 style" threads are not supported by the DBI.)

In the iThreads model each thread has its own copy of the perl interpreter. When a new thread is created the original perl interpreter is 'cloned' to create a new copy for the new thread.

If the DBI and drivers are loaded and handles created before the thread is created then it will get a cloned copy of the DBI, the drivers and the handles.

However, the internal pointer data within the handles will refer to the DBI and drivers in the original interpreter. Using those handles in the new interpreter thread is not safe, so the DBI detects this and croaks on any method call using handles that don't belong to the current thread (except for DESTROY).

Because of this (possibly temporary) restriction, newly created threads must make their own connections to the database. Handles can't be shared across threads.

But BEWARE, some underlying database APIs (the code the DBD driver uses to talk to the database, often supplied by the database vendor) are not thread safe. If it's not thread safe, then allowing more than one thread to enter the code at the same time may cause subtle/serious problems. In some cases allowing more than one thread to enter the code, even if not at the same time, can cause problems. You have been warned.

Using DBI with perl threads is not yet recommended for production environments. For more information see <http://www.perlmonks.org/index.pl?node_id=288022>

Note: There is a bug in perl 5.8.2 when configured with threads and debugging enabled (bug #24463) which causes a DBI test to fail.

Signal Handling and Canceling Operations

[The following only applies to systems with unix-like signal handling. I'd welcome additions for other systems, especially Windows.]

The first thing to say is that signal handling in Perl versions less than 5.8 is not safe. There is always a small risk of Perl crashing and/or core dumping when, or after, handling a signal because the signal could arrive and be handled while internal data structures are being changed. If the signal handling code used those same internal data structures it

could cause all manner of subtle and not-so-subtle problems. The risk was reduced with 5.4.4 but was still present in all perls up through 5.8.0.

Beginning in perl 5.8.0 perl implements 'safe' signal handling if your system has the POSIX sigaction() routine. Now when a signal is delivered perl just makes a note of it but does not run the %SIG handler. The handling is 'deferred' until a 'safe' moment.

Although this change made signal handling safe, it also lead to a problem with signals being deferred for longer than you'd like. If a signal arrived while executing a system call, such as waiting for data on a network connection, the signal is noted and then the system call that was executing returns with an EINTR error code to indicate that it was interrupted. All fine so far.

The problem comes when the code that made the system call sees the EINTR code and decides it's going to call it again. Perl doesn't do that, but database code sometimes does. If that happens then the signal handler doesn't get called until later. Maybe much later.

Fortunately there are ways around this which we'll discuss below. Unfortunately they make signals unsafe again.

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}.

Cancel

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. For example:

```
  $SIG{INT} = sub { $sth->cancel };
```

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

If "cancel" returns true, then it has successfully invoked the database engine's own cancel function. If it returns false, then "cancel" failed. If it returns "undef", then the database driver does not have cancel implemented - very few do.

Timeout

The traditional way to implement a timeout is to set $SIG{ALRM} to refer to some code that will be executed when an ALRM signal arrives and then to call alarm($seconds) to schedule an ALRM signal to be delivered $seconds in the future. For example:

```
  my $failed;
  eval {
    local $SIG{ALRM} = sub { die "TIMEOUT\n" }; # N.B. \n required
    eval {
      alarm($seconds);
      ... code to execute with timeout here (which may die) ...
      1;
    } or $failed = 1;
    # outer eval catches alarm that might fire JUST before this alarm(0)
    alarm(0);  # cancel alarm (if code ran fast)
    die "$@" if $failed;
    1;
  } or $failed = 1;
  if ( $failed ) {
    if ( defined $@ and $@ eq "TIMEOUT\n" ) { ... }
    else { ... } # some other error
  }
```

The first (outer) eval is used to avoid the unlikely but possible chance that the "code to execute" dies and the alarm fires before it is cancelled. Without the outer eval, if this happened your program will die if you have no ALRM handler or a non-local alarm handler will be called.

Unfortunately, as described above, this won't always work as expected, depending on your perl version and the underlying database code.

With Oracle for instance (DBD::Oracle), if the system which hosts the database is down the DBI->connect() call will hang for several minutes before returning an error.

The solution on these systems is to use the "POSIX::sigaction()" routine to gain low level access to how the signal handler is installed.

The code would look something like this (for the DBD-Oracle connect()):

```
  use POSIX qw(:signal_h);
  my $mask = POSIX::SigSet->new( SIGALRM ); # signals to mask in the handler
  my $action = POSIX::SigAction->new(
      sub { die "connect timeout\n" },        # the handler code ref
      $mask,
```

```
      # not using (perl 5.8.2 and later) 'safe' switch or sa_flags
  );
  my $oldaction = POSIX::SigAction->new();
  sigaction( SIGALRM, $action, $oldaction );
  my $dbh;
  my $failed;
  eval {
    eval {
      alarm(5); # seconds before time out
      $dbh = DBI->connect("dbi:Oracle:$dsn" ... );
      1;
    } or $failed = 1;
    alarm(0); # cancel alarm (if connect worked fast)
    die "$@\n" if $failed; # connect died
    1;
  } or $failed = 1;
  sigaction( SIGALRM, $oldaction );  # restore original signal handler
  if ( $failed ) {
   if ( defined $@ and $@ eq "connect timeout\n" ) {...}
   else { # connect died }
  }
```

See previous example for the reasoning around the double eval.

Similar techniques can be used for canceling statement execution.

Unfortunately, this solution is somewhat messy, and it does not work with perl versions less than perl 5.8 where "POSIX::sigaction()" appears to be broken.

For a cleaner implementation that works across perl versions, see Lincoln Baxter's Sys::SigAction module at Sys::SigAction. The documentation for Sys::SigAction includes an longer discussion of this problem, and a DBD::Oracle test script.

Be sure to read all the signal handling sections of the perlipc manual.

And finally, two more points to keep firmly in mind. Firstly, remember that what we've done here is essentially revert to old style unsafe handling of these signals. So do as little as possible in the handler. Ideally just die(). Secondly, the handles in use at the time the signal is handled may not be safe to use afterwards.

Subclassing the DBI

DBI can be subclassed and extended just like any other object oriented module. Before we talk about how to do that, it's important to be clear about the various DBI classes and how they work together.

By default "$dbh = DBI->connect(...)" returns a $dbh blessed into the "DBI::db" class. And the "$dbh->prepare" method returns an $sth blessed into the "DBI::st" class (actually it simply changes the last four characters of the calling handle class to be "::st").

The leading '"DBI"' is known as the 'root class' and the extra '"::db"' or '"::st"' are the 'handle type suffixes'. If you want to subclass the DBI you'll need to put your overriding methods into the appropriate classes. For example, if you want to use a root class of "MySubDBI" and override the do(), prepare() and execute() methods, then your do() and prepare() methods should be in the "MySubDBI::db" class and the execute() method should be in the "MySubDBI::st" class.

To setup the inheritance hierarchy the @ISA variable in "MySubDBI::db" should include "DBI::db" and the @ISA variable in "MySubDBI::st" should include "DBI::st". The "MySubDBI" root class itself isn't currently used for anything visible and so, apart from setting @ISA to include "DBI", it can be left empty.

So, having put your overriding methods into the right classes, and setup the inheritance hierarchy, how do you get the DBI to use them? You have two choices, either a static method call using the name of your subclass:

```
$dbh = MySubDBI->connect(...);
```

or specifying a "RootClass" attribute:

```
$dbh = DBI->connect(..., { RootClass => 'MySubDBI' });
```

If both forms are used then the attribute takes precedence.

The only differences between the two are that using an explicit RootClass attribute will a) make the DBI automatically attempt to load a module by that name if the class doesn't exist, and b) won't call your MySubDBI::connect() method, if you have one.

When subclassing is being used then, after a successful new connect, the DBI->connect method automatically calls:

```
$dbh->connected($dsn, $user, $pass, \%attr);
```

The default method does nothing. The call is made just to simplify any post-connection setup that your subclass may want to perform. The parameters are the same as passed to DBI->connect. If your subclass supplies a connected method, it should be part of the

MySubDBI::db package.

One more thing to note: you must let the DBI do the handle creation. If you want to override the connect() method in your *::dr class then it must still call SUPER::connect to get a $dbh to work with. Similarly, an overridden prepare() method in *::db must still call SUPER::prepare to get a $sth. If you try to create your own handles using bless() then you'll find the DBI will reject them with an "is not a DBI handle (has no magic)" error.

Here's a brief example of a DBI subclass. A more thorough example can be found in t/subclass.t in the DBI distribution.

```
package MySubDBI;
use strict;
use DBI;
use vars qw(@ISA);
@ISA = qw(DBI);
package MySubDBI::db;
use vars qw(@ISA);
@ISA = qw(DBI::db);
sub prepare {
  my ($dbh, @args) = @_;
  my $sth = $dbh->SUPER::prepare(@args)
      or return;
  $sth->{private_mysubdbi_info} = { foo => 'bar' };
  return $sth;
}
package MySubDBI::st;
use vars qw(@ISA);
@ISA = qw(DBI::st);
sub fetch {
  my ($sth, @args) = @_;
  my $row = $sth->SUPER::fetch(@args)
      or return;
  do_something_magical_with_row_data($row)
      or return $sth->set_err(1234, "The magic failed", undef, "fetch");
```

```
    return $row;
  }
```

When calling a SUPER::method that returns a handle, be careful to check the return value before trying to do other things with it in your overridden method. This is especially important if you want to set a hash attribute on the handle, as Perl's autovivification will bite you by (in)conveniently creating an unblessed hashref, which your method will then return with usually baffling results later on like the error "dbih_getcom handle HASH(0xa4451a8) is not a DBI handle (has no magic". It's best to check right after the call and return undef immediately on error, just like DBI would and just like the example above.

If your method needs to record an error it should call the set_err() method with the error code and error string, as shown in the example above. The error code and error string will be recorded in the handle and available via "$h->err" and $DBI::errstr etc. The set_err() method always returns an undef or empty list as appropriate. Since your method should nearly always return an undef or empty list as soon as an error is detected it's handy to simply return what set_err() returns, as shown in the example above.

If the handle has "RaiseError", "PrintError", or "HandleError" etc. set then the set_err() method will honour them. This means that if "RaiseError" is set then set_err() won't return in the normal way but will 'throw an exception' that can be caught with an "eval" block.

You can stash private data into DBI handles via "$h->{private_..._*}". See the entry under "ATTRIBUTES COMMON TO ALL HANDLES" for info and important caveats.

Memory Leaks

When tracking down memory leaks using tools like Devel::Leak you'll find that some DBI internals are reported as 'leaking' memory. This is very unlikely to be a real leak. The DBI has various caches to improve performance and the apparrent leaks are simply the normal operation of these caches.

The most frequent sources of the apparrent leaks are "ChildHandles", "prepare_cached" and "connect_cached".

For example http://stackoverflow.com/questions/13338308/perl-dbi-memory-leak

Given how widely the DBI is used, you can rest assured that if a new release of the DBI did have a real leak it would be discovered, reported, and fixed immediately. The leak you're looking for is probably elsewhere. Good luck!

TRACING

The DBI has a powerful tracing mechanism built in. It enables you to see what's going on 'behind the scenes', both within the DBI and the drivers you're using.

Trace Settings

Which details are written to the trace output is controlled by a combination of a trace level, an integer from 0 to 15, and a set of trace flags that are either on or off. Together these are known as the trace settings and are stored together in a single integer. For normal use you only need to set the trace level, and generally only to a value between 1 and 4.

Each handle has its own trace settings, and so does the DBI. When you call a method the DBI merges the handles settings into its own for the duration of the call: the trace flags of the handle are OR'd into the trace flags of the DBI, and if the handle has a higher trace level then the DBI trace level is raised to match it. The previous DBI trace settings are restored when the called method returns.

Trace Levels

Trace levels are as follows:

0 - Trace disabled.

1 - Trace top-level DBI method calls returning with results or errors.

2 - As above, adding tracing of top-level method entry with parameters.

3 - As above, adding some high-level information from the driver and some internal information from the DBI.

4 - As above, adding more detailed information from the driver. This is the first level to trace all the rows being fetched.

5 to 15 - As above but with more and more internal information.

Trace level 1 is best for a simple overview of what's happening. Trace levels 2 thru 4 a good choice for general purpose tracing. Levels 5 and above are best reserved for investigating a specific problem, when 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 "neat" function, so strings in the trace output may be edited and truncated by that function.

Trace Flags

Trace flags are used to enable tracing of specific activities within the DBI and drivers. The DBI defines some trace flags and drivers can define others. DBI trace flag names begin

with a capital letter and driver specific names begin with a lowercase letter, as usual.

Currently the DBI defines these trace flags:

```
ALL - turn on all DBI and driver flags (not recommended)
SQL - trace SQL statements executed
      (not yet implemented in DBI but implemented in some DBDs)
CON - trace connection process
ENC - trace encoding (unicode translations etc)
      (not yet implemented in DBI but implemented in some DBDs)
DBD - trace only DBD messages
      (not implemented by all DBDs yet)
TXN - trace transactions
      (not implemented in all DBDs yet)
```

The "parse_trace_flags" and "parse_trace_flag" methods are used to convert trace flag names into the corresponding integer bit flags.

Enabling Trace

The "$h->trace" method sets the trace settings for a handle and "DBI->trace" does the same for the DBI.

In addition to the "trace" method, you can enable the same trace information, and direct the output to a file, by setting the "DBI_TRACE" environment variable before starting Perl. See "DBI_TRACE" for more information.

Finally, you can set, or get, the trace settings for a handle using the "TraceLevel" attribute.

All of those methods use parse_trace_flags() and so allow you set both the trace level and multiple trace flags by using a string containing the trace level and/or flag names separated by vertical bar (""|"") or comma ("","") characters. For example:

```
local $h->{TraceLevel} = "3|SQL|foo";
```

Trace Output

Initially trace output is written to "STDERR". Both the "$h->trace" and "DBI->trace" methods take an optional $trace_file parameter, which may be either the name of a file to be opened by DBI in append mode, or a reference to an existing writable (possibly layered) filehandle. If $trace_file is a filename, and can be opened in append mode, or $trace_file is a writable filehandle, then all trace output (currently including that from other handles) is redirected to that file. A warning is generated if $trace_file can't be opened

or is not writable.

Further calls to trace() without $trace_file do not alter where the trace output is sent.

If $trace_file is undefined, then trace output is sent to "STDERR" and, if the prior trace was opened with $trace_file as a filename, the previous trace file is closed; if $trace_file was a filehandle, the filehandle is not closed.

NOTE: If $trace_file is specified as a filehandle, the filehandle should not be closed until all DBI operations are completed, or the application has reset the trace file via another call to "trace()" that changes the trace file.

Tracing to Layered Filehandles

NOTE:

? Tied filehandles are not currently supported, as tie operations are not available to the PerlIO methods used by the DBI.

? PerlIO layer support requires Perl version 5.8 or higher.

As of version 5.8, Perl provides the ability to layer various "disciplines" on an open filehandle via the PerlIO module.

A simple example of using PerlIO layers is to use a scalar as the output:

```
my $scalar = '';
open( my $fh, "+>:scalar", \$scalar );
$dbh->trace( 2, $fh );
```

Now all trace output is simply appended to $scalar.

A more complex application of tracing to a layered filehandle is the use of a custom layer (Refer to Perlio::via for details on creating custom PerlIO layers.). Consider an application with the following logger module:

```
package MyFancyLogger;
sub new
{
    my $self = {};
    my $fh;
    open $fh, '>', 'fancylog.log';
    $self->{_fh} = $fh;
    $self->{_buf} = '';
    return bless $self, shift;
}
```

```
sub log
{
    my $self = shift;
    return unless exists $self->{_fh};
    my $fh = $self->{_fh};
    $self->{_buf} .= shift;
#
# DBI feeds us pieces at a time, so accumulate a complete line
# before outputting
#
    print $fh "At ", scalar localtime(), ':', $self->{_buf}, "\n" and
    $self->{_buf} = ''
        if $self->{_buf}=~tr/\n//;
}
sub close {
    my $self = shift;
    return unless exists $self->{_fh};
    my $fh = $self->{_fh};
    print $fh "At ", scalar localtime(), ':', $self->{_buf}, "\n" and
    $self->{_buf} = ''
        if $self->{_buf};
    close $fh;
    delete $self->{_fh};
}
1;
```

To redirect DBI traces to this logger requires creating a package for the layer:

```
package PerlIO::via::MyFancyLogLayer;
sub PUSHED
{
    my ($class,$mode,$fh) = @_;
    my $logger;
    return bless \$logger,$class;
}
```

```
sub OPEN {
    my ($self, $path, $mode, $fh) = @_;
    #
    # $path is actually our logger object
    #
    $$self = $path;
    return 1;
}
sub WRITE
{
    my ($self, $buf, $fh) = @_;
    $$self->log($buf);
    return length($buf);
}
sub CLOSE {
    my $self = shift;
    $$self->close();
    return 0;
}
1;
```

The application can then cause DBI traces to be routed to the logger using

```
use PerlIO::via::MyFancyLogLayer;
open my $fh, '>:via(MyFancyLogLayer)', MyFancyLogger->new();
$dbh->trace('SQL', $fh);
```

Now all trace output will be processed by MyFancyLogger's log() method.

Trace Content

Many of the values embedded in trace output are formatted using the neat() utility function. This means they may be quoted, sanitized, and possibly truncated if longer than $DBI::neat_maxlen. See "neat" for more details.

Tracing Tips

You can add tracing to your own application code using the "trace_msg" method.

It can sometimes be handy to compare trace files from two different runs of the same script. However using a tool like "diff" on the original log output doesn't work well

because the trace file is full of object addresses that may differ on each run.

The DBI includes a handy utility called dbilogstrip that can be used to 'normalize' the log content. It can be used as a filter like this:

```
DBI_TRACE=2 perl yourscript.pl ...args1... 2>&1 | dbilogstrip > dbitrace1.log
DBI_TRACE=2 perl yourscript.pl ...args2... 2>&1 | dbilogstrip > dbitrace2.log
diff -u dbitrace1.log dbitrace2.log
```

See dbilogstrip for more information.

DBI ENVIRONMENT VARIABLES

The DBI module recognizes a number of environment variables, but most of them should not be used most of the time. It is better to be explicit about what you are doing to avoid the need for environment variables, especially in a web serving system where web servers are stingy about which environment variables are available.

DBI_DSN

The DBI_DSN environment variable is used by DBI->connect if you do not specify a data source when you issue the connect. It should have a format such as "dbi:Driver:databasename".

DBI_DRIVER

The DBI_DRIVER environment variable is used to fill in the database driver name in DBI->connect if the data source string starts "dbi::" (thereby omitting the driver). If DBI_DSN omits the driver name, DBI_DRIVER can fill the gap.

DBI_AUTOPROXY

The DBI_AUTOPROXY environment variable takes a string value that starts "dbi:Proxy:" and is typically followed by "hostname=...;port=...". It is used to alter the behaviour of DBI->connect. For full details, see DBI::Proxy documentation.

DBI_USER

The DBI_USER environment variable takes a string value that is used as the user name if the DBI->connect call is given undef (as distinct from an empty string) as the username argument. Be wary of the security implications of using this.

DBI_PASS

The DBI_PASS environment variable takes a string value that is used as the password if the DBI->connect call is given undef (as distinct from an empty string) as the password argument. Be extra wary of the security implications of using this.

DBI_DBNAME (obsolete)

The DBI_DBNAME environment variable takes a string value that is used only when the obsolescent style of DBI->connect (with driver name as fourth parameter) is used, and when no value is provided for the first (database name) argument.

DBI_TRACE

The DBI_TRACE environment variable specifies the global default trace settings for the DBI at startup. Can also be used to direct trace output to a file. When the DBI is loaded it does:

```
DBI->trace(split /=/, $ENV{DBI_TRACE}, 2) if $ENV{DBI_TRACE};
```

So if "DBI_TRACE" contains an ""="" character then what follows it is used as the name of the file to append the trace to.

output appended to that file. If the name begins with a number followed by an equal sign ("="), then the number and the equal sign are stripped off from the name, and the number is used to set the trace level. For example:

```
DBI_TRACE=1=dbitrace.log perl your_test_script.pl
```

On Unix-like systems using a Bourne-like shell, you can do this easily on the command line:

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

See "TRACING" for more information.

PERL_DBI_DEBUG (obsolete)

An old variable that should no longer be used; equivalent to DBI_TRACE.

DBI_PROFILE

The DBI_PROFILE environment variable can be used to enable profiling of DBI method calls. See DBI::Profile for more information.

DBI_PUREPERL

The DBI_PUREPERL environment variable can be used to enable the use of DBI::PurePerl. See DBI::PurePerl for more information.

WARNING AND ERROR MESSAGES

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 "RaiseError" 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 "RaiseError" attribute.

DBI/DBD internal version mismatch

The DBD driver module was built with a different version of DBI than the one currently being used. You should rebuild the DBD module under the current version of DBI. (Some rare platforms require "static linking". On those platforms, there may be an old DBI or DBD driver version actually embedded in the Perl executable being used.)

DBD driver has not implemented the AutoCommit attribute

The DBD driver implementation is incomplete. Consult the author.

Can't [sg]et %s->{%s}: unrecognised attribute

You attempted to set or get an unknown attribute of a handle. Make sure you have spelled the attribute name correctly; case is significant (e.g., "Autocommit" is not the same as "AutoCommit").

Pure-Perl DBI

A pure-perl emulation of the DBI is included in the distribution for people using pure-perl drivers who, for whatever reason, can't install the compiled DBI. See DBI::PurePerl.

SEE ALSO

Driver and Database Documentation

Refer to the documentation for the DBD driver that you are using.

Refer to the SQL Language Reference Manual for the database engine that you are using.

ODBC and SQL/CLI Standards Reference Information

More detailed information about the semantics of certain DBI methods that are based on ODBC and SQL/CLI standards is available on-line via microsoft.com, for ODBC, and www.jtc1sc32.org for the SQL/CLI standard:

```
DBI method       ODBC function     SQL/CLI Working Draft
----------       -------------     ---------------------
column_info      SQLColumns        Page 124
foreign_key_info SQLForeignKeys    Page 163
get_info         SQLGetInfo        Page 214
primary_key_info SQLPrimaryKeys    Page 254
table_info       SQLTables         Page 294
type_info        SQLGetTypeInfo    Page 239
```

statistics_info   SQLStatistics

To find documentation on the ODBC function you can use the MSDN search facility at:

http://msdn.microsoft.com/Search

and search for something like "SQLColumns returns".

And for SQL/CLI standard information on SQLColumns you'd read page 124 of the (very large)

SQL/CLI Working Draft available from:

http://jtc1sc32.org/doc/N0701-0750/32N0744T.pdf

Standards Reference Information

A hyperlinked, browsable version of the BNF syntax for SQL92 (plus Oracle 7 SQL and

PL/SQL) is available here:

http://cui.unige.ch/db-research/Enseignement/analyseinfo/SQL92/BNFindex.html

You can find more information about SQL standards online by searching for the appropriate

standard names and numbers. For example, searching for "ANSI/ISO/IEC International

Standard (IS) Database Language SQL - Part 1: SQL/Framework" you'll find a copy at:

ftp://ftp.iks-jena.de/mitarb/lutz/standards/sql/ansi-iso-9075-1-1999.pdf

Books and Articles

Programming the Perl DBI, by Alligator Descartes and Tim Bunce.

<http://books.perl.org/book/154>

Programming Perl 3rd Ed. by Larry Wall, Tom Christiansen & Jon Orwant.

<http://books.perl.org/book/134>

Learning Perl by Randal Schwartz. <http://books.perl.org/book/101>

Details of many other books related to perl can be found at <http://books.perl.org>

Perl Modules

Index of DBI related modules available from CPAN:

L<https://metacpan.org/search?q=DBD%3A%3A>

L<https://metacpan.org/search?q=DBIx%3A%3A>

L<https://metacpan.org/search?q=DBI>

For a good comparison of RDBMS-OO mappers and some OO-RDBMS mappers (including Class::DBI,

Alzabo, and DBIx::RecordSet in the former category and Tangram and SPOPS in the latter)

see the Perl Object-Oriented Persistence project pages at:

http://poop.sourceforge.net

A similar page for Java toolkits can be found at:

http://c2.com/cgi-bin/wiki?ObjectRelationalToolComparison

Mailing List

The dbi-users mailing list is the primary means of communication among users of the DBI and its related modules. For details send email to:

L<dbi-users-help@perl.org>

There are typically between 700 and 900 messages per month. You have to subscribe in order to be able to post. However you can opt for a 'post-only' subscription.

Mailing list archives (of variable quality) are held at:

http://groups.google.com/groups?group=perl.dbi.users

http://www.xray.mpe.mpg.de/mailing-lists/dbi/

http://www.mail-archive.com/dbi-users%40perl.org/

Assorted Related Links

The DBI "Home Page":

http://dbi.perl.org/

Other DBI related links:

http://www.perlmonks.org/?node=DBI%20recipes

http://www.perlmonks.org/?node=Speeding%20up%20the%20DBI

Other database related links:

http://www.connectionstrings.com/

Security, especially the "SQL Injection" attack:

http://bobby-tables.com/

http://online.securityfocus.com/infocus/1644

FAQ

See <http://faq.dbi-support.com/>

AUTHORS

DBI by Tim Bunce, <http://www.tim.bunce.name>

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) 1994-2012 Tim Bunce. Ireland. 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 5.10.0 README file.

SUPPORT / WARRANTY

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

Support

My consulting company, Data Plan Services, offers annual and multi-annual support contracts for the DBI. These provide sustained support for DBI development, and sustained value for you in return. Contact me for details.

Sponsor Enhancements

If your company would benefit from a specific new DBI feature, please consider sponsoring its development. Work is performed rapidly, and usually on a fixed-price payment-on-delivery basis. Contact me for details.

Using such targeted financing allows you to contribute to DBI development, and rapidly get something specific and valuable in return.

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 Peppler, 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.

Then, of course, there are the poor souls who have struggled through untold and undocumented obstacles to actually implement DBI drivers. Among their ranks are Jochen Wiedmann, Alligator Descartes, Jonathan Leffler, Jeff Urlwin, Michael Peppler, Henrik Tougaard, Edwin Pratomo, Davide Migliavacca, Jan Pazdziora, Peter Haworth, Edmund Mergl, Steve Williams, Thomas Lowery, and Phlip Plumlee. Without them, the DBI would not be the practical reality it is today. I'm also especially grateful to Alligator Descartes for starting work on the first edition of the "Programming the Perl DBI" book and letting me jump on board.

The DBI and DBD::Oracle were originally developed while I was Technical Director (CTO) of the Paul Ingram Group in the UK. So I'd especially like to thank Paul for his generosity and vision in supporting this work for many years.

A couple of specific DBI features have been sponsored by enlightened companies:

The development of the swap_inner_handle() method was sponsored by BizRate.com (<http://BizRate.com>)

The development of DBD::Gofer and related modules was sponsored by Shopzilla.com

(<http://Shopzilla.com>), where I currently work.

CONTRIBUTING

As you can see above, many people have contributed to the DBI and drivers in many ways over many years.

If you'd like to help then see <http://dbi.perl.org/contributing>.

If you'd like the DBI to do something new or different then a good way to make that happen is to do it yourself and send me a patch to the source code that shows the changes. (But read "Speak before you patch" below.)

Browsing the source code repository

Use https://github.com/perl5-dbi/dbi

How to create a patch using Git

The DBI source code is maintained using Git. To access the source you'll need to install a Git client. Then, to get the source code, do:

```
git clone https://github.com/perl5-dbi/dbi.git DBI-git
```

The source code will now be available in the new subdirectory "DBI-git".

When you want to synchronize later, issue the command

```
git pull --all
```

Make your changes, test them, test them again until everything passes. If there are no tests for the new feature you added or a behaviour change, the change should include a new test. Then commit the changes. Either use

```
git gui
```

or

```
git commit -a -m 'Message to my changes'
```

If you get any conflicts reported you'll need to fix them first.

Then generate the patch file to be mailed:

```
git format-patch -1 --attach
```

which will create a file 0001-*.patch (where * relates to the commit message). Read the patch file, as a sanity check, and then email it to dbi-dev@perl.org.

If you have a github <https://github.com> account, you can also fork the repository, commit your changes to the forked repository and then do a pull request.

How to create a patch without Git

Unpack a fresh copy of the distribution:

```
wget http://cpan.metacpan.org/authors/id/T/TI/TIMB/DBI-1.627.tar.gz
```

```
tar xfz DBI-1.627.tar.gz
```

Rename the newly created top level directory:

```
mv DBI-1.627 DBI-1.627.your_foo
```

Edit the contents of DBI-1.627.your_foo/* till it does what you want.

Test your changes and then remove all temporary files:

```
make test && make distclean
```

Go back to the directory you originally unpacked the distribution:

```
cd ..
```

Unpack another copy of the original distribution you started with:

```
tar xfz DBI-1.627.tar.gz
```

Then create a patch file by performing a recursive "diff" on the two top level

directories:

```
diff -purd DBI-1.627 DBI-1.627.your_foo > DBI-1.627.your_foo.patch
```

Speak before you patch

For anything non-trivial or possibly controversial it's a good idea to discuss (on dbi-dev@perl.org) the changes you propose before actually spending time working on them. Otherwise you run the risk of them being rejected because they don't fit into some larger plans you may not be aware of.

You can also reach the developers on IRC (chat). If they are on-line, the most likely place to talk to them is the #dbi channel on irc.perl.org

TRANSLATIONS

A German translation of this manual (possibly slightly out of date) is available, thanks to O'Reilly, at:

```
http://www.oreilly.de/catalog/perldbiger/
```

OTHER RELATED WORK AND PERL MODULES

Apache::DBI

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.

SQL Parser

See also the SQL::Statement module, SQL parser and engine.