MySQL and PHP

Abstract

This manual describes the PHP extensions and interfaces that can be used with MySQL.

For help with using MySQL, please visit either the MySQL Forums or MySQL Mailing Lists, where you can discuss your issues with other MySQL users.

For additional documentation on MySQL products, including translations of the documentation into other languages, and downloadable versions in variety of formats, including HTML and PDF formats, see the MySQL Documentation Library.

Document generated on: 2017-08-15 (revision: 53442)

Table of Contents

Preface and Legal Notices

This manual describes the PHP extensions and interfaces that can be used with MySQL.

Legal Notices

Chapter 1 Introduction to the MySQL PHP API

PHP is a server-side, HTML-embedded scripting language that may be used to create dynamic Web pages. It is available for most operating systems and Web servers, and can access most common databases, including MySQL. PHP may be run as a separate program or compiled as a module for use with a Web server.

PHP provides three different MySQL API extensions:

- Chapter 3, *MySQL Improved Extension*: Stands for “MySQL, Improved”; this extension is available as of PHP 5.0.0. It is intended for use with MySQL 4.1.1 and later. This extension fully supports the authentication protocol used in MySQL 5.0, as well as the Prepared Statements and Multiple Statements APIs. In addition, this extension provides an advanced, object-oriented programming interface.
- Chapter 4, *MySQL Functions (PDO_MYSQL)*: Not its own API, but instead it's a MySQL driver for the PHP database abstraction layer PDO (PHP Data Objects). The PDO MySQL driver sits in the layer below PDO itself, and provides MySQL-specific functionality. This extension is available as of PHP 5.1.0.
- Chapter 5, *Original MySQL API*: Available for PHP versions 4 and 5, this extension is intended for use with MySQL versions prior to MySQL 4.1. This extension does not support the improved authentication protocol used in MySQL 4.1, nor does it support prepared statements or multiple statements. To use this extension with MySQL 4.1, you will likely configure the MySQL server to set the `old_passwords` system variable to 1 (see Client does not support authentication protocol).

  > **Warning**
  >
  > This extension was removed from PHP 5.5.0. All users must migrate to either `mysqli` or `PDO_MySQL`. For further information, see Section 2.3, “Choosing an API”.

  > **Note**
  >
  > This documentation, and other publications, sometimes uses the term `Connector/PHP`. This term refers to the full set of MySQL related functionality in PHP, which includes the three APIs that are described in the preceding discussion, along with the mysqlnd core library and all of its plugins.

The PHP distribution and documentation are available from the PHP Web site.

Chapter 2 Overview of the MySQL PHP drivers

Table of Contents

2.1 Introduction

Depending on the version of PHP, there are either two or three PHP APIs for accessing the MySQL database. PHP 5 users can choose between the deprecated mysql extension, mysqli, or PDO_MySQL. PHP 7 removes the mysql extension, leaving only the latter two options.

This guide explains the terminology used to describe each API, information about choosing which API to use, and also information to help choose which MySQL library to use with the API.

2.2 Terminology overview

This section provides an introduction to the options available to you when developing a PHP application that needs to interact with a MySQL database.

What is an API?

An Application Programming Interface, or API, defines the classes, methods, functions and variables that your application will need to call in order to carry out its desired task. In the case of PHP applications that need to communicate with databases the necessary APIs are usually exposed via PHP extensions.

APIs can be procedural or object-oriented. With a procedural API you call functions to carry out tasks, with the object-oriented API you instantiate classes and then call methods on the resulting objects. Of the two the latter is usually the preferred interface, as it is more modern and leads to better organized code.

When writing PHP applications that need to connect to the MySQL server there are several API options available. This document discusses what is available and how to select the best solution for your application.

What is a Connector?

In the MySQL documentation, the term *connector* refers to a piece of software that allows your application to connect to the MySQL database server. MySQL provides connectors for a variety of languages, including PHP.

If your PHP application needs to communicate with a database server you will need to write PHP code to perform such activities as connecting to the database server, querying the database and other database-related functions. Software is required to provide the API that your PHP application will use, and also handle the communication between your application and the database server, possibly using other

intermediate libraries where necessary. This software is known generically as a connector, as it allows your application to *connect* to a database server.

What is a Driver?

A driver is a piece of software designed to communicate with a specific type of database server. The driver may also call a library, such as the MySQL Client Library or the MySQL Native Driver. These libraries implement the low-level protocol used to communicate with the MySQL database server.

By way of an example, the PHP Data Objects (PDO) database abstraction layer may use one of several database-specific drivers. One of the drivers it has available is the PDO MYSQL driver, which allows it to interface with the MySQL server.

Sometimes people use the terms connector and driver interchangeably, this can be confusing. In the MySQL-related documentation the term “driver” is reserved for software that provides the database-specific part of a connector package.

What is an Extension?

In the PHP documentation you will come across another term - *extension*. The PHP code consists of a core, with optional extensions to the core functionality. PHP's MySQL-related extensions, such as the `mysqli` extension, and the `mysql` extension, are implemented using the PHP extension framework.

An extension typically exposes an API to the PHP programmer, to allow its facilities to be used programmatically. However, some extensions which use the PHP extension framework do not expose an API to the PHP programmer.

The PDO MySQL driver extension, for example, does not expose an API to the PHP programmer, but provides an interface to the PDO layer above it.

The terms API and extension should not be taken to mean the same thing, as an extension may not necessarily expose an API to the programmer.

2.3 Choosing an API

PHP offers three different APIs to connect to MySQL. Below we show the APIs provided by the mysql, mysqli, and PDO extensions. Each code snippet creates a connection to a MySQL server running on "example.com" using the username "user" and the password "password". And a query is run to greet the user.

Example 2.1 Comparing the three MySQL APIs

```
<?php
// mysqli
$mysqli = new mysqli("example.com", "user", "password", "database");
$result = $mysqli->query("SELECT 'Hello, dear MySQL user!' AS _message FROM DUAL");
$row = $result->fetch_assoc();
echo htmlentities($row['_message']);

// PDO
$pdo = new PDO('mysql:host=example.com;dbname=database', 'user', 'password');
$statement = $pdo->query("SELECT 'Hello, dear MySQL user!' AS _message FROM DUAL");
$row = $statement->fetch(PDO::FETCH_ASSOC);
echo htmlentities($row['_message']);

// mysql
$c = mysql_connect("example.com", "user", "password");
```

```
mysql_select_db("database");
$result = mysql_query("SELECT 'Hello, dear MySQL user!' AS _message FROM DUAL");
$row = mysql_fetch_assoc($result);
echo htmlentities($row['_message']);
?>
```

Recommended API

It is recommended to use either the mysqli or PDO_MySQL extensions. It is not recommended to use the old mysql extension for new development, as it was deprecated in PHP 5.5.0 and was removed in PHP 7. A detailed feature comparison matrix is provided below. The overall performance of all three extensions is considered to be about the same. Although the performance of the extension contributes only a fraction of the total run time of a PHP web request. Often, the impact is as low as 0.1%.

Feature comparison

	ext/mysqli	**PDO_MySQL**	**ext/mysql**
PHP version introduced	5.0	5.1	2.0
Included with PHP 5.x	Yes	Yes	Yes
Included with PHP 7.x	Yes	Yes	No
Development status	Active	Active	Maintenance only in 5.x; removed in 7.x
Lifecycle	Active	Active	Deprecated in 5.x; removed in 7.x
Recommended for new projects	Yes	Yes	No
OOP Interface	Yes	Yes	No
Procedural Interface	Yes	No	Yes
API supports non-blocking, asynchronous queries with mysqlnd	Yes	No	No
Persistent Connections	Yes	Yes	Yes
API supports Charsets	Yes	Yes	Yes
API supports server-side Prepared Statements	Yes	Yes	No
API supports client-side Prepared Statements	No	Yes	No
API supports Stored Procedures	Yes	Yes	No
API supports Multiple Statements	Yes	Most	No
API supports Transactions	Yes	Yes	No
Transactions can be controlled with SQL	Yes	Yes	Yes
Supports all MySQL 5.1+ functionality	Yes	Most	No

2.4 Choosing a library

The mysqli, PDO_MySQL and mysql PHP extensions are lightweight wrappers on top of a C client library. The extensions can either use the mysqlnd library or the `libmysqlclient` library. Choosing a library is a compile time decision.

The mysqlnd library is part of the PHP distribution since 5.3.0. It offers features like lazy connections and query caching, features that are not available with libmysqlclient, so using the built-in mysqlnd library is highly recommended. See the mysqlnd documentation for additional details, and a listing of features and functionality that it offers.

Example 2.2 Configure commands for using mysqlnd or libmysqlclient

```
// Recommended, compiles with mysqlnd
$ ./configure --with-mysqli=mysqlnd --with-pdo-mysql=mysqlnd --with-mysql=mysqlnd

// Alternatively recommended, compiles with mysqlnd as of PHP 5.4
$ ./configure --with-mysqli --with-pdo-mysql --with-mysql

// Not recommended, compiles with libmysqlclient
$ ./configure --with-mysqli=/path/to/mysql_config --with-pdo-mysql=/path/to/mysql_config --with-mysql=/path/to
```

Library feature comparison

It is recommended to use the mysqlnd library instead of the MySQL Client Server library (libmysqlclient). Both libraries are supported and constantly being improved.

	MySQL native driver (mysqlnd)	**MySQL client server library (`libmysqlclient`)**
Part of the PHP distribution	Yes	No
PHP version introduced	5.3.0	N/A
License	PHP License 3.01	Dual-License
Development status	Active	Active
Lifecycle	No end announced	No end announced
PHP 5.4 and above; compile default (for all MySQL extensions)	Yes	No
PHP 5.3; compile default (for all MySQL extensions)	No	Yes
Compression protocol support	Yes (5.3.1+)	Yes
SSL support	Yes (5.3.3+)	Yes
Named pipe support	Yes (5.3.4+)	Yes
Non-blocking, asynchronous queries	Yes	No
Performance statistics	Yes	No
LOAD LOCAL INFILE respects the open_basedir directive	Yes	No

	MySQL native driver (mysqlnd)	MySQL client server library (libmysqlclient)
Uses PHP's native memory management system (e.g., follows PHP memory limits)	Yes	No
Return numeric column as double (COM_QUERY)	Yes	No
Return numeric column as string (COM_QUERY)	Yes	Yes
Plugin API	Yes	Limited
Read/Write splitting for MySQL Replication	Yes, with plugin	No
Load Balancing	Yes, with plugin	No
Fail over	Yes, with plugin	No
Lazy connections	Yes, with plugin	No
Query caching	Yes, with plugin	No
Transparent query manipulations (E.g., auto-EXPLAIN or monitoring)	Yes, with plugin	No
Automatic reconnect	No	Optional

2.5 Concepts

These concepts are specific to the MySQL drivers for PHP.

2.5.1 Buffered and Unbuffered queries

Queries are using the buffered mode by default. This means that query results are immediately transferred from the MySQL Server to PHP and then are kept in the memory of the PHP process. This allows additional operations like counting the number of rows, and moving (seeking) the current result pointer. It also allows issuing further queries on the same connection while working on the result set. The downside of the buffered mode is that larger result sets might require quite a lot memory. The memory will be kept occupied till all references to the result set are unset or the result set was explicitly freed, which will automatically happen during request end the latest. The terminology "store result" is also used for buffered mode, as the whole result set is stored at once.

> **Note**
>
> When using libmysqlclient as library PHP's memory limit won't count the memory used for result sets unless the data is fetched into PHP variables. With mysqlnd the memory accounted for will include the full result set.

Unbuffered MySQL queries execute the query and then return a resource while the data is still waiting on the MySQL server for being fetched. This uses less memory on the PHP-side, but can increase the load on the server. Unless the full result set was fetched from the server no further queries can be sent over the same connection. Unbuffered queries can also be referred to as "use result".

Following these characteristics buffered queries should be used in cases where you expect only a limited result set or need to know the amount of returned rows before reading all rows. Unbuffered mode should be used when you expect larger results.

Because buffered queries are the default, the examples below will demonstrate how to execute unbuffered queries with each API.

Example 2.3 Unbuffered query example: mysqli

```
<?php
$mysqli  = new mysqli("localhost", "my_user", "my_password", "world");
$uresult = $mysqli->query("SELECT Name FROM City", MYSQLI_USE_RESULT);

if ($uresult) {
   while ($row = $uresult->fetch_assoc()) {
       echo $row['Name'] . PHP_EOL;
   }
}
$uresult->close();
?>
```

Example 2.4 Unbuffered query example: pdo_mysql

```
<?php
$pdo = new PDO("mysql:host=localhost;dbname=world", 'my_user', 'my_pass');
$pdo->setAttribute(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, false);

$uresult = $pdo->query("SELECT Name FROM City");
if ($uresult) {
   while ($row = $uresult->fetch(PDO::FETCH_ASSOC)) {
       echo $row['Name'] . PHP_EOL;
   }
}
?>
```

Example 2.5 Unbuffered query example: mysql

```
<?php
$conn = mysql_connect("localhost", "my_user", "my_pass");
$db   = mysql_select_db("world");

$uresult = mysql_unbuffered_query("SELECT Name FROM City");
if ($uresult) {
   while ($row = mysql_fetch_assoc($uresult)) {
       echo $row['Name'] . PHP_EOL;
   }
}
?>
```

2.5.2 Character sets

Ideally a proper character set will be set at the server level, and doing this is described within the Character Set Configuration section of the MySQL Server manual. Alternatively, each MySQL API offers a method to set the character set at runtime.

> **The character set and character escaping**
>
> The character set should be understood and defined, as it has an affect on every action, and includes security implications. For example, the escaping mechanism (e.g., `mysqli_real_escape_string` for mysqli, `mysql_real_escape_string` for mysql, and `PDO::quote` for PDO_MySQL) will adhere to this setting. It is important to realize that these functions will not use the character set that is defined with a query, so for example the following will not have an effect on them:

Example 2.6 Problems with setting the character set with SQL

```
<?php

$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

// Will NOT affect $mysqli->real_escape_string();
$mysqli->query("SET NAMES utf8");

// Will NOT affect $mysqli->real_escape_string();
$mysqli->query("SET CHARACTER SET utf8");

// But, this will affect $mysqli->real_escape_string();
$mysqli->set_charset('utf8');

// But, this will NOT affect it (utf-8 vs utf8) -- don't use dashes here
$mysqli->set_charset('utf-8');

?>
```

Below are examples that demonstrate how to properly alter the character set at runtime using each API.

> **Possible UTF-8 confusion**
>
> Because character set names in MySQL do not contain dashes, the string "utf8" is valid in MySQL to set the character set to UTF-8. The string "utf-8" is not valid, as using "utf-8" will fail to change the character set.

Example 2.7 Setting the character set example: mysqli

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

printf("Initial character set: %s\n", $mysqli->character_set_name());

if (!$mysqli->set_charset('utf8')) {
    printf("Error loading character set utf8: %s\n", $mysqli->error);
    exit;
}

echo "New character set information:\n";
print_r( $mysqli->get_charset() );

?>
```

```
```

Example 2.8 Setting the character set example: pdo_mysql

Note: This only works as of PHP 5.3.6.

```
<?php
$pdo = new PDO("mysql:host=localhost;dbname=world;charset=utf8", 'my_user', 'my_pass');
?>
```

Example 2.9 Setting the character set example: mysql

```
<?php
$conn = mysql_connect("localhost", "my_user", "my_pass");
$db   = mysql_select_db("world");

echo 'Initial character set: ' .  mysql_client_encoding($conn) . "\n";

if (!mysql_set_charset('utf8', $conn)) {
    echo "Error: Unable to set the character set.\n";
    exit;
}

echo 'Your current character set is: ' .  mysql_client_encoding($conn);
?>
```

Chapter 3 MySQL Improved Extension

Table of Contents

The mysqli extension allows you to access the functionality provided by MySQL 4.1 and above. More information about the MySQL Database server can be found at http://www.mysql.com/

An overview of software available for using MySQL from PHP can be found at Section 3.1, “Overview”

Documentation for MySQL can be found at http://dev.mysql.com/doc/.

Parts of this documentation included from MySQL manual with permissions of Oracle Corporation.

Examples use either the world or sakila database, which are freely available.

3.1 Overview

This section provides an introduction to the options available to you when developing a PHP application that needs to interact with a MySQL database.

What is an API?

An Application Programming Interface, or API, defines the classes, methods, functions and variables that your application will need to call in order to carry out its desired task. In the case of PHP applications that need to communicate with databases the necessary APIs are usually exposed via PHP extensions.

APIs can be procedural or object-oriented. With a procedural API you call functions to carry out tasks, with the object-oriented API you instantiate classes and then call methods on the resulting objects. Of the two the latter is usually the preferred interface, as it is more modern and leads to better organized code.

When writing PHP applications that need to connect to the MySQL server there are several API options available. This document discusses what is available and how to select the best solution for your application.

What is a Connector?

In the MySQL documentation, the term *connector* refers to a piece of software that allows your application to connect to the MySQL database server. MySQL provides connectors for a variety of languages, including PHP.

If your PHP application needs to communicate with a database server you will need to write PHP code to perform such activities as connecting to the database server, querying the database and other database-related functions. Software is required to provide the API that your PHP application will use, and also handle the communication between your application and the database server, possibly using other intermediate libraries where necessary. This software is known generically as a connector, as it allows your application to *connect* to a database server.

What is a Driver?

A driver is a piece of software designed to communicate with a specific type of database server. The driver may also call a library, such as the MySQL Client Library or the MySQL Native Driver. These libraries implement the low-level protocol used to communicate with the MySQL database server.

By way of an example, the PHP Data Objects (PDO) database abstraction layer may use one of several database-specific drivers. One of the drivers it has available is the PDO MYSQL driver, which allows it to interface with the MySQL server.

Sometimes people use the terms connector and driver interchangeably, this can be confusing. In the MySQL-related documentation the term “driver” is reserved for software that provides the database-specific part of a connector package.

What is an Extension?

In the PHP documentation you will come across another term - *extension*. The PHP code consists of a core, with optional extensions to the core functionality. PHP's MySQL-related extensions, such as the `mysqli` extension, and the `mysql` extension, are implemented using the PHP extension framework.

An extension typically exposes an API to the PHP programmer, to allow its facilities to be used programmatically. However, some extensions which use the PHP extension framework do not expose an API to the PHP programmer.

The PDO MySQL driver extension, for example, does not expose an API to the PHP programmer, but provides an interface to the PDO layer above it.

The terms API and extension should not be taken to mean the same thing, as an extension may not necessarily expose an API to the programmer.

What are the main PHP API offerings for using MySQL?

There are three main API options when considering connecting to a MySQL database server:

- PHP's MySQL Extension
- PHP's mysqli Extension
- PHP Data Objects (PDO)

Each has its own advantages and disadvantages. The following discussion aims to give a brief introduction to the key aspects of each API.

What is PHP's MySQL Extension?

This is the original extension designed to allow you to develop PHP applications that interact with a MySQL database. The `mysql` extension provides a procedural interface and is intended for use only with MySQL versions older than 4.1.3. This extension can be used with versions of MySQL 4.1.3 or newer, but not all of the latest MySQL server features will be available.

> **Note**
>
> If you are using MySQL versions 4.1.3 or later it is *strongly* recommended that you use the `mysqli` extension instead.

The `mysql` extension source code is located in the PHP extension directory `ext/mysql`.

For further information on the `mysql` extension, see Chapter 5, *Original MySQL API*.

What is PHP's mysqli Extension?

The `mysqli` extension, or as it is sometimes known, the MySQL *improved* extension, was developed to take advantage of new features found in MySQL systems versions 4.1.3 and newer. The `mysqli` extension is included with PHP versions 5 and later.

The `mysqli` extension has a number of benefits, the key enhancements over the `mysql` extension being:

- Object-oriented interface
- Support for Prepared Statements
- Support for Multiple Statements
- Support for Transactions
- Enhanced debugging capabilities
- Embedded server support

> **Note**
>
> If you are using MySQL versions 4.1.3 or later it is *strongly* recommended that you use this extension.

As well as the object-oriented interface the extension also provides a procedural interface.

The `mysqli` extension is built using the PHP extension framework, its source code is located in the directory `ext/mysqli`.

For further information on the `mysqli` extension, see Chapter 3, *MySQL Improved Extension*.

What is PDO?

PHP Data Objects, or PDO, is a database abstraction layer specifically for PHP applications. PDO provides a consistent API for your PHP application regardless of the type of database server your application will connect to. In theory, if you are using the PDO API, you could switch the database server you used, from say Firebird to MySQL, and only need to make minor changes to your PHP code.

Other examples of database abstraction layers include JDBC for Java applications and DBI for Perl.

While PDO has its advantages, such as a clean, simple, portable API, its main disadvantage is that it doesn't allow you to use all of the advanced features that are available in the latest versions of MySQL server. For example, PDO does not allow you to use MySQL's support for Multiple Statements.

PDO is implemented using the PHP extension framework, its source code is located in the directory `ext/pdo`.

For further information on PDO, see the http://www.php.net/book.pdo.

What is the PDO MYSQL driver?

The PDO MYSQL driver is not an API as such, at least from the PHP programmer's perspective. In fact the PDO MYSQL driver sits in the layer below PDO itself and provides MySQL-specific functionality. The programmer still calls the PDO API, but PDO uses the PDO MYSQL driver to carry out communication with the MySQL server.

The PDO MYSQL driver is one of several available PDO drivers. Other PDO drivers available include those for the Firebird and PostgreSQL database servers.

The PDO MYSQL driver is implemented using the PHP extension framework. Its source code is located in the directory `ext/pdo_mysql`. It does not expose an API to the PHP programmer.

For further information on the PDO MYSQL driver, see Chapter 4, *MySQL Functions (PDO_MYSQL)*.

What is PHP's MySQL Native Driver?

In order to communicate with the MySQL database server the `mysql` extension, `mysqli` and the PDO MYSQL driver each use a low-level library that implements the required protocol. In the past, the only available library was the MySQL Client Library, otherwise known as `libmysqlclient`.

However, the interface presented by `libmysqlclient` was not optimized for communication with PHP applications, as `libmysqlclient` was originally designed with C applications in mind. For this reason the MySQL Native Driver, `mysqlnd`, was developed as an alternative to `libmysqlclient` for PHP applications.

The `mysql` extension, the `mysqli` extension and the PDO MySQL driver can each be individually configured to use either `libmysqlclient` or `mysqlnd`. As `mysqlnd` is designed specifically to be utilised in the PHP system it has numerous memory and speed enhancements over `libmysqlclient`. You are strongly encouraged to take advantage of these improvements.

> **Note**
>
> The MySQL Native Driver can only be used with MySQL server versions 4.1.3 and later.

The MySQL Native Driver is implemented using the PHP extension framework. The source code is located in `ext/mysqlnd`. It does not expose an API to the PHP programmer.

Comparison of Features

The following table compares the functionality of the three main methods of connecting to MySQL from PHP:

Table 3.1 Comparison of MySQL API options for PHP

	PHP's mysqli Extension	PDO (Using PDO MySQL Driver and MySQL Native Driver)	PHP's MySQL Extension
PHP version introduced	5.0	5.0	Prior to 3.0
Included with PHP 5.x	yes	yes	Yes
MySQL development status	Active development	Active development as of PHP 5.3	Maintenance only
Recommended by MySQL for new projects	Yes - preferred option	Yes	No
API supports Charsets	Yes	Yes	No
API supports server-side Prepared Statements	Yes	Yes	No
API supports client-side Prepared Statements	No	Yes	No

	PHP's mysqli Extension	PDO (Using PDO MySQL Driver and MySQL Native Driver)	PHP's MySQL Extension
API supports Stored Procedures	Yes	Yes	No
API supports Multiple Statements	Yes	Most	No
Supports all MySQL 4.1+ functionality	Yes	Most	No

3.2 Quick start guide

This quick start guide will help with choosing and gaining familiarity with the PHP MySQL API.

This quick start gives an overview on the mysqli extension. Code examples are provided for all major aspects of the API. Database concepts are explained to the degree needed for presenting concepts specific to MySQL.

Required: A familiarity with the PHP programming language, the SQL language, and basic knowledge of the MySQL server.

3.2.1 Dual procedural and object-oriented interface

The mysqli extension features a dual interface. It supports the procedural and object-oriented programming paradigm.

Users migrating from the old mysql extension may prefer the procedural interface. The procedural interface is similar to that of the old mysql extension. In many cases, the function names differ only by prefix. Some mysqli functions take a connection handle as their first argument, whereas matching functions in the old mysql interface take it as an optional last argument.

Example 3.1 Easy migration from the old mysql extension

```
<?php
$mysqli = mysqli_connect("example.com", "user", "password", "database");
$res = mysqli_query($mysqli, "SELECT 'Please, do not use ' AS _msg FROM DUAL");
$row = mysqli_fetch_assoc($res);
echo $row['_msg'];

$mysql = mysql_connect("example.com", "user", "password");
mysql_select_db("test");
$res = mysql_query("SELECT 'the mysql extension for new developments.' AS _msg FROM DUAL", $mysql);
$row = mysql_fetch_assoc($res);
echo $row['_msg'];
?>
```

The above example will output:

```
Please, do not use the mysql extension for new developments.
```

The object-oriented interface

In addition to the classical procedural interface, users can choose to use the object-oriented interface. The documentation is organized using the object-oriented interface. The object-oriented interface shows functions grouped by their purpose, making it easier to get started. The reference section gives examples for both syntax variants.

There are no significant performance differences between the two interfaces. Users can base their choice on personal preference.

Example 3.2 Object-oriented and procedural interface

```
<?php
$mysqli = mysqli_connect("example.com", "user", "password", "database");
if (mysqli_connect_errno($mysqli)) {
    echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$res = mysqli_query($mysqli, "SELECT 'A world full of ' AS _msg FROM DUAL");
$row = mysqli_fetch_assoc($res);
echo $row['_msg'];

$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: " . $mysqli->connect_error;
}

$res = $mysqli->query("SELECT 'choices to please everybody.' AS _msg FROM DUAL");
$row = $res->fetch_assoc();
echo $row['_msg'];
?>
```

The above example will output:

```
A world full of choices to please everybody.
```

The object oriented interface is used for the quickstart because the reference section is organized that way.

Mixing styles

It is possible to switch between styles at any time. Mixing both styles is not recommended for code clarity and coding style reasons.

Example 3.3 Bad coding style

```
<?php
```

```
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: " . $mysqli->connect_error;
}

$res = mysqli_query($mysqli, "SELECT 'Possible but bad style.' AS _msg FROM DUAL");
if (!$res) {
    echo "Failed to run query: (" . $mysqli->errno . ") " . $mysqli->error;
}

if ($row = $res->fetch_assoc()) {
    echo $row['_msg'];
}
?>
```

The above example will output:

```
Possible but bad style.
```

See also

`mysqli::__construct`
`mysqli::query`
`mysqli_result::fetch_assoc`
$mysqli::connect_errno
$mysqli::connect_error
$mysqli::errno
$mysqli::error
The MySQLi Extension Function Summary

3.2.2 Connections

The MySQL server supports the use of different transport layers for connections. Connections use TCP/IP, Unix domain sockets or Windows named pipes.

The hostname `localhost` has a special meaning. It is bound to the use of Unix domain sockets. It is not possible to open a TCP/IP connection using the hostname `localhost` you must use `127.0.0.1` instead.

Example 3.4 Special meaning of localhost

```
<?php
$mysqli = new mysqli("localhost", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}
echo $mysqli->host_info . "\n";

$mysqli = new mysqli("127.0.0.1", "user", "password", "database", 3306);
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}
```

```
echo $mysqli->host_info . "\n";
?>
```

The above example will output:

```
Localhost via UNIX socket
127.0.0.1 via TCP/IP
```

Connection parameter defaults

Depending on the connection function used, assorted parameters can be omitted. If a parameter is not provided, then the extension attempts to use the default values that are set in the PHP configuration file.

Example 3.5 Setting defaults

```
mysqli.default_host=192.168.2.27
mysqli.default_user=root
mysqli.default_pw=""
mysqli.default_port=3306
mysqli.default_socket=/tmp/mysql.sock
```

The resulting parameter values are then passed to the client library that is used by the extension. If the client library detects empty or unset parameters, then it may default to the library built-in values.

Built-in connection library defaults

If the host value is unset or empty, then the client library will default to a Unix socket connection on `localhost`. If socket is unset or empty, and a Unix socket connection is requested, then a connection to the default socket on `/tmp/mysql.sock` is attempted.

On Windows systems, the host name `.` is interpreted by the client library as an attempt to open a Windows named pipe based connection. In this case the socket parameter is interpreted as the pipe name. If not given or empty, then the socket (pipe name) defaults to `\\.\pipe\MySQL`.

If neither a Unix domain socket based not a Windows named pipe based connection is to be established and the port parameter value is unset, the library will default to port `3306`.

The mysqlnd library and the MySQL Client Library (libmysqlclient) implement the same logic for determining defaults.

Connection options

Connection options are available to, for example, set init commands which are executed upon connect, or for requesting use of a certain charset. Connection options must be set before a network connection is established.

For setting a connection option, the connect operation has to be performed in three steps: creating a connection handle with `mysqli_init`, setting the requested options using `mysqli_options`, and establishing the network connection with `mysqli_real_connect`.

Connection pooling

The mysqli extension supports persistent database connections, which are a special kind of pooled connections. By default, every database connection opened by a script is either explicitly closed by the user during runtime or released automatically at the end of the script. A persistent connection is not. Instead it is put into a pool for later reuse, if a connection to the same server using the same username, password, socket, port and default database is opened. Reuse saves connection overhead.

Every PHP process is using its own mysqli connection pool. Depending on the web server deployment model, a PHP process may serve one or multiple requests. Therefore, a pooled connection may be used by one or more scripts subsequently.

Persistent connection

If a unused persistent connection for a given combination of host, username, password, socket, port and default database can not be found in the connection pool, then mysqli opens a new connection. The use of persistent connections can be enabled and disabled using the PHP directive mysqli.allow_persistent. The total number of connections opened by a script can be limited with mysqli.max_links. The maximum number of persistent connections per PHP process can be restricted with mysqli.max_persistent. Please note, that the web server may spawn many PHP processes.

A common complain about persistent connections is that their state is not reset before reuse. For example, open and unfinished transactions are not automatically rolled back. But also, authorization changes which happened in the time between putting the connection into the pool and reusing it are not reflected. This may be seen as an unwanted side-effect. On the contrary, the name `persistent` may be understood as a promise that the state is persisted.

The mysqli extension supports both interpretations of a persistent connection: state persisted, and state reset before reuse. The default is reset. Before a persistent connection is reused, the mysqli extension implicitly calls `mysqli_change_user` to reset the state. The persistent connection appears to the user as if it was just opened. No artifacts from previous usages are visible.

The `mysqli_change_user` function is an expensive operation. For best performance, users may want to recompile the extension with the compile flag `MYSQLI_NO_CHANGE_USER_ON_PCONNECT` being set.

It is left to the user to choose between safe behavior and best performance. Both are valid optimization goals. For ease of use, the safe behavior has been made the default at the expense of maximum performance.

See also

`mysqli::__construct`
`mysqli::init`
`mysqli::options`
`mysqli::real_connect`
`mysqli::change_user`
$mysqli::host_info
MySQLi Configuration Options
Persistent Database Connections

3.2.3 Executing statements

Statements can be executed with the `mysqli_query`, `mysqli_real_query` and `mysqli_multi_query` functions. The `mysqli_query` function is the most common, and combines the

executing statement with a buffered fetch of its result set, if any, in one call. Calling `mysqli_query` is identical to calling `mysqli_real_query` followed by `mysqli_store_result`.

Example 3.6 Connecting to MySQL

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT)") ||
    !$mysqli->query("INSERT INTO test(id) VALUES (1)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

Buffered result sets

After statement execution results can be retrieved at once to be buffered by the client or by read row by row. Client-side result set buffering allows the server to free resources associated with the statement results as early as possible. Generally speaking, clients are slow consuming result sets. Therefore, it is recommended to use buffered result sets. `mysqli_query` combines statement execution and result set buffering.

PHP applications can navigate freely through buffered results. Navigation is fast because the result sets are held in client memory. Please, keep in mind that it is often easier to scale by client than it is to scale the server.

Example 3.7 Navigation through buffered results

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT)") ||
    !$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$res = $mysqli->query("SELECT id FROM test ORDER BY id ASC");

echo "Reverse order...\n";
for ($row_no = $res->num_rows - 1; $row_no >= 0; $row_no--) {
    $res->data_seek($row_no);
    $row = $res->fetch_assoc();
    echo " id = " . $row['id'] . "\n";
}

echo "Result set order...\n";
$res->data_seek(0);
while ($row = $res->fetch_assoc()) {
    echo " id = " . $row['id'] . "\n";
}
```

```
?>
```

The above example will output:

```
Reverse order...
 id = 3
 id = 2
 id = 1
Result set order...
 id = 1
 id = 2
 id = 3
```

Unbuffered result sets

If client memory is a short resource and freeing server resources as early as possible to keep server load low is not needed, unbuffered results can be used. Scrolling through unbuffered results is not possible before all rows have been read.

Example 3.8 Navigation through unbuffered results

```
<?php
$mysqli->real_query("SELECT id FROM test ORDER BY id ASC");
$res = $mysqli->use_result();

echo "Result set order...\n";
while ($row = $res->fetch_assoc()) {
    echo " id = " . $row['id'] . "\n";
}
?>
```

Result set values data types

The `mysqli_query`, `mysqli_real_query` and `mysqli_multi_query` functions are used to execute non-prepared statements. At the level of the MySQL Client Server Protocol, the command `COM_QUERY` and the text protocol are used for statement execution. With the text protocol, the MySQL server converts all data of a result sets into strings before sending. This conversion is done regardless of the SQL result set column data type. The mysql client libraries receive all column values as strings. No further client-side casting is done to convert columns back to their native types. Instead, all values are provided as PHP strings.

Example 3.9 Text protocol returns strings by default

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
```

```
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$res = $mysqli->query("SELECT id, label FROM test WHERE id = 1");
$row = $res->fetch_assoc();

printf("id = %s (%s)\n", $row['id'], gettype($row['id']));
printf("label = %s (%s)\n", $row['label'], gettype($row['label']));
?>
```

The above example will output:

```
id = 1 (string)
label = a (string)
```

It is possible to convert integer and float columns back to PHP numbers by setting the MYSQLI_OPT_INT_AND_FLOAT_NATIVE connection option, if using the mysqlnd library. If set, the mysqlnd library will check the result set meta data column types and convert numeric SQL columns to PHP numbers, if the PHP data type value range allows for it. This way, for example, SQL INT columns are returned as integers.

Example 3.10 Native data types with mysqlnd and connection option

```
<?php
$mysqli = mysqli_init();
$mysqli->options(MYSQLI_OPT_INT_AND_FLOAT_NATIVE, 1);
$mysqli->real_connect("example.com", "user", "password", "database");

if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$res = $mysqli->query("SELECT id, label FROM test WHERE id = 1");
$row = $res->fetch_assoc();

printf("id = %s (%s)\n", $row['id'], gettype($row['id']));
printf("label = %s (%s)\n", $row['label'], gettype($row['label']));
?>
```

The above example will output:

```
id = 1 (integer)
label = a (string)
```

See also

mysqli::__construct
mysqli::init
mysqli::options
mysqli::real_connect
mysqli::query
mysqli::multi_query
mysqli::use_result
mysqli::store_result
mysqli_result::free

3.2.4 Prepared Statements

The MySQL database supports prepared statements. A prepared statement or a parameterized statement is used to execute the same statement repeatedly with high efficiency.

Basic workflow

The prepared statement execution consists of two stages: prepare and execute. At the prepare stage a statement template is sent to the database server. The server performs a syntax check and initializes server internal resources for later use.

The MySQL server supports using anonymous, positional placeholder with ?.

Example 3.11 First stage: prepare

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

/* Non-prepared statement */
if (!$mysqli->query("DROP TABLE IF EXISTS test") || !$mysqli->query("CREATE TABLE test(id INT)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

/* Prepared statement, stage 1: prepare */
if (!($stmt = $mysqli->prepare("INSERT INTO test(id) VALUES (?)"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

Prepare is followed by execute. During execute the client binds parameter values and sends them to the server. The server creates a statement from the statement template and the bound values to execute it using the previously created internal resources.

Example 3.12 Second stage: bind and execute

```
<?php
/* Prepared statement, stage 2: bind and execute */
```

```
$id = 1;
if (!$stmt->bind_param("i", $id)) {
    echo "Binding parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}
?>
```

Repeated execution

A prepared statement can be executed repeatedly. Upon every execution the current value of the bound variable is evaluated and sent to the server. The statement is not parsed again. The statement template is not transferred to the server again.

Example 3.13 INSERT prepared once, executed multiple times

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

/* Non-prepared statement */
if (!$mysqli->query("DROP TABLE IF EXISTS test") || !$mysqli->query("CREATE TABLE test(id INT)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

/* Prepared statement, stage 1: prepare */
if (!($stmt = $mysqli->prepare("INSERT INTO test(id) VALUES (?)"))) {
     echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

/* Prepared statement, stage 2: bind and execute */
$id = 1;
if (!$stmt->bind_param("i", $id)) {
    echo "Binding parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

/* Prepared statement: repeated execution, only data transferred from client to server */
for ($id = 2; $id < 5; $id++) {
    if (!$stmt->execute()) {
        echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
    }
}

/* explicit close recommended */
$stmt->close();

/* Non-prepared statement */
$res = $mysqli->query("SELECT id FROM test");
var_dump($res->fetch_all());
?>
```

The above example will output:

```
array(4) {
  [0]=>
  array(1) {
    [0]=>
    string(1) "1"
  }
  [1]=>
  array(1) {
    [0]=>
    string(1) "2"
  }
  [2]=>
  array(1) {
    [0]=>
    string(1) "3"
  }
  [3]=>
  array(1) {
    [0]=>
    string(1) "4"
  }
}
```

Every prepared statement occupies server resources. Statements should be closed explicitly immediately after use. If not done explicitly, the statement will be closed when the statement handle is freed by PHP.

Using a prepared statement is not always the most efficient way of executing a statement. A prepared statement executed only once causes more client-server round-trips than a non-prepared statement. This is why the `SELECT` is not run as a prepared statement above.

Also, consider the use of the MySQL multi-INSERT SQL syntax for INSERTs. For the example, multi-INSERT requires less round-trips between the server and client than the prepared statement shown above.

Example 3.14 Less round trips using multi-INSERT SQL

```
<?php
if (!$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3), (4)")) {
    echo "Multi-INSERT failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

Result set values data types

The MySQL Client Server Protocol defines a different data transfer protocol for prepared statements and non-prepared statements. Prepared statements are using the so called binary protocol. The MySQL server sends result set data "as is" in binary format. Results are not serialized into strings before sending. The client libraries do not receive strings only. Instead, they will receive binary data and try to convert the values into appropriate PHP data types. For example, results from an SQL `INT` column will be provided as PHP integer variables.

Example 3.15 Native datatypes

```
```

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$stmt = $mysqli->prepare("SELECT id, label FROM test WHERE id = 1");
$stmt->execute();
$res = $stmt->get_result();
$row = $res->fetch_assoc();

printf("id = %s (%s)\n", $row['id'], gettype($row['id']));
printf("label = %s (%s)\n", $row['label'], gettype($row['label']));
?>
```

The above example will output:

```
id = 1 (integer)
label = a (string)
```

This behavior differs from non-prepared statements. By default, non-prepared statements return all results as strings. This default can be changed using a connection option. If the connection option is used, there are no differences.

Fetching results using bound variables

Results from prepared statements can either be retrieved by binding output variables, or by requesting a `mysqli_result` object.

Output variables must be bound after statement execution. One variable must be bound for every column of the statements result set.

Example 3.16 Output variable binding

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($stmt = $mysqli->prepare("SELECT id, label FROM test"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
```

```
if (!$stmt->execute()) {
    echo "Execute failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$out_id    = NULL;
$out_label = NULL;
if (!$stmt->bind_result($out_id, $out_label)) {
    echo "Binding output parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}

while ($stmt->fetch()) {
    printf("id = %s (%s), label = %s (%s)\n", $out_id, gettype($out_id), $out_label, gettype($out_label));
}
?>
```

The above example will output:

```
id = 1 (integer), label = a (string)
```

Prepared statements return unbuffered result sets by default. The results of the statement are not implicitly fetched and transferred from the server to the client for client-side buffering. The result set takes server resources until all results have been fetched by the client. Thus it is recommended to consume results timely. If a client fails to fetch all results or the client closes the statement before having fetched all data, the data has to be fetched implicitly by `mysqli`.

It is also possible to buffer the results of a prepared statement using `mysqli_stmt_store_result`.

Fetching results using mysqli_result interface

Instead of using bound results, results can also be retrieved through the mysqli_result interface. `mysqli_stmt_get_result` returns a buffered result set.

Example 3.17 Using mysqli_result to fetch results

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($stmt = $mysqli->prepare("SELECT id, label FROM test ORDER BY id ASC"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$stmt->execute()) {
     echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!($res = $stmt->get_result())) {
    echo "Getting result set failed: (" . $stmt->errno . ") " . $stmt->error;
```

```
}

var_dump($res->fetch_all());
?>
```

The above example will output:

```
array(1) {
  [0]=>
  array(2) {
    [0]=>
    int(1)
    [1]=>
    string(1) "a"
  }
}
```

Using the `mysqli_result interface` offers the additional benefit of flexible client-side result set navigation.

Example 3.18 Buffered result set for flexible read out

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a'), (2, 'b'), (3, 'c')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($stmt = $mysqli->prepare("SELECT id, label FROM test"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$stmt->execute()) {
     echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!($res = $stmt->get_result())) {
    echo "Getting result set failed: (" . $stmt->errno . ") " . $stmt->error;
}

for ($row_no = ($res->num_rows - 1); $row_no >= 0; $row_no--) {
    $res->data_seek($row_no);
    var_dump($res->fetch_assoc());
}
$res->close();
?>
```

The above example will output:

```
array(2) {
  ["id"]=>
  int(3)
  ["label"]=>
  string(1) "c"
}
array(2) {
  ["id"]=>
  int(2)
  ["label"]=>
  string(1) "b"
}
array(2) {
  ["id"]=>
  int(1)
  ["label"]=>
  string(1) "a"
}
```

Escaping and SQL injection

Bound variables are sent to the server separately from the query and thus cannot interfere with it. The server uses these values directly at the point of execution, after the statement template is parsed. Bound parameters do not need to be escaped as they are never substituted into the query string directly. A hint must be provided to the server for the type of bound variable, to create an appropriate conversion. See the `mysqli_stmt_bind_param` function for more information.

Such a separation sometimes considered as the only security feature to prevent SQL injection, but the same degree of security can be achieved with non-prepared statements, if all the values are formatted correctly. It should be noted that correct formatting is not the same as escaping and involves more logic than simple escaping. Thus, prepared statements are simply a more convenient and less error-prone approach to this element of database security.

Client-side prepared statement emulation

The API does not include emulation for client-side prepared statement emulation.

Quick prepared - non-prepared statement comparison

The table below compares server-side prepared and non-prepared statements.

Table 3.2 Comparison of prepared and non-prepared statements

	Prepared Statement	**Non-prepared statement**
Client-server round trips, SELECT, single execution	2	1
Statement string transferred from client to server	1	1
Client-server round trips, SELECT, repeated (n) execution	1 + n	n
Statement string transferred from client to server	1 template, n times bound parameter, if any	n times together with parameter, if any
Input parameter binding API	Yes, automatic input escaping	No, manual input escaping

	Prepared Statement	Non-prepared statement
Output variable binding API	Yes	No
Supports use of mysqli_result API	Yes, use `mysqli_stmt_get_result`	Yes
Buffered result sets	Yes, use `mysqli_stmt_get_result` or binding with `mysqli_stmt_store_result`	Yes, default of `mysqli_query`
Unbuffered result sets	Yes, use output binding API	Yes, use `mysqli_real_query` with `mysqli_use_result`
MySQL Client Server protocol data transfer flavor	Binary protocol	Text protocol
Result set values SQL data types	Preserved when fetching	Converted to string or preserved when fetching
Supports all SQL statements	Recent MySQL versions support most but not all	Yes

See also

`mysqli::__construct`
`mysqli::query`
`mysqli::prepare`
`mysqli_stmt::prepare`
`mysqli_stmt::execute`
`mysqli_stmt::bind_param`
`mysqli_stmt::bind_result`

3.2.5 Stored Procedures

The MySQL database supports stored procedures. A stored procedure is a subroutine stored in the database catalog. Applications can call and execute the stored procedure. The `CALL` SQL statement is used to execute a stored procedure.

Parameter

Stored procedures can have `IN`, `INOUT` and `OUT` parameters, depending on the MySQL version. The mysqli interface has no special notion for the different kinds of parameters.

IN parameter

Input parameters are provided with the `CALL` statement. Please, make sure values are escaped correctly.

Example 3.19 Calling a stored procedure

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}
```

```
if (!$mysqli->query("DROP TABLE IF EXISTS test") || !$mysqli->query("CREATE TABLE test(id INT)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$mysqli->query("DROP PROCEDURE IF EXISTS p") ||
    !$mysqli->query("CREATE PROCEDURE p(IN id_val INT) BEGIN INSERT INTO test(id) VALUES(id_val); END;")) {
    echo "Stored procedure creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$mysqli->query("CALL p(1)")) {
    echo "CALL failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($res = $mysqli->query("SELECT id FROM test"))) {
    echo "SELECT failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

var_dump($res->fetch_assoc());
?>
```

The above example will output:

```
array(1) {
  ["id"]=>
  string(1) "1"
}
```

INOUT/OUT parameter

The values of `INOUT`/`OUT` parameters are accessed using session variables.

Example 3.20 Using session variables

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP PROCEDURE IF EXISTS p") ||
    !$mysqli->query('CREATE PROCEDURE p(OUT msg VARCHAR(50)) BEGIN SELECT "Hi!" INTO msg; END;')) {
    echo "Stored procedure creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}


if (!$mysqli->query("SET @msg = ''") || !$mysqli->query("CALL p(@msg)")) {
    echo "CALL failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($res = $mysqli->query("SELECT @msg as _p_out"))) {
    echo "Fetch failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$row = $res->fetch_assoc();
echo $row['_p_out'];
?>
```

The above example will output:

```
Hi!
```

Application and framework developers may be able to provide a more convenient API using a mix of session variables and databased catalog inspection. However, please note the possible performance impact of a custom solution based on catalog inspection.

Handling result sets

Stored procedures can return result sets. Result sets returned from a stored procedure cannot be fetched correctly using `mysqli_query`. The `mysqli_query` function combines statement execution and fetching the first result set into a buffered result set, if any. However, there are additional stored procedure result sets hidden from the user which cause `mysqli_query` to fail returning the user expected result sets.

Result sets returned from a stored procedure are fetched using `mysqli_real_query` or `mysqli_multi_query`. Both functions allow fetching any number of result sets returned by a statement, such as `CALL`. Failing to fetch all result sets returned by a stored procedure causes an error.

Example 3.21 Fetching results from stored procedures

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT)") ||
    !$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$mysqli->query("DROP PROCEDURE IF EXISTS p") ||
    !$mysqli->query('CREATE PROCEDURE p() READS SQL DATA BEGIN SELECT id FROM test; SELECT id + 1 FROM tes
    echo "Stored procedure creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$mysqli->multi_query("CALL p()")) {
    echo "CALL failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

do {
    if ($res = $mysqli->store_result()) {
        printf("---\n");
        var_dump($res->fetch_all());
        $res->free();
    } else {
        if ($mysqli->errno) {
            echo "Store failed: (" . $mysqli->errno . ") " . $mysqli->error;
        }
    }
} while ($mysqli->more_results() && $mysqli->next_result());
?>
```

The above example will output:

```
---
array(3) {
  [0]=>
  array(1) {
    [0]=>
    string(1) "1"
  }
  [1]=>
  array(1) {
    [0]=>
    string(1) "2"
  }
  [2]=>
  array(1) {
    [0]=>
    string(1) "3"
  }
}
---
array(3) {
  [0]=>
  array(1) {
    [0]=>
    string(1) "2"
  }
  [1]=>
  array(1) {
    [0]=>
    string(1) "3"
  }
  [2]=>
  array(1) {
    [0]=>
    string(1) "4"
  }
}
```

Use of prepared statements

No special handling is required when using the prepared statement interface for fetching results from the same stored procedure as above. The prepared statement and non-prepared statement interfaces are similar. Please note, that not every MYSQL server version may support preparing the `CALL` SQL statement.

Example 3.22 Stored Procedures and Prepared Statements

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT)") ||
    !$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)")) {
```

```
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$mysqli->query("DROP PROCEDURE IF EXISTS p") ||
    !$mysqli->query('CREATE PROCEDURE p() READS SQL DATA BEGIN SELECT id FROM test; SELECT id + 1 FROM tes
    echo "Stored procedure creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($stmt = $mysqli->prepare("CALL p()"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

do {
    if ($res = $stmt->get_result()) {
        printf("---\n");
        var_dump(mysqli_fetch_all($res));
        mysqli_free_result($res);
    } else {
        if ($stmt->errno) {
            echo "Store failed: (" . $stmt->errno . ") " . $stmt->error;
        }
    }
} while ($stmt->more_results() && $stmt->next_result());
?>
```

Of course, use of the bind API for fetching is supported as well.

Example 3.23 Stored Procedures and Prepared Statements using bind API

```
<?php
if (!($stmt = $mysqli->prepare("CALL p()"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

do {

    $id_out = NULL;
    if (!$stmt->bind_result($id_out)) {
        echo "Bind failed: (" . $stmt->errno . ") " . $stmt->error;
    }

    while ($stmt->fetch()) {
        echo "id = $id_out\n";
    }
} while ($stmt->more_results() && $stmt->next_result());
?>
```

See also

mysqli::query
mysqli::multi_query
mysqli_result::next-result

mysqli_result::more-results

3.2.6 Multiple Statements

MySQL optionally allows having multiple statements in one statement string. Sending multiple statements at once reduces client-server round trips but requires special handling.

Multiple statements or multi queries must be executed with mysqli_multi_query. The individual statements of the statement string are separated by semicolon. Then, all result sets returned by the executed statements must be fetched.

The MySQL server allows having statements that do return result sets and statements that do not return result sets in one multiple statement.

Example 3.24 Multiple Statements

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") || !$mysqli->query("CREATE TABLE test(id INT)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$sql = "SELECT COUNT(*) AS _num FROM test; ";
$sql.= "INSERT INTO test(id) VALUES (1); ";
$sql.= "SELECT COUNT(*) AS _num FROM test; ";

if (!$mysqli->multi_query($sql)) {
    echo "Multi query failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

do {
    if ($res = $mysqli->store_result()) {
        var_dump($res->fetch_all(MYSQLI_ASSOC));
        $res->free();
    }
} while ($mysqli->more_results() && $mysqli->next_result());
?>
```

The above example will output:

```
array(1) {
  [0]=>
  array(1) {
    ["_num"]=>
    string(1) "0"
  }
}
array(1) {
  [0]=>
  array(1) {
    ["_num"]=>
```

```
    string(1) "1"
  }
}
```

Security considerations

The API functions `mysqli_query` and `mysqli_real_query` do not set a connection flag necessary for activating multi queries in the server. An extra API call is used for multiple statements to reduce the likeliness of accidental SQL injection attacks. An attacker may try to add statements such as `; DROP DATABASE mysql` or `; SELECT SLEEP(999)`. If the attacker succeeds in adding SQL to the statement string but `mysqli_multi_query` is not used, the server will not execute the second, injected and malicious SQL statement.

Example 3.25 SQL Injection

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
$res    = $mysqli->query("SELECT 1; DROP TABLE mysql.user");
if (!$res) {
    echo "Error executing query: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

The above example will output:

```
Error executing query: (1064) You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server version for the right syntax
to use near 'DROP TABLE mysql.user' at line 1
```

Prepared statements

Use of the multiple statement with prepared statements is not supported.

See also

`mysqli::query`
`mysqli::multi_query`
`mysqli_result::next-result`
`mysqli_result::more-results`

3.2.7 API support for transactions

The MySQL server supports transactions depending on the storage engine used. Since MySQL 5.5, the default storage engine is InnoDB. InnoDB has full ACID transaction support.

Transactions can either be controlled using SQL or API calls. It is recommended to use API calls for enabling and disabling the auto commit mode and for committing and rolling back transactions.

Example 3.26 Setting auto commit mode with SQL and through the API

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

/* Recommended: using API to control transactional settings */
$mysqli->autocommit(false);

/* Won't be monitored and recognized by the replication and the load balancing plugin */
if (!$mysqli->query('SET AUTOCOMMIT = 0')) {
    echo "Query failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

Optional feature packages, such as the replication and load balancing plugin, can easily monitor API calls. The replication plugin offers transaction aware load balancing, if transactions are controlled with API calls. Transaction aware load balancing is not available if SQL statements are used for setting auto commit mode, committing or rolling back a transaction.

Example 3.27 Commit and rollback

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
$mysqli->autocommit(false);

$mysqli->query("INSERT INTO test(id) VALUES (1)");
$mysqli->rollback();

$mysqli->query("INSERT INTO test(id) VALUES (2)");
$mysqli->commit();
?>
```

Please note, that the MySQL server cannot roll back all statements. Some statements cause an implicit commit.

See also

`mysqli::autocommit`
`mysqli_result::commit`
`mysqli_result::rollback`

3.2.8 Metadata

A MySQL result set contains metadata. The metadata describes the columns found in the result set. All metadata sent by MySQL is accessible through the `mysqli` interface. The extension performs no or negligible changes to the information it receives. Differences between MySQL server versions are not aligned.

Meta data is access through the `mysqli_result` interface.

Example 3.28 Accessing result set meta data

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

$res = $mysqli->query("SELECT 1 AS _one, 'Hello' AS _two FROM DUAL");
var_dump($res->fetch_fields());
?>
```

The above example will output:

```
array(2) {
  [0]=>
  object(stdClass)#3 (13) {
    ["name"]=>
    string(4) "_one"
    ["orgname"]=>
    string(0) ""
    ["table"]=>
    string(0) ""
    ["orgtable"]=>
    string(0) ""
    ["def"]=>
    string(0) ""
    ["db"]=>
    string(0) ""
    ["catalog"]=>
    string(3) "def"
    ["max_length"]=>
    int(1)
    ["length"]=>
    int(1)
    ["charsetnr"]=>
    int(63)
    ["flags"]=>
    int(32897)
    ["type"]=>
    int(8)
    ["decimals"]=>
    int(0)
  }
  [1]=>
  object(stdClass)#4 (13) {
    ["name"]=>
    string(4) "_two"
    ["orgname"]=>
    string(0) ""
    ["table"]=>
    string(0) ""
    ["orgtable"]=>
    string(0) ""
    ["def"]=>
    string(0) ""
    ["db"]=>
    string(0) ""
    ["catalog"]=>
    string(3) "def"
    ["max_length"]=>
```

```
    int(5)
    ["length"]=>
    int(5)
    ["charsetnr"]=>
    int(8)
    ["flags"]=>
    int(1)
    ["type"]=>
    int(253)
    ["decimals"]=>
    int(31)
  }
}
```

Prepared statements

Meta data of result sets created using prepared statements are accessed the same way. A suitable `mysqli_result` handle is returned by `mysqli_stmt_result_metadata`.

Example 3.29 Prepared statements metadata

```
<?php
$stmt = $mysqli->prepare("SELECT 1 AS _one, 'Hello' AS _two FROM DUAL");
$stmt->execute();
$res = $stmt->result_metadata();
var_dump($res->fetch_fields());
?>
```

See also

`mysqli::query`
`mysqli_result::fetch_fields`

3.3 Installing/Configuring

3.3.1 Requirements

In order to have these functions available, you must compile PHP with support for the mysqli extension.

> **Note**
>
> The mysqli extension is designed to work with MySQL version 4.1.13 or newer, or 5.0.7 or newer. For previous versions, please see the MySQL extension documentation.

3.3.2 Installation

The `mysqli` extension was introduced with PHP version 5.0.0. The MySQL Native Driver was included in PHP version 5.3.0.

3.3.2.1 Installation on Linux

The common Unix distributions include binary versions of PHP that can be installed. Although these binary versions are typically built with support for the MySQL extensions, the extension libraries themselves may need to be installed using an additional package. Check the package manager that comes with your chosen distribution for availability.

For example, on Ubuntu the `php5-mysql` package installs the ext/mysql, ext/mysqli, and pdo_mysql PHP extensions. On CentOS, the `php-mysql` package also installs these three PHP extensions.

Alternatively, you can compile this extension yourself. Building PHP from source allows you to specify the MySQL extensions you want to use, as well as your choice of client library for each extension.

The MySQL Native Driver is the recommended client library option, as it results in improved performance and gives access to features not available when using the MySQL Client Library. Refer to What is PHP's MySQL Native Driver? for a brief overview of the advantages of MySQL Native Driver.

The `/path/to/mysql_config` represents the location of the `mysql_config` program that comes with MySQL Server.

Table 3.3 mysqli compile time support matrix

PHP Version	Default	Configure Options: mysqlnd	Configure Options: `libmysqlclient`	Changelog
5.4.x and above	mysqlnd	`--with-mysqli`	`--with-mysqli=/path/to/mysql_config`	mysqlnd is the default
5.3.x	libmysqlclient	`--with-mysqli=mysqlnd`	`--with-mysqli=/path/to/mysql_config`	mysqlnd is supported
5.0.x, 5.1.x, 5.2.x	libmysqlclient	Not Available	`--with-mysqli=/path/to/mysql_config`	mysqlnd is not supported

Note that it is possible to freely mix MySQL extensions and client libraries. For example, it is possible to enable the MySQL extension to use the MySQL Client Library (libmysqlclient), while configuring the `mysqli` extension to use the MySQL Native Driver. However, all permutations of extension and client library are possible.

3.3.2.2 Installation on Windows Systems

On Windows, PHP is most commonly installed using the binary installer.

PHP 5.3.0 and newer

On Windows, for PHP versions 5.3 and newer, the `mysqli` extension is enabled and uses the MySQL Native Driver by default. This means you don't need to worry about configuring access to `libmysql.dll`.

PHP 5.0, 5.1, 5.2

On these old unsupported PHP versions (PHP 5.2 reached EOL on '6 Jan 2011'), additional configuration procedures are required to enable `mysqli` and specify the client library you want it to use.

The `mysqli` extension is not enabled by default, so the `php_mysqli.dll` DLL must be enabled inside of `php.ini`. In order to do this you need to find the `php.ini` file (typically located in `c:\php`), and make sure you remove the comment (semi-colon) from the start of the line `extension=php_mysqli.dll`, in the section marked `[PHP_MYSQLI]`.

Also, if you want to use the MySQL Client Library with `mysqli`, you need to make sure PHP can access the client library file. The MySQL Client Library is included as a file named `libmysql.dll` in the Windows PHP distribution. This file needs to be available in the Windows system's `PATH` environment variable, so that it can be successfully loaded. See the FAQ titled "How do I add my PHP directory to the PATH on Windows" for information on how to do this. Copying `libmysql.dll` to the Windows system directory (typically `c:\Windows\system`) also works, as the system directory is by default in the system's `PATH`. However, this practice is strongly discouraged.

As with enabling any PHP extension (such as `php_mysqli.dll`), the PHP directive extension_dir should be set to the directory where the PHP extensions are located. See also the Manual Windows Installation Instructions. An example `extension_dir` value for PHP 5 is `c:\php\ext`.

> **Note**
>
> If when starting the web server an error similar to the following occurs: `"Unable to load dynamic library './php_mysqli.dll'"`, this is because `php_mysqli.dll` and/or `libmysql.dll` cannot be found by the system.

3.3.3 Runtime Configuration

The behaviour of these functions is affected by settings in `php.ini`.

Table 3.4 MySQLi Configuration Options

Name	Default	Changeable	Changelog
mysqli.allow_local_infile	"1"	PHP_INI_SYSTEM	Available since PHP 5.2.4.
mysqli.allow_persistent	"1"	PHP_INI_SYSTEM	Available since PHP 5.3.0.
mysqli.max_persistent	"-1"	PHP_INI_SYSTEM	Available since PHP 5.3.0.
mysqli.max_links	"-1"	PHP_INI_SYSTEM	Available since PHP 5.0.0.
mysqli.default_port	"3306"	PHP_INI_ALL	Available since PHP 5.0.0.
mysqli.default_socket	NULL	PHP_INI_ALL	Available since PHP 5.0.0.
mysqli.default_host	NULL	PHP_INI_ALL	Available since PHP 5.0.0.

Name	Default	Changeable	Changelog
mysqli.default_user	NULL	PHP_INI_ALL	Available since PHP 5.0.0.
mysqli.default_pw	NULL	PHP_INI_ALL	Available since PHP 5.0.0.
mysqli.reconnect	"0"	PHP_INI_SYSTEM	Available since PHP 4.3.5.
mysqli.rollback_on_cached_plink	TRUE	PHP_INI_SYSTEM	Available since PHP 5.6.0.
mysqli.cache_size	"2000"	PHP_INI_SYSTEM	Available since PHP 5.3.0.

For further details and definitions of the preceding PHP_INI_* constants, see the chapter on configuration changes.

Here's a short explanation of the configuration directives.

mysqli.allow_local_infile integer
: Allow accessing, from PHP's perspective, local files with LOAD DATA statements

mysqli.allow_persistent integer
: Enable the ability to create persistent connections using `mysqli_connect`.

mysqli.max_persistent integer
: Maximum of persistent connections that can be made. Set to 0 for unlimited.

mysqli.max_links integer
: The maximum number of MySQL connections per process.

mysqli.default_port integer
: The default TCP port number to use when connecting to the database server if no other port is specified. If no default is specified, the port will be obtained from the `MYSQL_TCP_PORT` environment variable, the `mysql-tcp` entry in `/etc/services` or the compile-time `MYSQL_PORT` constant, in that order. Win32 will only use the `MYSQL_PORT` constant.

mysqli.default_socket string
: The default socket name to use when connecting to a local database server if no other socket name is specified.

mysqli.default_host string
: The default server host to use when connecting to the database server if no other host is specified. Doesn't apply in safe mode.

mysqli.default_user string
: The default user name to use when connecting to the database server if no other name is specified. Doesn't apply in safe mode.

mysqli.default_pw string
: The default password to use when connecting to the database server if no other password is specified. Doesn't apply in safe mode.

mysqli.reconnect integer
: Automatically reconnect if the connection was lost.

> **Note**
>
> This `php.ini` setting is ignored by the mysqlnd driver.

mysqli.rollback_on_cached_plink bool
: Used for rollbacking connections put back into the persistent connection pool.

mysqli.cache_size integer Available only with mysqlnd.

Users cannot set MYSQL_OPT_READ_TIMEOUT through an API call or runtime configuration setting. Note that if it were possible there would be differences between how libmysqlclient and streams would interpret the value of MYSQL_OPT_READ_TIMEOUT.

3.3.4 Resource Types

This extension has no resource types defined.

3.4 The mysqli Extension and Persistent Connections

Persistent connection support was introduced in PHP 5.3 for the mysqli extension. Support was already present in PDO MYSQL and ext/mysql. The idea behind persistent connections is that a connection between a client process and a database can be reused by a client process, rather than being created and destroyed multiple times. This reduces the overhead of creating fresh connections every time one is required, as unused connections are cached and ready to be reused.

Unlike the mysql extension, mysqli does not provide a separate function for opening persistent connections. To open a persistent connection you must prepend p: to the hostname when connecting.

The problem with persistent connections is that they can be left in unpredictable states by clients. For example, a table lock might be activated before a client terminates unexpectedly. A new client process reusing this persistent connection will get the connection “as is”. Any cleanup would need to be done by the new client process before it could make good use of the persistent connection, increasing the burden on the programmer.

The persistent connection of the mysqli extension however provides built-in cleanup handling code. The cleanup carried out by mysqli includes:

- Rollback active transactions
- Close and drop temporary tables
- Unlock tables
- Reset session variables
- Close prepared statements (always happens with PHP)
- Close handler
- Release locks acquired with GET_LOCK

This ensures that persistent connections are in a clean state on return from the connection pool, before the client process uses them.

The mysqli extension does this cleanup by automatically calling the C-API function mysql_change_user().

The automatic cleanup feature has advantages and disadvantages though. The advantage is that the programmer no longer needs to worry about adding cleanup code, as it is called automatically. However,

the disadvantage is that the code could *potentially* be a little slower, as the code to perform the cleanup needs to run each time a connection is returned from the connection pool.

It is possible to switch off the automatic cleanup code, by compiling PHP with `MYSQLI_NO_CHANGE_USER_ON_PCONNECT` defined.

> **Note**
>
> The `mysqli` extension supports persistent connections when using either MySQL Native Driver or MySQL Client Library.

3.5 Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

`MYSQLI_READ_DEFAULT_GROUP`	Read options from the named group from `my.cnf` or the file specified with `MYSQLI_READ_DEFAULT_FILE`
`MYSQLI_READ_DEFAULT_FILE`	Read options from the named option file instead of from `my.cnf`
`MYSQLI_OPT_CONNECT_TIMEOUT`	Connect timeout in seconds
`MYSQLI_OPT_LOCAL_INFILE`	Enables command `LOAD LOCAL INFILE`
`MYSQLI_INIT_COMMAND`	Command to execute when connecting to MySQL server. Will automatically be re-executed when reconnecting.
`MYSQLI_CLIENT_SSL`	Use SSL (encrypted protocol). This option should not be set by application programs; it is set internally in the MySQL client library
`MYSQLI_CLIENT_COMPRESS`	Use compression protocol
`MYSQLI_CLIENT_INTERACTIVE`	Allow `interactive_timeout` seconds (instead of `wait_timeout` seconds) of inactivity before closing the connection. The client's session `wait_timeout` variable will be set to the value of the session `interactive_timeout` variable.
`MYSQLI_CLIENT_IGNORE_SPACE`	Allow spaces after function names. Makes all functions names reserved words.
`MYSQLI_CLIENT_NO_SCHEMA`	Don't allow the `db_name.tbl_name.col_name` syntax.
`MYSQLI_CLIENT_MULTI_QUERIES`	Allows multiple semicolon-delimited queries in a single `mysqli_query` call.
`MYSQLI_STORE_RESULT`	For using buffered resultsets
`MYSQLI_USE_RESULT`	For using unbuffered resultsets
`MYSQLI_ASSOC`	Columns are returned into the array having the fieldname as the array index.
`MYSQLI_NUM`	Columns are returned into the array having an enumerated index.
`MYSQLI_BOTH`	Columns are returned into the array having both a numerical index and the fieldname as the associative index.

MYSQLI_NOT_NULL_FLAG	Indicates that a field is defined as NOT NULL
MYSQLI_PRI_KEY_FLAG	Field is part of a primary index
MYSQLI_UNIQUE_KEY_FLAG	Field is part of a unique index.
MYSQLI_MULTIPLE_KEY_FLAG	Field is part of an index.
MYSQLI_BLOB_FLAG	Field is defined as BLOB
MYSQLI_UNSIGNED_FLAG	Field is defined as UNSIGNED
MYSQLI_ZEROFILL_FLAG	Field is defined as ZEROFILL
MYSQLI_AUTO_INCREMENT_FLAG	Field is defined as AUTO_INCREMENT
MYSQLI_TIMESTAMP_FLAG	Field is defined as TIMESTAMP
MYSQLI_SET_FLAG	Field is defined as SET
MYSQLI_NUM_FLAG	Field is defined as NUMERIC
MYSQLI_PART_KEY_FLAG	Field is part of an multi-index
MYSQLI_GROUP_FLAG	Field is part of GROUP BY
MYSQLI_TYPE_DECIMAL	Field is defined as DECIMAL
MYSQLI_TYPE_NEWDECIMAL	Precision math DECIMAL or NUMERIC field (MySQL 5.0.3 and up)
MYSQLI_TYPE_BIT	Field is defined as BIT (MySQL 5.0.3 and up)
MYSQLI_TYPE_TINY	Field is defined as TINYINT
MYSQLI_TYPE_SHORT	Field is defined as SMALLINT
MYSQLI_TYPE_LONG	Field is defined as INT
MYSQLI_TYPE_FLOAT	Field is defined as FLOAT
MYSQLI_TYPE_DOUBLE	Field is defined as DOUBLE
MYSQLI_TYPE_NULL	Field is defined as DEFAULT NULL
MYSQLI_TYPE_TIMESTAMP	Field is defined as TIMESTAMP
MYSQLI_TYPE_LONGLONG	Field is defined as BIGINT
MYSQLI_TYPE_INT24	Field is defined as MEDIUMINT
MYSQLI_TYPE_DATE	Field is defined as DATE
MYSQLI_TYPE_TIME	Field is defined as TIME
MYSQLI_TYPE_DATETIME	Field is defined as DATETIME
MYSQLI_TYPE_YEAR	Field is defined as YEAR
MYSQLI_TYPE_NEWDATE	Field is defined as DATE

MYSQLI_TYPE_INTERVAL	Field is defined as INTERVAL
MYSQLI_TYPE_ENUM	Field is defined as ENUM
MYSQLI_TYPE_SET	Field is defined as SET
MYSQLI_TYPE_TINY_BLOB	Field is defined as TINYBLOB
MYSQLI_TYPE_MEDIUM_BLOB	Field is defined as MEDIUMBLOB
MYSQLI_TYPE_LONG_BLOB	Field is defined as LONGBLOB
MYSQLI_TYPE_BLOB	Field is defined as BLOB
MYSQLI_TYPE_VAR_STRING	Field is defined as VARCHAR
MYSQLI_TYPE_STRING	Field is defined as CHAR or BINARY
MYSQLI_TYPE_CHAR	Field is defined as TINYINT. For CHAR, see MYSQLI_TYPE_STRING
MYSQLI_TYPE_GEOMETRY	Field is defined as GEOMETRY
MYSQLI_NEED_DATA	More data available for bind variable
MYSQLI_NO_DATA	No more data available for bind variable
MYSQLI_DATA_TRUNCATED	Data truncation occurred. Available since PHP 5.1.0 and MySQL 5.0.5.
MYSQLI_ENUM_FLAG	Field is defined as ENUM. Available since PHP 5.3.0.
MYSQLI_BINARY_FLAG	Field is defined as BINARY. Available since PHP 5.3.0.
MYSQLI_CURSOR_TYPE_FOR_UPDATE	
MYSQLI_CURSOR_TYPE_NO_CURSOR	
MYSQLI_CURSOR_TYPE_READ_ONLY	
MYSQLI_CURSOR_TYPE_SCROLLABLE	
MYSQLI_STMT_ATTR_CURSOR_TYPE	
MYSQLI_STMT_ATTR_PREFETCH_ROWS	
MYSQLI_STMT_ATTR_UPDATE_MAX_LENGTH	
MYSQLI_SET_CHARSET_NAME	
MYSQLI_REPORT_INDEX	Report if no index or bad index was used in a query.
MYSQLI_REPORT_ERROR	Report errors from mysqli function calls.
MYSQLI_REPORT_STRICT	Throw a mysqli_sql_exception for errors instead of warnings.
MYSQLI_REPORT_ALL	Set all options on (report all).
MYSQLI_REPORT_OFF	Turns reporting off.
MYSQLI_DEBUG_TRACE_ENABLED	Is set to 1 if mysqli_debug functionality is enabled.

MYSQLI_SERVER_QUERY_NO_GOOD_INDEX_USED	
MYSQLI_SERVER_QUERY_NO_INDEX_USED	
MYSQLI_REFRESH_GRANT	Refreshes the grant tables.
MYSQLI_REFRESH_LOG	Flushes the logs, like executing the FLUSH LOGS SQL statement.
MYSQLI_REFRESH_TABLES	Flushes the table cache, like executing the FLUSH TABLES SQL statement.
MYSQLI_REFRESH_HOSTS	Flushes the host cache, like executing the FLUSH HOSTS SQL statement.
MYSQLI_REFRESH_STATUS	Reset the status variables, like executing the FLUSH STATUS SQL statement.
MYSQLI_REFRESH_THREADS	Flushes the thread cache.
MYSQLI_REFRESH_SLAVE	On a slave replication server: resets the master server information, and restarts the slave. Like executing the RESET SLAVE SQL statement.
MYSQLI_REFRESH_MASTER	On a master replication server: removes the binary log files listed in the binary log index, and truncates the index file. Like executing the RESET MASTER SQL statement.
MYSQLI_TRANS_COR_AND_CHAIN	Appends "AND CHAIN" to mysqli_commit or mysqli_rollback.
MYSQLI_TRANS_COR_AND_NO_CHAIN	Appends "AND NO CHAIN" to mysqli_commit or mysqli_rollback.
MYSQLI_TRANS_COR_RELEASE	Appends "RELEASE" to mysqli_commit or mysqli_rollback.
MYSQLI_TRANS_COR_NO_RELEASE	Appends "NO RELEASE" to mysqli_commit or mysqli_rollback.
MYSQLI_TRANS_START_READ_ONLY	Start the transaction as "START TRANSACTION READ ONLY" with mysqli_begin_transaction.
MYSQLI_TRANS_START_READ_WRITE	Start the transaction as "START TRANSACTION READ WRITE" with mysqli_begin_transaction.
MYSQLI_TRANS_START_CONSISTENT_SNAPSHOT	Start the transaction as "START TRANSACTION WITH CONSISTENT SNAPSHOT" with mysqli_begin_transaction.

3.6 Notes

Some implementation notes:

1. Support was added for MYSQL_TYPE_GEOMETRY to the MySQLi extension in PHP 5.3.
2. Note there are different internal implementations within libmysqlclient and mysqlnd for handling columns of type MYSQL_TYPE_GEOMETRY. Generally speaking, mysqlnd will allocate significantly less memory. For example, if there is a POINT column in a result set, libmysqlclient may pre-allocate up to 4GB of RAM although less than 50 bytes are needed for holding a POINT column in memory. Memory allocation is much lower, less than 50 bytes, if using mysqlnd.

3.7 The MySQLi Extension Function Summary

Table 3.5 Summary of `mysqli` methods

mysqli Class			
OOP Interface	**Procedural Interface**	**Alias (Do not use)**	**Description**
Properties			
$mysqli::affected_rows	`mysqli_affected_rows`	N/A	Gets the number of affected rows in a previous MySQL operation
$mysqli::client_info	`mysqli_get_client_info`	N/A	Returns the MySQL client version as a string
$mysqli::client_version	`mysqli_get_client_version`	N/A	Returns MySQL client version info as an integer
$mysqli::connect_errno	`mysqli_connect_errno`	N/A	Returns the error code from last connect call
$mysqli::connect_error	`mysqli_connect_error`	N/A	Returns a string description of the last connect error
$mysqli::errno	`mysqli_errno`	N/A	Returns the error code for the most recent function call
$mysqli::error	`mysqli_error`	N/A	Returns a string description of the last error
$mysqli::field_count	`mysqli_field_count`	N/A	Returns the number of columns for the most recent query
$mysqli::host_info	`mysqli_get_host_info`	N/A	Returns a string representing the type of connection used
$mysqli::protocol_version	`mysqli_get_proto_info`	N/A	Returns the version of the MySQL protocol used
$mysqli::server_info	`mysqli_get_server_info`	N/A	Returns the version of the MySQL server
$mysqli::server_version	`mysqli_get_server_version`	N/A	Returns the version of the MySQL server as an integer
$mysqli::info	`mysqli_info`	N/A	Retrieves information about the most recently executed query
$mysqli::insert_id	`mysqli_insert_id`	N/A	Returns the auto generated id used in the last query

mysqli Class			
OOP Interface	**Procedural Interface**	**Alias (Do not use)**	**Description**
$mysqli::sqlstate	mysqli_sqlstate	N/A	Returns the SQLSTATE error from previous MySQL operation
$mysqli::warning_count	mysqli_warning_count	N/A	Returns the number of warnings from the last query for the given link
Methods			
mysqli::autocommit	mysqli_autocommit	N/A	Turns on or off auto-committing database modifications
mysqli::change_user	mysqli_change_user	N/A	Changes the user of the specified database connection
mysqli::character_set_name, mysqli::client_encoding	mysqli_character_set_name	mysqli_client_encoding	Returns the default character set for the database connection
mysqli::close	mysqli_close	N/A	Closes a previously opened database connection
mysqli::commit	mysqli_commit	N/A	Commits the current transaction
mysqli::__construct	mysqli_connect	N/A	Open a new connection to the MySQL server [Note: static (i.e. class) method]
mysqli::debug	mysqli_debug	N/A	Performs debugging operations
mysqli::dump_debug_info	mysqli_dump_debug_info	N/A	Dump debugging information into the log
mysqli::get_charset	mysqli_get_charset	N/A	Returns a character set object
mysqli::get_connection_stats	mysqli_get_connection_stats	N/A	Returns client connection statistics. Available only with mysqlnd.
mysqli::get_client_info	mysqli_get_client_info	N/A	Returns the MySQL client version as a string
mysqli::get_client_stats	mysqli_get_client_stats	N/A	Returns client per-process statistics. Available only with mysqlnd.
mysqli::get_cache_stats	mysqli_get_cache_stats	N/A	Returns client Zval cache statistics. Available only with mysqlnd.
mysqli::get_server_info	mysqli_get_server_info	N/A	Returns a string representing the version

mysqli Class			
OOP Interface	**Procedural Interface**	**Alias (Do not use)**	**Description**
			of the MySQL server that the MySQLi extension is connected to
mysqli::get_warnings	mysqli_get_warnings	N/A	NOT DOCUMENTED
mysqli::init	mysqli_init	N/A	Initializes MySQLi and returns a resource for use with mysqli_real_connect. [Not called on an object, as it returns a $mysqli object.]
mysqli::kill	mysqli_kill	N/A	Asks the server to kill a MySQL thread
mysqli::more_results	mysqli_more_results	N/A	Check if there are any more query results from a multi query
mysqli::multi_query	mysqli_multi_query	N/A	Performs a query on the database
mysqli::next_result	mysqli_next_result	N/A	Prepare next result from multi_query
mysqli::options	mysqli_options	mysqli_set_opt	Set options
mysqli::ping	mysqli_ping	N/A	Pings a server connection, or tries to reconnect if the connection has gone down
mysqli::prepare	mysqli_prepare	N/A	Prepare an SQL statement for execution
mysqli::query	mysqli_query	N/A	Performs a query on the database
mysqli::real_connect	mysqli_real_connect	N/A	Opens a connection to a mysql server
mysqli::real_escape_string, mysqli::escape_string	mysqli_real_escape_string	mysqli_escape_string	Escapes special characters in a string for use in an SQL statement, taking into account the current charset of the connection
mysqli::real_query	mysqli_real_query	N/A	Execute an SQL query
mysqli::refresh	mysqli_refresh	N/A	Flushes tables or caches, or resets the replication server information
mysqli::rollback	mysqli_rollback	N/A	Rolls back current transaction

mysqli Class			
OOP Interface	**Procedural Interface**	**Alias (Do not use)**	**Description**
mysqli::select_db	mysqli_select_db	N/A	Selects the default database for database queries
mysqli::set_charset	mysqli_set_charset	N/A	Sets the default client character set
mysqli::set_local_infile_default	mysqli_set_local_infile_default	N/A	Unsets user defined handler for load local infile command
mysqli::set_local_infile_handler	mysqli_set_local_infile_handler	N/A	Set callback function for LOAD DATA LOCAL INFILE command
mysqli::ssl_set	mysqli_ssl_set	N/A	Used for establishing secure connections using SSL
mysqli::stat	mysqli_stat	N/A	Gets the current system status
mysqli::stmt_init	mysqli_stmt_init	N/A	Initializes a statement and returns an object for use with mysqli_stmt_prepare
mysqli::store_result	mysqli_store_result	N/A	Transfers a result set from the last query
mysqli::thread_id	mysqli_thread_id	N/A	Returns the thread ID for the current connection
mysqli::thread_safe	mysqli_thread_safe	N/A	Returns whether thread safety is given or not
mysqli::use_result	mysqli_use_result	N/A	Initiate a result set retrieval

Table 3.6 Summary of `mysqli_stmt` methods

MySQL_STMT			
OOP Interface	**Procedural Interface**	**Alias (Do not use)**	**Description**
Properties			
$mysqli_stmt::affected_rows	mysqli_stmt_affected_rows	N/A	Returns the total number of rows changed, deleted, or inserted by the last executed statement
$mysqli_stmt::errno	mysqli_stmt_errno	N/A	Returns the error code for the most recent statement call
$mysqli_stmt::error	mysqli_stmt_error	N/A	Returns a string description for last statement error

MySQL_STMT			
OOP Interface	**Procedural Interface**	**Alias (Do not use)**	**Description**
$mysqli_stmt::field_count	mysqli_stmt_field_count	N/A	Returns the number of field in the given statement - not documented
$mysqli_stmt::insert_id	mysqli_stmt_insert_id	N/A	Get the ID generated from the previous INSERT operation
$mysqli_stmt::num_rows	mysqli_stmt_num_rows	N/A	Return the number of rows in statements result set
$mysqli_stmt::param_count	mysqli_stmt_param_count	mysqli_param_count	Returns the number of parameter for the given statement
$mysqli_stmt::sqlstate	mysqli_stmt_sqlstate	N/A	Returns SQLSTATE error from previous statement operation
Methods			
mysqli_stmt::attr_get	mysqli_stmt_attr_get	N/A	Used to get the current value of a statement attribute
mysqli_stmt::attr_set	mysqli_stmt_attr_set	N/A	Used to modify the behavior of a prepared statement
mysqli_stmt::bind_param	mysqli_stmt_bind_param	mysqli_bind_param	Binds variables to a prepared statement as parameters
mysqli_stmt::bind_result	mysqli_stmt_bind_result	mysqli_bind_result	Binds variables to a prepared statement for result storage
mysqli_stmt::close	mysqli_stmt_close	N/A	Closes a prepared statement
mysqli_stmt::data_seek	mysqli_stmt_data_seek	N/A	Seeks to an arbitrary row in statement result set
mysqli_stmt::execute	mysqli_stmt_execute	mysqli_execute	Executes a prepared Query
mysqli_stmt::fetch	mysqli_stmt_fetch	mysqli_fetch	Fetch results from a prepared statement into the bound variables
mysqli_stmt::free_result	mysqli_stmt_free_result	N/A	Frees stored result memory for the given statement handle
mysqli_stmt::get_result	mysqli_stmt_get_result	N/A	Gets a result set from a prepared statement. Available only with mysqlnd.

MySQL_STMT			
OOP Interface	**Procedural Interface**	**Alias (Do not use)**	**Description**
mysqli_stmt::get_warnings	mysqli_stmt_get_warnings	N/A	NOT DOCUMENTED
$mysqli_stmt::more_results	mysqli_stmt_more_results	N/A	Checks if there are more query results from a multiple query
$mysqli_stmt::next_result	mysqli_stmt_next_result	N/A	Reads the next result from a multiple query
mysqli_stmt::num_rows	mysqli_stmt_num_rows	N/A	See also property $mysqli_stmt::num_rows
mysqli_stmt::prepare	mysqli_stmt_prepare	N/A	Prepare an SQL statement for execution
mysqli_stmt::reset	mysqli_stmt_reset	N/A	Resets a prepared statement
mysqli_stmt::result_metadata	mysqli_stmt_result_metadata	mysqli_get_metadata	Returns result set metadata from a prepared statement
mysqli_stmt::send_long_data	mysqli_stmt_send_long_data	mysqli_send_long_data	Send data in blocks
mysqli_stmt::store_result	mysqli_stmt_store_result	N/A	Transfers a result set from a prepared statement

Table 3.7 Summary of `mysqli_result` methods

mysqli_result			
OOP Interface	**Procedural Interface**	**Alias (Do not use)**	**Description**
Properties			
$mysqli_result::current_field	mysqli_field_tell	N/A	Get current field offset of a result pointer
$mysqli_result::field_count	mysqli_num_fields	N/A	Get the number of fields in a result
$mysqli_result::lengths	mysqli_fetch_lengths	N/A	Returns the lengths of the columns of the current row in the result set
$mysqli_result::num_rows	mysqli_num_rows	N/A	Gets the number of rows in a result
Methods			
mysqli_result::data_seek	mysqli_data_seek	N/A	Adjusts the result pointer to an arbitrary row in the result
mysqli_result::fetch_all	mysqli_fetch_all	N/A	Fetches all result rows and returns the result set as an associative array, a numeric array, or both. Available only with mysqlnd.

mysqli_result			
OOP Interface	**Procedural Interface**	**Alias (Do not use)**	**Description**
mysqli_result::fetch_array	mysqli_fetch_array	N/A	Fetch a result row as an associative, a numeric array, or both
mysqli_result::fetch_assoc	mysqli_fetch_assoc	N/A	Fetch a result row as an associative array
mysqli_result::fetch_field_direct	mysqli_fetch_field_direct	N/A	Fetch meta-data for a single field
mysqli_result::fetch_field	mysqli_fetch_field	N/A	Returns the next field in the result set
mysqli_result::fetch_fields	mysqli_fetch_fields	N/A	Returns an array of objects representing the fields in a result set
mysqli_result::fetch_object	mysqli_fetch_object	N/A	Returns the current row of a result set as an object
mysqli_result::fetch_row	mysqli_fetch_row	N/A	Get a result row as an enumerated array
mysqli_result::field_seek	mysqli_field_seek	N/A	Set result pointer to a specified field offset
mysqli_result::free, mysqli_result::close, mysqli_result::free_result	mysqli_free_result	N/A	Frees the memory associated with a result

Table 3.8 Summary of `mysqli_driver` methods

MySQL_Driver			
OOP Interface	**Procedural Interface**	**Alias (Do not use)**	**Description**
Properties			
N/A			
Methods			
mysqli_driver::embedded_server_end	mysqli_embedded_server_end	N/A	NOT DOCUMENTED
mysqli_driver::embedded_server_start	mysqli_embedded_server_start	N/A	NOT DOCUMENTED

> **Note**
>
> Alias functions are provided for backward compatibility purposes only. Do not use them in new projects.

3.8 Examples

3.8.1 MySQLi extension basic examples

This example shows how to connect, execute a query, use basic error handling, print resulting rows, and disconnect from a MySQL database.

This example uses the freely available Sakila database that can be downloaded from dev.mysql.com, as described here. To get this example to work, (a) install sakila and (b) modify the connection variables (host, your_user, your_pass).

Example 3.30 MySQLi extension overview example

```
<?php
// Let's pass in a $_GET variable to our example, in this case
// it's aid for actor_id in our Sakila database. Let's make it
// default to 1, and cast it to an integer as to avoid SQL injection
// and/or related security problems. Handling all of this goes beyond
// the scope of this simple example. Example:
//   http://example.org/script.php?aid=42
if (isset($_GET['aid']) && is_numeric($_GET['aid'])) {
    $aid = (int) $_GET['aid'];
} else {
    $aid = 1;
}

// Connecting to and selecting a MySQL database named sakila
// Hostname: 127.0.0.1, username: your_user, password: your_pass, db: sakila
$mysqli = new mysqli('127.0.0.1', 'your_user', 'your_pass', 'sakila');

// Oh no! A connect_errno exists so the connection attempt failed!
if ($mysqli->connect_errno) {
    // The connection failed. What do you want to do?
    // You could contact yourself (email?), log the error, show a nice page, etc.
    // You do not want to reveal sensitive information

    // Let's try this:
    echo "Sorry, this website is experiencing problems.";

    // Something you should not do on a public site, but this example will show you
    // anyways, is print out MySQL error related information -- you might log this
    echo "Error: Failed to make a MySQL connection, here is why: \n";
    echo "Errno: " . $mysqli->connect_errno . "\n";
    echo "Error: " . $mysqli->connect_error . "\n";

    // You might want to show them something nice, but we will simply exit
    exit;
}

// Perform an SQL query
$sql = "SELECT actor_id, first_name, last_name FROM actor WHERE actor_id = $aid";
if (!$result = $mysqli->query($sql)) {
    // Oh no! The query failed.
    echo "Sorry, the website is experiencing problems.";

    // Again, do not do this on a public site, but we'll show you how
    // to get the error information
    echo "Error: Our query failed to execute and here is why: \n";
    echo "Query: " . $sql . "\n";
    echo "Errno: " . $mysqli->errno . "\n";
    echo "Error: " . $mysqli->error . "\n";
    exit;
}

// Phew, we made it. We know our MySQL connection and query
// succeeded, but do we have a result?
if ($result->num_rows === 0) {
```

```
    // Oh, no rows! Sometimes that's expected and okay, sometimes
    // it is not. You decide. In this case, maybe actor_id was too
    // large?
    echo "We could not find a match for ID $aid, sorry about that. Please try again.";
    exit;
}

// Now, we know only one result will exist in this example so let's
// fetch it into an associated array where the array's keys are the
// table's column names
$actor = $result->fetch_assoc();
echo "Sometimes I see " . $actor['first_name'] . " " . $actor['last_name'] . " on TV.";

// Now, let's fetch five random actors and output their names to a list.
// We'll add less error handling here as you can do that on your own now
$sql = "SELECT actor_id, first_name, last_name FROM actor ORDER BY rand() LIMIT 5";
if (!$result = $mysqli->query($sql)) {
    echo "Sorry, the website is experiencing problems.";
    exit;
}

// Print our 5 random actors in a list, and link to each actor
echo "<ul>\n";
while ($actor = $result->fetch_assoc()) {
    echo "<li><a href='" . $_SERVER['SCRIPT_FILENAME'] . "?aid=" . $actor['actor_id'] . "'>\n";
    echo $actor['first_name'] . ' ' . $actor['last_name'];
    echo "</a></li>\n";
}
echo "</ul>\n";

// The script will automatically free the result and close the MySQL
// connection when it exits, but let's just do it anyways
$result->free();
$mysqli->close();
?>
```

3.9 The mysqli class

Represents a connection between PHP and a MySQL database.

```
mysqli {
mysqli

      Properties

  int
    mysqli->affected_rows ;

  string
    mysqli->client_info ;

  int
    mysqli->client_version ;

  int
    mysqli->connect_errno ;

  string
    mysqli->connect_error ;
```

```
  int
    mysqli->errno ;

  array
    mysqli->error_list ;

  string
    mysqli->error ;

  int
    mysqli->field_count ;

  int
    mysqli->client_version ;

  string
    mysqli->host_info ;

  string
    mysqli->protocol_version ;

  string
    mysqli->server_info ;

  int
    mysqli->server_version ;

  string
    mysqli->info ;

  mixed
    mysqli->insert_id ;

  string
    mysqli->sqlstate ;

  int
    mysqli->thread_id ;

  int
    mysqli->warning_count ;

Methods

  mysqli::__construct(
    string host
        =  =ini_get("mysqli.default_host"),
    string username
        =  =ini_get("mysqli.default_user"),
    string passwd
        =  =ini_get("mysqli.default_pw"),
    string dbname
        =  ="",
    int port
        =  =ini_get("mysqli.default_port"),
    string socket
        =  =ini_get("mysqli.default_socket"));

  bool mysqli::autocommit(
    bool mode);

  bool mysqli::change_user(
    string user,
    string password,
    string database);
```

```
string mysqli::character_set_name();

bool mysqli::close();

bool mysqli::commit(
  int flags,
  string name);

bool mysqli::debug(
  string message);

bool mysqli::dump_debug_info();

object mysqli::get_charset();

string mysqli::get_client_info();

bool mysqli::get_connection_stats();

mysqli_warning mysqli::get_warnings();

mysqli mysqli::init();

bool mysqli::kill(
  int processid);

bool mysqli::more_results();

bool mysqli::multi_query(
  string query);

bool mysqli::next_result();

bool mysqli::options(
  int option,
  mixed value);

bool mysqli::ping();

public static int mysqli::poll(
  array read,
  array error,
  array reject,
  int sec,
  int usec);

mysqli_stmt mysqli::prepare(
  string query);

mixed mysqli::query(
  string query,
  int resultmode
      =  =MYSQLI_STORE_RESULT);

bool mysqli::real_connect(
  string host,
  string username,
  string passwd,
  string dbname,
  int port,
  string socket,
  int flags);

string mysqli::escape_string(
```

```
    string escapestr);

  bool mysqli::real_query(
    string query);

  public mysqli_result mysqli::reap_async_query();

  public bool mysqli::refresh(
    int options);

  bool mysqli::rollback(
    int flags,
    string name);

  int mysqli::rpl_query_type(
    string query);

  bool mysqli::select_db(
    string dbname);

  bool mysqli::send_query(
    string query);

  bool mysqli::set_charset(
    string charset);

  bool mysqli::set_local_infile_handler(
    mysqli link,
    callable read_func);

  bool mysqli::ssl_set(
    string key,
    string cert,
    string ca,
    string capath,
    string cipher);

  string mysqli::stat();

  mysqli_stmt mysqli::stmt_init();

  mysqli_result mysqli::store_result(
    int option);

  mysqli_result mysqli::use_result();

}
```

3.9.1 mysqli::$affected_rows, mysqli_affected_rows

- mysqli::$affected_rows

  mysqli_affected_rows

  Gets the number of affected rows in a previous MySQL operation

Description

Object oriented style

```
  int
```

```
    mysqli->affected_rows ;
```

Procedural style

```
  int mysqli_affected_rows(
    mysqli link);
```

Returns the number of rows affected by the last INSERT, UPDATE, REPLACE or DELETE query.

For SELECT statements mysqli_affected_rows works like mysqli_num_rows.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

An integer greater than zero indicates the number of rows affected or retrieved. Zero indicates that no records were updated for an UPDATE statement, no rows matched the WHERE clause in the query or that no query has yet been executed. -1 indicates that the query returned an error.

> **Note**
>
> If the number of affected rows is greater than the maximum integer value(PHP_INT_MAX), the number of affected rows will be returned as a string.

Examples

Example 3.31 $mysqli->affected_rows example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* Insert rows */
$mysqli->query("CREATE TABLE Language SELECT * from CountryLanguage");
printf("Affected rows (INSERT): %d\n", $mysqli->affected_rows);

$mysqli->query("ALTER TABLE Language ADD Status int default 0");

/* update rows */
$mysqli->query("UPDATE Language SET Status=1 WHERE Percentage > 50");
printf("Affected rows (UPDATE): %d\n", $mysqli->affected_rows);

/* delete rows */
$mysqli->query("DELETE FROM Language WHERE Percentage < 50");
printf("Affected rows (DELETE): %d\n", $mysqli->affected_rows);

/* select all rows */
$result = $mysqli->query("SELECT CountryCode FROM Language");
printf("Affected rows (SELECT): %d\n", $mysqli->affected_rows);
```

```
$result->close();

/* Delete table Language */
$mysqli->query("DROP TABLE Language");

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

if (!$link) {
    printf("Can't connect to localhost. Error: %s\n", mysqli_connect_error());
    exit();
}

/* Insert rows */
mysqli_query($link, "CREATE TABLE Language SELECT * from CountryLanguage");
printf("Affected rows (INSERT): %d\n", mysqli_affected_rows($link));

mysqli_query($link, "ALTER TABLE Language ADD Status int default 0");

/* update rows */
mysqli_query($link, "UPDATE Language SET Status=1 WHERE Percentage > 50");
printf("Affected rows (UPDATE): %d\n", mysqli_affected_rows($link));

/* delete rows */
mysqli_query($link, "DELETE FROM Language WHERE Percentage < 50");
printf("Affected rows (DELETE): %d\n", mysqli_affected_rows($link));

/* select all rows */
$result = mysqli_query($link, "SELECT CountryCode FROM Language");
printf("Affected rows (SELECT): %d\n", mysqli_affected_rows($link));

mysqli_free_result($result);

/* Delete table Language */
mysqli_query($link, "DROP TABLE Language");

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Affected rows (INSERT): 984
Affected rows (UPDATE): 168
Affected rows (DELETE): 815
Affected rows (SELECT): 169
```

See Also

mysqli_num_rows

mysqli_info

3.9.2 mysqli::autocommit, mysqli_autocommit

- mysqli::autocommit

  mysqli_autocommit

  Turns on or off auto-committing database modifications

Description

Object oriented style

```
bool mysqli::autocommit(
  bool mode);
```

Procedural style

```
bool mysqli_autocommit(
  mysqli link,
  bool mode);
```

Turns on or off auto-commit mode on queries for the database connection.

To determine the current state of autocommit use the SQL command SELECT @@autocommit.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
mode	Whether to turn on auto-commit or not.

Return Values

Returns TRUE on success or FALSE on failure.

Notes

> **Note**
>
> This function doesn't work with non transactional table types (like MyISAM or ISAM).

Examples

Example 3.32 mysqli::autocommit example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");
```

```
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* turn autocommit on */
$mysqli->autocommit(TRUE);

if ($result = $mysqli->query("SELECT @@autocommit")) {
    $row = $result->fetch_row();
    printf("Autocommit is %s\n", $row[0]);
    $result->free();
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```


<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

if (!$link) {
    printf("Can't connect to localhost. Error: %s\n", mysqli_connect_error());
    exit();
}

/* turn autocommit on */
mysqli_autocommit($link, TRUE);

if ($result = mysqli_query($link, "SELECT @@autocommit")) {
    $row = mysqli_fetch_row($result);
    printf("Autocommit is %s\n", $row[0]);
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```


Autocommit is 1
```

See Also

mysqli_begin_transaction
mysqli_commit
mysqli_rollback

3.9.3 mysqli::begin_transaction, mysqli_begin_transaction

- mysqli::begin_transaction

  mysqli_begin_transaction

  Starts a transaction

Description

Object oriented style (method):

```
public bool mysqli::begin_transaction(
  int flags,
  string name);
```

Procedural style:

```
bool mysqli_begin_transaction(
  mysqli link,
  int flags,
  string name);
```

Begins a transaction. Requires MySQL 5.6 and above, and the InnoDB engine (it is enabled by default). For additional details about how MySQL transactions work, see http://dev.mysql.com/doc/mysql/en/commit.html.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
flags	Valid flags are: • MYSQLI_TRANS_START_READ_ONLY: Start the transaction as "START TRANSACTION READ ONLY". • MYSQLI_TRANS_START_READ_WRITE: Start the transaction as "START TRANSACTION READ WRITE". • MYSQLI_TRANS_START_WITH_CONSISTENT_SNAPSHOT: Start the transaction as "START TRANSACTION WITH CONSISTENT SNAPSHOT".
name	Savepoint name for the transaction.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.33 $mysqli->begin_transaction example

Object oriented style

```
<?php
$mysqli = new mysqli("127.0.0.1", "my_user", "my_password", "sakila");
```

```
if ($mysqli->connect_errno) {
    printf("Connect failed: %s\n", $mysqli->connect_error);
    exit();
}

$mysqli->begin_transaction(MYSQLI_TRANS_START_READ_ONLY);

$mysqli->query("SELECT first_name, last_name FROM actor");
$mysqli->commit();

$mysqli->close();
?>
```

Procedural style

```


<?php
$link = mysqli_connect("127.0.0.1", "my_user", "my_password", "sakila");

if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

mysqli_begin_transaction($link, MYSQLI_TRANS_START_READ_ONLY);

mysqli_query($link, "SELECT first_name, last_name FROM actor LIMIT 1");
mysqli_commit($link);

mysqli_close($link);
?>
```

See Also

mysqli_autocommit
mysqli_commit
mysqli_rollback

3.9.4 mysqli::change_user, mysqli_change_user

- mysqli::change_user

  mysqli_change_user

  Changes the user of the specified database connection

Description

Object oriented style

```
  bool mysqli::change_user(
    string user,
    string password,
    string database);
```

Procedural style

```
bool mysqli_change_user(
  mysqli link,
  string user,
  string password,
  string database);
```

Changes the user of the specified database connection and sets the current database.

In order to successfully change users a valid *username* and *password* parameters must be provided and that user must have sufficient permissions to access the desired database. If for any reason authorization fails, the current user authentication will remain.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
user	The MySQL user name.
password	The MySQL password.
database	The database to change to. If desired, the NULL value may be passed resulting in only changing the user and not selecting a database. To select a database in this case use the mysqli_select_db function.

Return Values

Returns TRUE on success or FALSE on failure.

Notes

> **Note**
>
> Using this command will always cause the current database connection to behave as if was a completely new database connection, regardless of if the operation was completed successfully. This reset includes performing a rollback on any active transactions, closing all temporary tables, and unlocking all locked tables.

Examples

Example 3.34 mysqli::change_user example

Object oriented style

```
<?php

/* connect database test */
$mysqli = new mysqli("localhost", "my_user", "my_password", "test");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
```

```
    exit();
}

/* Set Variable a */
$mysqli->query("SET @a:=1");

/* reset all and select a new database */
$mysqli->change_user("my_user", "my_password", "world");

if ($result = $mysqli->query("SELECT DATABASE()")) {
    $row = $result->fetch_row();
    printf("Default database: %s\n", $row[0]);
    $result->close();
}

if ($result = $mysqli->query("SELECT @a")) {
    $row = $result->fetch_row();
    if ($row[0] === NULL) {
        printf("Value of variable a is NULL\n");
    }
    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
/* connect database test */
$link = mysqli_connect("localhost", "my_user", "my_password", "test");

/* check connection */
if (!$link) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* Set Variable a */
mysqli_query($link, "SET @a:=1");

/* reset all and select a new database */
mysqli_change_user($link, "my_user", "my_password", "world");

if ($result = mysqli_query($link, "SELECT DATABASE()")) {
    $row = mysqli_fetch_row($result);
    printf("Default database: %s\n", $row[0]);
    mysqli_free_result($result);
}

if ($result = mysqli_query($link, "SELECT @a")) {
    $row = mysqli_fetch_row($result);
    if ($row[0] === NULL) {
        printf("Value of variable a is NULL\n");
    }
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

```
```

The above examples will output:

```
Default database: world
Value of variable a is NULL
```

See Also

mysqli_connect
mysqli_select_db

3.9.5 mysqli::character_set_name, mysqli_character_set_name

- mysqli::character_set_name

  mysqli_character_set_name

  Returns the default character set for the database connection

Description

Object oriented style

```
string mysqli::character_set_name();
```

Procedural style

```
string mysqli_character_set_name(
  mysqli link);
```

Returns the current character set for the database connection.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

The default character set for the current connection

Examples

Example 3.35 mysqli::character_set_name example

Object oriented style

```
<?php
/* Open a connection */
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");
```

```
/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* Print current character set */
$charset = $mysqli->character_set_name();
printf ("Current character set is %s\n", $charset);

$mysqli->close();
?>
```

Procedural style

```
<?php
/* Open a connection */
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (!$link) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* Print current character set */
$charset = mysqli_character_set_name($link);
printf ("Current character set is %s\n",$charset);

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Current character set is latin1_swedish_ci
```

See Also

mysqli_set_charset
mysqli_client_encoding
mysqli_real_escape_string

3.9.6 mysqli::$client_info, mysqli_get_client_info

- mysqli::$client_info

  mysqli_get_client_info

  Get MySQL client info

Description

Object oriented style

```
  string
    mysqli->client_info ;
```

Procedural style

```
  string mysqli_get_client_info(
    mysqli link);
```

Returns a string that represents the MySQL client library version.

Return Values

A string that represents the MySQL client library version

Examples

Example 3.36 mysqli_get_client_info

```
<?php

/* We don't need a connection to determine
   the version of mysql client library */

printf("Client library version: %s\n", mysqli_get_client_info());
?>
```

See Also

mysqli_get_client_version
mysqli_get_server_info
mysqli_get_server_version

3.9.7 mysqli::$client_version, mysqli_get_client_version

- mysqli::$client_version

  mysqli_get_client_version

  Returns the MySQL client version as a number

Description

Object oriented style

```
  int
    mysqli->client_version ;
```

Procedural style

```
  int mysqli_get_client_version(
```

```
    mysqli link);
```

Returns client version number as an integer.

Return Values

A number that represents the MySQL client library version in format: `main_version*10000 + minor_version *100 + sub_version`. For example, 4.1.0 is returned as 40100.

This is useful to quickly determine the version of the client library to know if some capability exists.

Examples

Example 3.37 mysqli_get_client_version

```
<?php

/* We don't need a connection to determine
   the version of mysql client library */

printf("Client library version: %d\n", mysqli_get_client_version());
?>
```

See Also

`mysqli_get_client_info`
`mysqli_get_server_info`
`mysqli_get_server_version`

3.9.8 `mysqli::close`, `mysqli_close`

- `mysqli::close`

  `mysqli_close`

  Closes a previously opened database connection

Description

Object oriented style

```
  bool mysqli::close();
```

Procedural style

```
  bool mysqli_close(
    mysqli link);
```

Closes a previously opened database connection.

Open non-persistent MySQL connections and result sets are automatically destroyed when a PHP script finishes its execution. So, while explicitly closing open connections and freeing result sets is optional,

doing so is recommended. This will immediately return resources to PHP and MySQL, which can improve performance. For related information, see freeing resources

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

Returns TRUE on success or FALSE on failure.

Examples

See mysqli_connect.

Notes

> **Note**
>
> mysqli_close will not close persistent connections. For additional details, see the manual page on persistent connections.

See Also

mysqli::__construct
mysqli_init
mysqli_real_connect
mysqli_free_result

3.9.9 mysqli::commit, mysqli_commit

- mysqli::commit

  mysqli_commit

  Commits the current transaction

Description

Object oriented style

```
bool mysqli::commit(
  int flags,
  string name);
```

Procedural style

```
bool mysqli_commit(
  mysqli link,
  int flags,
  string name);
```

Commits the current transaction for the database connection.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
flags	A bitmask of MYSQLI_TRANS_COR_* constants.
name	If provided then COMMIT/*name*/ is executed.

Return Values

Returns TRUE on success or FALSE on failure.

Changelog

Version	Description
5.5.0	Added *flags* and *name* parameters.

Examples

Example 3.38 mysqli::commit example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$mysqli->query("CREATE TABLE Language LIKE CountryLanguage");

/* set autocommit to off */
$mysqli->autocommit(FALSE);

/* Insert some values */
$mysqli->query("INSERT INTO Language VALUES ('DEU', 'Bavarian', 'F', 11.2)");
$mysqli->query("INSERT INTO Language VALUES ('DEU', 'Swabian', 'F', 9.4)");

/* commit transaction */
if (!$mysqli->commit()) {
    print("Transaction commit failed\n");
    exit();
}

/* drop table */
$mysqli->query("DROP TABLE Language");

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "test");
```

```
/* check connection */
if (!$link) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* set autocommit to off */
mysqli_autocommit($link, FALSE);

mysqli_query($link, "CREATE TABLE Language LIKE CountryLanguage");

/* Insert some values */
mysqli_query($link, "INSERT INTO Language VALUES ('DEU', 'Bavarian', 'F', 11.2)");
mysqli_query($link, "INSERT INTO Language VALUES ('DEU', 'Swabian', 'F', 9.4)");

/* commit transaction */
if (!mysqli_commit($link)) {
    print("Transaction commit failed\n");
    exit();
}

/* close connection */
mysqli_close($link);
?>
```

See Also

mysqli_autocommit
mysqli_begin_transaction
mysqli_rollback
mysqli_savepoint

3.9.10 mysqli::$connect_errno, mysqli_connect_errno

- mysqli::$connect_errno

  mysqli_connect_errno

  Returns the error code from last connect call

Description

Object oriented style

```
int
  mysqli->connect_errno ;
```

Procedural style

```
int mysqli_connect_errno();
```

Returns the last error code number from the last call to mysqli_connect.

> **Note**
>
> Client error message numbers are listed in the MySQL errmsg.h header file, server error message numbers are listed in mysqld_error.h. In the MySQL

> source distribution you can find a complete list of error messages and error numbers in the file Docs/mysqld_error.txt.

Return Values

An error code value for the last call to mysqli_connect, if it failed. zero means no error occurred.

Examples

Example 3.39 $mysqli->connect_errno example

Object oriented style

```
<?php
$mysqli = @new mysqli('localhost', 'fake_user', 'my_password', 'my_db');

if ($mysqli->connect_errno) {
    die('Connect Error: ' . $mysqli->connect_errno);
}
?>
```

Procedural style

```
<?php
$link = @mysqli_connect('localhost', 'fake_user', 'my_password', 'my_db');

if (!$link) {
    die('Connect Error: ' . mysqli_connect_errno());
}
?>
```

The above examples will output:

```
Connect Error: 1045
```

See Also

mysqli_connect
mysqli_connect_error
mysqli_errno
mysqli_error
mysqli_sqlstate

3.9.11 mysqli::$connect_error, mysqli_connect_error

- mysqli::$connect_error

mysqli_connect_error

Returns a string description of the last connect error

Description

Object oriented style

```
string
  mysqli->connect_error ;
```

Procedural style

```
string mysqli_connect_error();
```

Returns the last error message string from the last call to mysqli_connect.

Return Values

A string that describes the error. NULL is returned if no error occurred.

Examples

Example 3.40 $mysqli->connect_error example

Object oriented style

```
<?php
$mysqli = @new mysqli('localhost', 'fake_user', 'my_password', 'my_db');

// Works as of PHP 5.2.9 and 5.3.0.
if ($mysqli->connect_error) {
    die('Connect Error: ' . $mysqli->connect_error);
}
?>
```

Procedural style

```
<?php
$link = @mysqli_connect('localhost', 'fake_user', 'my_password', 'my_db');

if (!$link) {
    die('Connect Error: ' . mysqli_connect_error());
}
?>
```

The above examples will output:

```
Connect Error: Access denied for user 'fake_user'@'localhost' (using password: YES)
```

Notes

> **Warning**
>
> The mysqli->connect_error property only works properly as of PHP versions 5.2.9 and 5.3.0. Use the mysqli_connect_error function if compatibility with earlier PHP versions is required.

See Also

mysqli_connect
mysqli_connect_errno
mysqli_errno
mysqli_error
mysqli_sqlstate

3.9.12 mysqli::__construct, mysqli_connect

- mysqli::__construct

  mysqli_connect

  Open a new connection to the MySQL server

Description

Object oriented style

```
mysqli::__construct(
  string host
      =  =ini_get("mysqli.default_host"),
  string username
      =  =ini_get("mysqli.default_user"),
  string passwd
      =  =ini_get("mysqli.default_pw"),
  string dbname
      =  ="",
  int port
      =  =ini_get("mysqli.default_port"),
  string socket
      =  =ini_get("mysqli.default_socket"));
```

Procedural style

```
mysqli mysqli_connect(
  string host
      =  =ini_get("mysqli.default_host"),
  string username
      =  =ini_get("mysqli.default_user"),
  string passwd
      =  =ini_get("mysqli.default_pw"),
  string dbname
      =  ="",
  int port
      =  =ini_get("mysqli.default_port"),
  string socket
```

```
        =  =ini_get("mysqli.default_socket"));
```

Opens a connection to the MySQL Server running on.

Parameters

host	Can be either a host name or an IP address. Passing the NULL value or the string "localhost" to this parameter, the local host is assumed. When possible, pipes will be used instead of the TCP/IP protocol. Prepending host by p: opens a persistent connection. mysqli_change_user is automatically called on connections opened from the connection pool.
username	The MySQL user name.
passwd	If not provided or NULL, the MySQL server will attempt to authenticate the user against those user records which have no password only. This allows one username to be used with different permissions (depending on if a password is provided or not).
dbname	If provided will specify the default database to be used when performing queries.
port	Specifies the port number to attempt to connect to the MySQL server.
socket	Specifies the socket or named pipe that should be used.

Note

Specifying the *socket* parameter will not explicitly determine the type of connection to be used when connecting to the MySQL server. How the connection is made to the MySQL database is determined by the *host* parameter.

Return Values

Returns an object which represents the connection to a MySQL Server.

Changelog

Version	Description
5.3.0	Added the ability of persistent connections.

Examples

Example 3.41 mysqli::__construct example

Object oriented style

```
<?php
$mysqli = new mysqli('localhost', 'my_user', 'my_password', 'my_db');

/*
```

```
 * This is the "official" OO way to do it,
 * BUT $connect_error was broken until PHP 5.2.9 and 5.3.0.
 */
if ($mysqli->connect_error) {
    die('Connect Error (' . $mysqli->connect_errno . ') '
            . $mysqli->connect_error);
}

/*
 * Use this instead of $connect_error if you need to ensure
 * compatibility with PHP versions prior to 5.2.9 and 5.3.0.
 */
if (mysqli_connect_error()) {
    die('Connect Error (' . mysqli_connect_errno() . ') '
            . mysqli_connect_error());
}

echo 'Success... ' . $mysqli->host_info . "\n";

$mysqli->close();
?>
```

Object oriented style when extending mysqli class

```
<?php

class foo_mysqli extends mysqli {
    public function __construct($host, $user, $pass, $db) {
        parent::__construct($host, $user, $pass, $db);

        if (mysqli_connect_error()) {
            die('Connect Error (' . mysqli_connect_errno() . ') '
                    . mysqli_connect_error());
        }
    }
}

$db = new foo_mysqli('localhost', 'my_user', 'my_password', 'my_db');

echo 'Success... ' . $db->host_info . "\n";

$db->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect('localhost', 'my_user', 'my_password', 'my_db');

if (!$link) {
    die('Connect Error (' . mysqli_connect_errno() . ') '
            . mysqli_connect_error());
}

echo 'Success... ' . mysqli_get_host_info($link) . "\n";

mysqli_close($link);
?>
```

The above examples will output:

```
Success... MySQL host info: localhost via TCP/IP
```

Notes

> **Note**
>
> MySQLnd always assumes the server default charset. This charset is sent during connection hand-shake/authentication, which mysqlnd will use.
>
> Libmysqlclient uses the default charset set in the `my.cnf` or by an explicit call to `mysqli_options` prior to calling `mysqli_real_connect`, but after `mysqli_init`.

> **Note**
>
> OO syntax only: If a connection fails an object is still returned. To check if the connection failed then use either the `mysqli_connect_error` function or the mysqli->connect_error property as in the preceding examples.

> **Note**
>
> If it is necessary to set options, such as the connection timeout, `mysqli_real_connect` must be used instead.

> **Note**
>
> Calling the constructor with no parameters is the same as calling `mysqli_init`.

> **Note**
>
> Error "Can't create TCP/IP socket (10106)" usually means that the variables_order configure directive doesn't contain character `E`. On Windows, if the environment is not copied the `SYSTEMROOT` environment variable won't be available and PHP will have problems loading Winsock.

See Also

`mysqli_real_connect`
`mysqli_options`
`mysqli_connect_errno`
`mysqli_connect_error`
`mysqli_close`

3.9.13 `mysqli::debug`, `mysqli_debug`

- `mysqli::debug`

  `mysqli_debug`

Performs debugging operations

Description

Object oriented style

```
bool mysqli::debug(
  string message);
```

Procedural style

```
bool mysqli_debug(
  string message);
```

Performs debugging operations using the Fred Fish debugging library.

Parameters

message	A string representing the debugging operation to perform

Return Values

Returns TRUE.

Notes

> **Note**
>
> To use the mysqli_debug function you must compile the MySQL client library to support debugging.

Examples

Example 3.42 Generating a Trace File

```
<?php

/* Create a trace file in '/tmp/client.trace' on the local (client) machine: */
mysqli_debug("d:t:o,/tmp/client.trace");

?>
```

See Also

mysqli_dump_debug_info
mysqli_report

3.9.14 mysqli::dump_debug_info, mysqli_dump_debug_info

- mysqli::dump_debug_info

  mysqli_dump_debug_info

Dump debugging information into the log

Description

Object oriented style

```
bool mysqli::dump_debug_info();
```

Procedural style

```
bool mysqli_dump_debug_info(
  mysqli link);
```

This function is designed to be executed by an user with the SUPER privilege and is used to dump debugging information into the log for the MySQL Server relating to the connection.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

Returns TRUE on success or FALSE on failure.

See Also

mysqli_debug

3.9.15 mysqli::$errno, mysqli_errno

- mysqli::$errno

  mysqli_errno

  Returns the error code for the most recent function call

Description

Object oriented style

```
int
  mysqli->errno ;
```

Procedural style

```
int mysqli_errno(
  mysqli link);
```

Returns the last error code for the most recent MySQLi function call that can succeed or fail.

Client error message numbers are listed in the MySQL errmsg.h header file, server error message numbers are listed in mysqld_error.h. In the MySQL source distribution you can find a complete list of error messages and error numbers in the file Docs/mysqld_error.txt.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

An error code value for the last call, if it failed. zero means no error occurred.

Examples

Example 3.43 $mysqli->errno example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if ($mysqli->connect_errno) {
    printf("Connect failed: %s\n", $mysqli->connect_error);
    exit();
}

if (!$mysqli->query("SET a=1")) {
    printf("Errorcode: %d\n", $mysqli->errno);
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

if (!mysqli_query($link, "SET a=1")) {
    printf("Errorcode: %d\n", mysqli_errno($link));
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Errorcode: 1193
```

See Also

mysqli_connect_errno
mysqli_connect_error
mysqli_error
mysqli_sqlstate

3.9.16 mysqli::$error_list, mysqli_error_list

- mysqli::$error_list

  mysqli_error_list

  Returns a list of errors from the last command executed

Description

Object oriented style

```
array
  mysqli->error_list ;
```

Procedural style

```
array mysqli_error_list(
  mysqli link);
```

Returns a array of errors for the most recent MySQLi function call that can succeed or fail.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

A list of errors, each as an associative array containing the errno, error, and sqlstate.

Examples

Example 3.44 $mysqli->error_list example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "nobody", "");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

if (!$mysqli->query("SET a=1")) {
    print_r($mysqli->error_list);
}
```

```
/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

if (!mysqli_query($link, "SET a=1")) {
    print_r(mysqli_error_list($link));
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Array
(
    [0] => Array
        (
            [errno] => 1193
            [sqlstate] => HY000
            [error] => Unknown system variable 'a'
        )

)
```

See Also

mysqli_connect_errno
mysqli_connect_error
mysqli_error
mysqli_sqlstate

3.9.17 mysqli::$error, mysqli_error

- mysqli::$error

  mysqli_error

  Returns a string description of the last error

Description

Object oriented style

```
  string
    mysqli->error ;
```

Procedural style

```
  string mysqli_error(
    mysqli link);
```

Returns the last error message for the most recent MySQLi function call that can succeed or fail.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

A string that describes the error. An empty string if no error occurred.

Examples

Example 3.45 $mysqli->error example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if ($mysqli->connect_errno) {
    printf("Connect failed: %s\n", $mysqli->connect_error);
    exit();
}

if (!$mysqli->query("SET a=1")) {
    printf("Errormessage: %s\n", $mysqli->error);
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}
```

```
if (!mysqli_query($link, "SET a=1")) {
    printf("Errormessage: %s\n", mysqli_error($link));
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Errormessage: Unknown system variable 'a'
```

See Also

mysqli_connect_errno
mysqli_connect_error
mysqli_errno
mysqli_sqlstate

3.9.18 mysqli::$field_count, mysqli_field_count

- mysqli::$field_count

  mysqli_field_count

  Returns the number of columns for the most recent query

Description

Object oriented style

```
  int
    mysqli->field_count ;
```

Procedural style

```
  int mysqli_field_count(
    mysqli link);
```

Returns the number of columns for the most recent query on the connection represented by the *link* parameter. This function can be useful when using the mysqli_store_result function to determine if the query should have produced a non-empty result set or not without knowing the nature of the query.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

An integer representing the number of fields in a result set.

Examples

Example 3.46 `$mysqli->field_count` example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "test");

$mysqli->query( "DROP TABLE IF EXISTS friends");
$mysqli->query( "CREATE TABLE friends (id int, name varchar(20))");

$mysqli->query( "INSERT INTO friends VALUES (1,'Hartmut'), (2, 'Ulf')");


$mysqli->real_query("SELECT * FROM friends");

if ($mysqli->field_count) {
    /* this was a select/show or describe query */
    $result = $mysqli->store_result();

    /* process resultset */
    $row = $result->fetch_row();

    /* free resultset */
    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "test");

mysqli_query($link, "DROP TABLE IF EXISTS friends");
mysqli_query($link, "CREATE TABLE friends (id int, name varchar(20))");

mysqli_query($link, "INSERT INTO friends VALUES (1,'Hartmut'), (2, 'Ulf')");

mysqli_real_query($link, "SELECT * FROM friends");

if (mysqli_field_count($link)) {
    /* this was a select/show or describe query */
    $result = mysqli_store_result($link);

    /* process resultset */
    $row = mysqli_fetch_row($result);

    /* free resultset */
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

3.9.19 mysqli::get_charset, mysqli_get_charset

- mysqli::get_charset

  mysqli_get_charset

  Returns a character set object

Description

Object oriented style

```
object mysqli::get_charset();
```

Procedural style

```
object mysqli_get_charset(
  mysqli link);
```

Returns a character set object providing several properties of the current active character set.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

The function returns a character set object with the following properties:

charset	Character set name
collation	Collation name
dir	Directory the charset description was fetched from (?) or "" for built-in character sets
min_length	Minimum character length in bytes
max_length	Maximum character length in bytes
number	Internal character set number
state	Character set status (?)

Examples

Example 3.47 mysqli::get_charset example

Object oriented style

```
<?php
  $db = mysqli_init();
  $db->real_connect("localhost","root","","test");
  var_dump($db->get_charset());
```

```
?>
```

Procedural style

```
<?php
  $db = mysqli_init();
  mysqli_real_connect($db, "localhost","root","","test");
  var_dump(mysqli_get_charset($db));
?>
```

The above examples will output:

```
object(stdClass)#2 (7) {
  ["charset"]=>
  string(6) "latin1"
  ["collation"]=>
  string(17) "latin1_swedish_ci"
  ["dir"]=>
  string(0) ""
  ["min_length"]=>
  int(1)
  ["max_length"]=>
  int(1)
  ["number"]=>
  int(8)
  ["state"]=>
  int(801)
}
```

See Also

mysqli_character_set_name
mysqli_set_charset

3.9.20 mysqli::get_client_info, mysqli_get_client_info

- mysqli::get_client_info

  mysqli_get_client_info

  Get MySQL client info

Description

Object oriented style

```
  string mysqli::get_client_info();
```

Procedural style

```
  string mysqli_get_client_info(
```

```
    mysqli link);
```

Returns a string that represents the MySQL client library version.

Return Values

A string that represents the MySQL client library version

Examples

Example 3.48 mysqli_get_client_info

```
<?php

/* We don't need a connection to determine
   the version of mysql client library */

printf("Client library version: %s\n", mysqli_get_client_info());
?>
```

See Also

mysqli_get_client_version
mysqli_get_server_info
mysqli_get_server_version

3.9.21 mysqli_get_client_stats

- mysqli_get_client_stats

  Returns client per-process statistics

Description

```
  array mysqli_get_client_stats();
```

Returns client per-process statistics. Available only with mysqlnd.

Parameters

Return Values

Returns an array with client stats if success, FALSE otherwise.

Examples

Example 3.49 A mysqli_get_client_stats example

```
<?php
$link = mysqli_connect();
print_r(mysqli_get_client_stats());
?>
```

The above example will output something similar to:

```
Array
(
    [bytes_sent] => 43
    [bytes_received] => 80
    [packets_sent] => 1
    [packets_received] => 2
    [protocol_overhead_in] => 8
    [protocol_overhead_out] => 4
    [bytes_received_ok_packet] => 11
    [bytes_received_eof_packet] => 0
    [bytes_received_rset_header_packet] => 0
    [bytes_received_rset_field_meta_packet] => 0
    [bytes_received_rset_row_packet] => 0
    [bytes_received_prepare_response_packet] => 0
    [bytes_received_change_user_packet] => 0
    [packets_sent_command] => 0
    [packets_received_ok] => 1
    [packets_received_eof] => 0
    [packets_received_rset_header] => 0
    [packets_received_rset_field_meta] => 0
    [packets_received_rset_row] => 0
    [packets_received_prepare_response] => 0
    [packets_received_change_user] => 0
    [result_set_queries] => 0
    [non_result_set_queries] => 0
    [no_index_used] => 0
    [bad_index_used] => 0
    [slow_queries] => 0
    [buffered_sets] => 0
    [unbuffered_sets] => 0
    [ps_buffered_sets] => 0
    [ps_unbuffered_sets] => 0
    [flushed_normal_sets] => 0
    [flushed_ps_sets] => 0
    [ps_prepared_never_executed] => 0
    [ps_prepared_once_executed] => 0
    [rows_fetched_from_server_normal] => 0
    [rows_fetched_from_server_ps] => 0
    [rows_buffered_from_client_normal] => 0
    [rows_buffered_from_client_ps] => 0
    [rows_fetched_from_client_normal_buffered] => 0
    [rows_fetched_from_client_normal_unbuffered] => 0
    [rows_fetched_from_client_ps_buffered] => 0
    [rows_fetched_from_client_ps_unbuffered] => 0
    [rows_fetched_from_client_ps_cursor] => 0
    [rows_skipped_normal] => 0
    [rows_skipped_ps] => 0
    [copy_on_write_saved] => 0
    [copy_on_write_performed] => 0
    [command_buffer_too_small] => 0
    [connect_success] => 1
    [connect_failure] => 0
    [connection_reused] => 0
    [reconnect] => 0
    [pconnect_success] => 0
    [active_connections] => 1
    [active_persistent_connections] => 0
    [explicit_close] => 0
    [implicit_close] => 0
    [disconnect_close] => 0
    [in_middle_of_command_close] => 0
```

```
    [explicit_free_result] => 0
    [implicit_free_result] => 0
    [explicit_stmt_close] => 0
    [implicit_stmt_close] => 0
    [mem_emalloc_count] => 0
    [mem_emalloc_ammount] => 0
    [mem_ecalloc_count] => 0
    [mem_ecalloc_ammount] => 0
    [mem_erealloc_count] => 0
    [mem_erealloc_ammount] => 0
    [mem_efree_count] => 0
    [mem_malloc_count] => 0
    [mem_malloc_ammount] => 0
    [mem_calloc_count] => 0
    [mem_calloc_ammount] => 0
    [mem_realloc_count] => 0
    [mem_realloc_ammount] => 0
    [mem_free_count] => 0
    [proto_text_fetched_null] => 0
    [proto_text_fetched_bit] => 0
    [proto_text_fetched_tinyint] => 0
    [proto_text_fetched_short] => 0
    [proto_text_fetched_int24] => 0
    [proto_text_fetched_int] => 0
    [proto_text_fetched_bigint] => 0
    [proto_text_fetched_decimal] => 0
    [proto_text_fetched_float] => 0
    [proto_text_fetched_double] => 0
    [proto_text_fetched_date] => 0
    [proto_text_fetched_year] => 0
    [proto_text_fetched_time] => 0
    [proto_text_fetched_datetime] => 0
    [proto_text_fetched_timestamp] => 0
    [proto_text_fetched_string] => 0
    [proto_text_fetched_blob] => 0
    [proto_text_fetched_enum] => 0
    [proto_text_fetched_set] => 0
    [proto_text_fetched_geometry] => 0
    [proto_text_fetched_other] => 0
    [proto_binary_fetched_null] => 0
    [proto_binary_fetched_bit] => 0
    [proto_binary_fetched_tinyint] => 0
    [proto_binary_fetched_short] => 0
    [proto_binary_fetched_int24] => 0
    [proto_binary_fetched_int] => 0
    [proto_binary_fetched_bigint] => 0
    [proto_binary_fetched_decimal] => 0
    [proto_binary_fetched_float] => 0
    [proto_binary_fetched_double] => 0
    [proto_binary_fetched_date] => 0
    [proto_binary_fetched_year] => 0
    [proto_binary_fetched_time] => 0
    [proto_binary_fetched_datetime] => 0
    [proto_binary_fetched_timestamp] => 0
    [proto_binary_fetched_string] => 0
    [proto_binary_fetched_blob] => 0
    [proto_binary_fetched_enum] => 0
    [proto_binary_fetched_set] => 0
    [proto_binary_fetched_geometry] => 0
    [proto_binary_fetched_other] => 0
)
```

See Also

Stats description

3.9.22 mysqli_get_client_version, mysqli::$client_version

- `mysqli_get_client_version`

  `mysqli::$client_version`

  Returns the MySQL client version as an integer

Description

Object oriented style

```
int
  mysqli->client_version ;
```

Procedural style

```
int mysqli_get_client_version(
  mysqli link);
```

Returns client version number as an integer.

Return Values

A number that represents the MySQL client library version in format: `main_version*10000 + minor_version *100 + sub_version`. For example, 4.1.0 is returned as 40100.

This is useful to quickly determine the version of the client library to know if some capability exits.

Examples

Example 3.50 mysqli_get_client_version

```
<?php

/* We don't need a connection to determine
   the version of mysql client library */

printf("Client library version: %d\n", mysqli_get_client_version());
?>
```

See Also

`mysqli_get_client_info`
`mysqli_get_server_info`
`mysqli_get_server_version`

3.9.23 mysqli::get_connection_stats, mysqli_get_connection_stats

- `mysqli::get_connection_stats`

  `mysqli_get_connection_stats`

  Returns statistics about the client connection

Description

Object oriented style

```
bool mysqli::get_connection_stats();
```

Procedural style

```
array mysqli_get_connection_stats(
  mysqli link);
```

Returns statistics about the client connection. Available only with mysqlnd.

Parameters

link	Procedural style only: A link identifier returned by `mysqli_connect` or `mysqli_init`

Return Values

Returns an array with connection stats if success, `FALSE` otherwise.

Examples

Example 3.51 A `mysqli_get_connection_stats` example

```
<?php
$link = mysqli_connect();
print_r(mysqli_get_connection_stats($link));
?>
```

The above example will output something similar to:

```
Array
(
    [bytes_sent] => 43
    [bytes_received] => 80
    [packets_sent] => 1
    [packets_received] => 2
    [protocol_overhead_in] => 8
    [protocol_overhead_out] => 4
    [bytes_received_ok_packet] => 11
    [bytes_received_eof_packet] => 0
    [bytes_received_rset_header_packet] => 0
    [bytes_received_rset_field_meta_packet] => 0
    [bytes_received_rset_row_packet] => 0
    [bytes_received_prepare_response_packet] => 0
    [bytes_received_change_user_packet] => 0
    [packets_sent_command] => 0
    [packets_received_ok] => 1
    [packets_received_eof] => 0
    [packets_received_rset_header] => 0
    [packets_received_rset_field_meta] => 0
    [packets_received_rset_row] => 0
    [packets_received_prepare_response] => 0
    [packets_received_change_user] => 0
    [result_set_queries] => 0
    [non_result_set_queries] => 0
    [no_index_used] => 0
    [bad_index_used] => 0
```

```
    [slow_queries] => 0
    [buffered_sets] => 0
    [unbuffered_sets] => 0
    [ps_buffered_sets] => 0
    [ps_unbuffered_sets] => 0
    [flushed_normal_sets] => 0
    [flushed_ps_sets] => 0
    [ps_prepared_never_executed] => 0
    [ps_prepared_once_executed] => 0
    [rows_fetched_from_server_normal] => 0
    [rows_fetched_from_server_ps] => 0
    [rows_buffered_from_client_normal] => 0
    [rows_buffered_from_client_ps] => 0
    [rows_fetched_from_client_normal_buffered] => 0
    [rows_fetched_from_client_normal_unbuffered] => 0
    [rows_fetched_from_client_ps_buffered] => 0
    [rows_fetched_from_client_ps_unbuffered] => 0
    [rows_fetched_from_client_ps_cursor] => 0
    [rows_skipped_normal] => 0
    [rows_skipped_ps] => 0
    [copy_on_write_saved] => 0
    [copy_on_write_performed] => 0
    [command_buffer_too_small] => 0
    [connect_success] => 1
    [connect_failure] => 0
    [connection_reused] => 0
    [reconnect] => 0
    [pconnect_success] => 0
    [active_connections] => 1
    [active_persistent_connections] => 0
    [explicit_close] => 0
    [implicit_close] => 0
    [disconnect_close] => 0
    [in_middle_of_command_close] => 0
    [explicit_free_result] => 0
    [implicit_free_result] => 0
    [explicit_stmt_close] => 0
    [implicit_stmt_close] => 0
    [mem_emalloc_count] => 0
    [mem_emalloc_ammount] => 0
    [mem_ecalloc_count] => 0
    [mem_ecalloc_ammount] => 0
    [mem_erealloc_count] => 0
    [mem_erealloc_ammount] => 0
    [mem_efree_count] => 0
    [mem_malloc_count] => 0
    [mem_malloc_ammount] => 0
    [mem_calloc_count] => 0
    [mem_calloc_ammount] => 0
    [mem_realloc_count] => 0
    [mem_realloc_ammount] => 0
    [mem_free_count] => 0
    [proto_text_fetched_null] => 0
    [proto_text_fetched_bit] => 0
    [proto_text_fetched_tinyint] => 0
    [proto_text_fetched_short] => 0
    [proto_text_fetched_int24] => 0
    [proto_text_fetched_int] => 0
    [proto_text_fetched_bigint] => 0
    [proto_text_fetched_decimal] => 0
    [proto_text_fetched_float] => 0
    [proto_text_fetched_double] => 0
    [proto_text_fetched_date] => 0
    [proto_text_fetched_year] => 0
    [proto_text_fetched_time] => 0
    [proto_text_fetched_datetime] => 0
    [proto_text_fetched_timestamp] => 0
```

```
    [proto_text_fetched_string] => 0
    [proto_text_fetched_blob] => 0
    [proto_text_fetched_enum] => 0
    [proto_text_fetched_set] => 0
    [proto_text_fetched_geometry] => 0
    [proto_text_fetched_other] => 0
    [proto_binary_fetched_null] => 0
    [proto_binary_fetched_bit] => 0
    [proto_binary_fetched_tinyint] => 0
    [proto_binary_fetched_short] => 0
    [proto_binary_fetched_int24] => 0
    [proto_binary_fetched_int] => 0
    [proto_binary_fetched_bigint] => 0
    [proto_binary_fetched_decimal] => 0
    [proto_binary_fetched_float] => 0
    [proto_binary_fetched_double] => 0
    [proto_binary_fetched_date] => 0
    [proto_binary_fetched_year] => 0
    [proto_binary_fetched_time] => 0
    [proto_binary_fetched_datetime] => 0
    [proto_binary_fetched_timestamp] => 0
    [proto_binary_fetched_string] => 0
    [proto_binary_fetched_blob] => 0
    [proto_binary_fetched_enum] => 0
    [proto_binary_fetched_set] => 0
    [proto_binary_fetched_geometry] => 0
    [proto_binary_fetched_other] => 0
)
```

See Also

Stats description

3.9.24 mysqli::$host_info, mysqli_get_host_info

- mysqli::$host_info

  mysqli_get_host_info

  Returns a string representing the type of connection used

Description

Object oriented style

```
  string
    mysqli->host_info ;
```

Procedural style

```
  string mysqli_get_host_info(
    mysqli link);
```

Returns a string describing the connection represented by the *link* parameter (including the server host name).

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

A character string representing the server hostname and the connection type.

Examples

Example 3.52 `$mysqli->host_info` example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* print host information */
printf("Host info: %s\n", $mysqli->host_info);

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* print host information */
printf("Host info: %s\n", mysqli_get_host_info($link));

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Host info: Localhost via UNIX socket
```

See Also

mysqli_get_proto_info

3.9.25 mysqli::$protocol_version, mysqli_get_proto_info

- mysqli::$protocol_version

  mysqli_get_proto_info

  Returns the version of the MySQL protocol used

Description

Object oriented style

```
  string
    mysqli->protocol_version ;
```

Procedural style

```
  int mysqli_get_proto_info(
    mysqli link);
```

Returns an integer representing the MySQL protocol version used by the connection represented by the *link* parameter.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

Returns an integer representing the protocol version.

Examples

Example 3.53 $mysqli->protocol_version example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* print protocol version */
printf("Protocol version: %d\n", $mysqli->protocol_version);

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* print protocol version */
printf("Protocol version: %d\n", mysqli_get_proto_info($link));

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Protocol version: 10
```

See Also

mysqli_get_host_info

3.9.26 mysqli::$server_info, mysqli_get_server_info

- mysqli::$server_info

  mysqli_get_server_info

  Returns the version of the MySQL server

Description

Object oriented style

```
  string
    mysqli->server_info ;
```

Procedural style

```
  string mysqli_get_server_info(
    mysqli link);
```

Returns a string representing the version of the MySQL server that the MySQLi extension is connected to.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

A character string representing the server version.

Examples

Example 3.54 `$mysqli->server_info` example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* print server version */
printf("Server version: %s\n", $mysqli->server_info);

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* print server version */
printf("Server version: %s\n", mysqli_get_server_info($link));

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Server version: 4.1.2-alpha-debug
```

See Also

mysqli_get_client_info
mysqli_get_client_version
mysqli_get_server_version

3.9.27 mysqli::$server_version, mysqli_get_server_version

- mysqli::$server_version

  mysqli_get_server_version

  Returns the version of the MySQL server as an integer

Description

Object oriented style

```
int
  mysqli->server_version ;
```

Procedural style

```
int mysqli_get_server_version(
  mysqli link);
```

The mysqli_get_server_version function returns the version of the server connected to (represented by the *link* parameter) as an integer.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

An integer representing the server version.

The form of this version number is main_version * 10000 + minor_version * 100 + sub_version (i.e. version 4.1.0 is 40100).

Examples

Example 3.55 $mysqli->server_version example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* print server version */
printf("Server version: %d\n", $mysqli->server_version);

/* close connection */
$mysqli->close();
?>
```

```
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* print server version */
printf("Server version: %d\n", mysqli_get_server_version($link));

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Server version: 40102
```

See Also

mysqli_get_client_info
mysqli_get_client_version
mysqli_get_server_info

3.9.28 mysqli::get_warnings, mysqli_get_warnings

- mysqli::get_warnings

  mysqli_get_warnings

  Get result of SHOW WARNINGS

Description

Object oriented style

```
  mysqli_warning mysqli::get_warnings();
```

Procedural style

```
  mysqli_warning mysqli_get_warnings(
    mysqli link);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

3.9.29 mysqli::$info, mysqli_info

- mysqli::$info

  mysqli_info

  Retrieves information about the most recently executed query

Description

Object oriented style

```
string
  mysqli->info ;
```

Procedural style

```
string mysqli_info(
  mysqli link);
```

The mysqli_info function returns a string providing information about the last query executed. The nature of this string is provided below:

Table 3.9 Possible mysqli_info return values

Query type	Example result string
INSERT INTO...SELECT...	Records: 100 Duplicates: 0 Warnings: 0
INSERT INTO...VALUES (...),(...),(...)	Records: 3 Duplicates: 0 Warnings: 0
LOAD DATA INFILE ...	Records: 1 Deleted: 0 Skipped: 0 Warnings: 0
ALTER TABLE ...	Records: 3 Duplicates: 0 Warnings: 0
UPDATE ...	Rows matched: 40 Changed: 40 Warnings: 0

Note

Queries which do not fall into one of the preceding formats are not supported. In these situations, mysqli_info will return an empty string.

Parameters

link — Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

A character string representing additional information about the most recently executed query.

Examples

Example 3.56 $mysqli->info example

Object oriented style

```

<?php
```

```
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$mysqli->query("CREATE TEMPORARY TABLE t1 LIKE City");

/* INSERT INTO .. SELECT */
$mysqli->query("INSERT INTO t1 SELECT * FROM City ORDER BY ID LIMIT 150");
printf("%s\n", $mysqli->info);

/* close connection */
$mysqli->close();
?>
```

Procedural style

```


<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

mysqli_query($link, "CREATE TEMPORARY TABLE t1 LIKE City");

/* INSERT INTO .. SELECT */
mysqli_query($link, "INSERT INTO t1 SELECT * FROM City ORDER BY ID LIMIT 150");
printf("%s\n", mysqli_info($link));

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```


Records: 150  Duplicates: 0  Warnings: 0
```

See Also

mysqli_affected_rows
mysqli_warning_count
mysqli_num_rows

3.9.30 mysqli::init, mysqli_init

- mysqli::init

mysqli_init

Initializes MySQLi and returns a resource for use with mysqli_real_connect()

Description

Object oriented style

```
mysqli mysqli::init();
```

Procedural style

```
mysqli mysqli_init();
```

Allocates or initializes a MYSQL object suitable for mysqli_options and mysqli_real_connect.

> **Note**
>
> Any subsequent calls to any mysqli function (except mysqli_options) will fail until mysqli_real_connect was called.

Return Values

Returns an object.

Examples

See mysqli_real_connect.

See Also

mysqli_options
mysqli_close
mysqli_real_connect
mysqli_connect

3.9.31 mysqli::$insert_id, mysqli_insert_id

- mysqli::$insert_id

  mysqli_insert_id

  Returns the auto generated id used in the latest query

Description

Object oriented style

```
mixed
  mysqli->insert_id ;
```

Procedural style

```
mixed mysqli_insert_id(
  mysqli link);
```

The mysqli_insert_id function returns the ID generated by a query (usually INSERT) on a table with a column having the AUTO_INCREMENT attribute. If no INSERT or UPDATE statements were sent via this connection, or if the modified table does not have a column with the AUTO_INCREMENT attribute, this function will return zero.

> **Note**
>
> Performing an INSERT or UPDATE statement using the LAST_INSERT_ID() function will also modify the value returned by the mysqli_insert_id function.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

The value of the AUTO_INCREMENT field that was updated by the previous query. Returns zero if there was no previous query on the connection or if the query did not update an AUTO_INCREMENT value.

> **Note**
>
> If the number is greater than maximal int value, mysqli_insert_id will return a string.

Examples

Example 3.57 $mysqli->insert_id example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$mysqli->query("CREATE TABLE myCity LIKE City");

$query = "INSERT INTO myCity VALUES (NULL, 'Stuttgart', 'DEU', 'Stuttgart', 617000)";
$mysqli->query($query);

printf ("New Record has id %d.\n", $mysqli->insert_id);

/* drop table */
$mysqli->query("DROP TABLE myCity");

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

mysqli_query($link, "CREATE TABLE myCity LIKE City");

$query = "INSERT INTO myCity VALUES (NULL, 'Stuttgart', 'DEU', 'Stuttgart', 617000)";
mysqli_query($link, $query);

printf ("New Record has id %d.\n", mysqli_insert_id($link));

/* drop table */
mysqli_query($link, "DROP TABLE myCity");

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
New Record has id 1.
```

3.9.32 mysqli::kill, mysqli_kill

- `mysqli::kill`

  `mysqli_kill`

  Asks the server to kill a MySQL thread

Description

Object oriented style

```
bool mysqli::kill(
  int processid);
```

Procedural style

```
bool mysqli_kill(
  mysqli link,
  int processid);
```

This function is used to ask the server to kill a MySQL thread specified by the *`processid`* parameter. This value must be retrieved by calling the `mysqli_thread_id` function.

To stop a running query you should use the SQL command `KILL QUERY processid`.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.58 mysqli::kill example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* determine our thread id */
$thread_id = $mysqli->thread_id;

/* Kill connection */
$mysqli->kill($thread_id);

/* This should produce an error */
if (!$mysqli->query("CREATE TABLE myCity LIKE City")) {
    printf("Error: %s\n", $mysqli->error);
    exit;
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* determine our thread id */
$thread_id = mysqli_thread_id($link);

/* Kill connection */
mysqli_kill($link, $thread_id);

/* This should produce an error */
if (!mysqli_query($link, "CREATE TABLE myCity LIKE City")) {
    printf("Error: %s\n", mysqli_error($link));
    exit;
```

```
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Error: MySQL server has gone away
```

See Also

mysqli_thread_id

3.9.33 mysqli::more_results, mysqli_more_results

- mysqli::more_results

  mysqli_more_results

  Check if there are any more query results from a multi query

Description

Object oriented style

```
bool mysqli::more_results();
```

Procedural style

```
bool mysqli_more_results(
  mysqli link);
```

Indicates if one or more result sets are available from a previous call to mysqli_multi_query.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

Returns TRUE if one or more result sets are available from a previous call to mysqli_multi_query, otherwise FALSE.

Examples

See mysqli_multi_query.

See Also

mysqli_multi_query
mysqli_next_result

mysqli_store_result
mysqli_use_result

3.9.34 mysqli::multi_query, mysqli_multi_query

- mysqli::multi_query

  mysqli_multi_query

  Performs a query on the database

Description

Object oriented style

```
bool mysqli::multi_query(
  string query);
```

Procedural style

```
bool mysqli_multi_query(
  mysqli link,
  string query);
```

Executes one or multiple queries which are concatenated by a semicolon.

To retrieve the resultset from the first query you can use mysqli_use_result or mysqli_store_result. All subsequent query results can be processed using mysqli_more_results and mysqli_next_result.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
query	The query, as a string. Data inside the query should be properly escaped.

Return Values

Returns FALSE if the first statement failed. To retrieve subsequent errors from other statements you have to call mysqli_next_result first.

Examples

Example 3.59 mysqli::multi_query example

Object oriented style

```


<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
```

```
    exit();
}

$query  = "SELECT CURRENT_USER();";
$query .= "SELECT Name FROM City ORDER BY ID LIMIT 20, 5";

/* execute multi query */
if ($mysqli->multi_query($query)) {
    do {
        /* store first result set */
        if ($result = $mysqli->store_result()) {
            while ($row = $result->fetch_row()) {
                printf("%s\n", $row[0]);
            }
            $result->free();
        }
        /* print divider */
        if ($mysqli->more_results()) {
            printf("-----------------\n");
        }
    } while ($mysqli->next_result());
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query  = "SELECT CURRENT_USER();";
$query .= "SELECT Name FROM City ORDER BY ID LIMIT 20, 5";

/* execute multi query */
if (mysqli_multi_query($link, $query)) {
    do {
        /* store first result set */
        if ($result = mysqli_store_result($link)) {
            while ($row = mysqli_fetch_row($result)) {
                printf("%s\n", $row[0]);
            }
            mysqli_free_result($result);
        }
        /* print divider */
        if (mysqli_more_results($link)) {
            printf("-----------------\n");
        }
    } while (mysqli_next_result($link));
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output something similar to:

```
my_user@localhost
-----------------
Amersfoort
Maastricht
Dordrecht
Leiden
Haarlemmermeer
```

See Also

mysqli_query
mysqli_use_result
mysqli_store_result
mysqli_next_result
mysqli_more_results

3.9.35 mysqli::next_result, mysqli_next_result

- mysqli::next_result

  mysqli_next_result

  Prepare next result from multi_query

Description

Object oriented style

```
bool mysqli::next_result();
```

Procedural style

```
bool mysqli_next_result(
  mysqli link);
```

Prepares next result set from a previous call to mysqli_multi_query which can be retrieved by mysqli_store_result or mysqli_use_result.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

Returns TRUE on success or FALSE on failure.

Examples

See mysqli_multi_query.

See Also

mysqli_multi_query
mysqli_more_results
mysqli_store_result
mysqli_use_result

3.9.36 mysqli::options, mysqli_options

- mysqli::options

  mysqli_options

  Set options

Description

Object oriented style

```
bool mysqli::options(
  int option,
  mixed value);
```

Procedural style

```
bool mysqli_options(
  mysqli link,
  int option,
  mixed value);
```

Used to set extra connect options and affect behavior for a connection.

This function may be called multiple times to set several options.

mysqli_options should be called after mysqli_init and before mysqli_real_connect.

Parameters

link

Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

option

The option that you want to set. It can be one of the following values:

Table 3.10 Valid options

Name	Description
MYSQLI_OPT_CONNECT_TIMEOUT	connection timeout in seconds (supported on Windows with TCP/IP since PHP 5.3.1)
MYSQLI_OPT_LOCAL_INFILE	enable/disable use of LOAD LOCAL INFILE
MYSQLI_INIT_COMMAND	command to execute after when connecting to MySQL server
MYSQLI_READ_DEFAULT_FILE	Read options from named option file instead of my.cnf
MYSQLI_READ_DEFAULT_GROUP	Read options from the named group from my.cnf

Name	Description
	or the file specified with MYSQL_READ_DEFAULT_FILE.
MYSQLI_SERVER_PUBLIC_KEY	RSA public key file used with the SHA-256 based authentication.
MYSQLI_OPT_NET_CMD_BUFFER_SIZE	The size of the internal command/network buffer. Only valid for mysqlnd.
MYSQLI_OPT_NET_READ_BUFFER_SIZE	Maximum read chunk size in bytes when reading the body of a MySQL command packet. Only valid for mysqlnd.
MYSQLI_OPT_INT_AND_FLOAT_NATIVE	Convert integer and float columns back to PHP numbers. Only valid for mysqlnd.
MYSQLI_OPT_SSL_VERIFY_SERVER_CERT	

value The value for the option.

Return Values

Returns TRUE on success or FALSE on failure.

Changelog

Version	Description
5.5.0	The MYSQLI_SERVER_PUBLIC_KEY and MYSQLI_SERVER_PUBLIC_KEY options were added.
5.3.0	The MYSQLI_OPT_INT_AND_FLOAT_NATIVE, MYSQLI_OPT_NET_CMD_BUFFER_SIZE, MYSQLI_OPT_NET_READ_BUFFER_SIZE, and MYSQLI_OPT_SSL_VERIFY_SERVER_CERT options were added.

Examples

See mysqli_real_connect.

Notes

> **Note**
>
> MySQLnd always assumes the server default charset. This charset is sent during connection hand-shake/authentication, which mysqlnd will use.
>
> Libmysqlclient uses the default charset set in the my.cnf or by an explicit call to mysqli_options prior to calling mysqli_real_connect, but after mysqli_init.

See Also

mysqli_init

mysqli_real_connect

3.9.37 mysqli::ping, mysqli_ping

- mysqli::ping

  mysqli_ping

  Pings a server connection, or tries to reconnect if the connection has gone down

Description

Object oriented style

```
bool mysqli::ping();
```

Procedural style

```
bool mysqli_ping(
  mysqli link);
```

Checks whether the connection to the server is working. If it has gone down and global option mysqli.reconnect is enabled, an automatic reconnection is attempted.

> **Note**
>
> The php.ini setting mysqli.reconnect is ignored by the mysqlnd driver, so automatic reconnection is never attempted.

This function can be used by clients that remain idle for a long while, to check whether the server has closed the connection and reconnect if necessary.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.60 mysqli::ping example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if ($mysqli->connect_errno) {
    printf("Connect failed: %s\n", $mysqli->connect_error);
    exit();
}

/* check if server is alive */
```

```
if ($mysqli->ping()) {
    printf ("Our connection is ok!\n");
} else {
    printf ("Error: %s\n", $mysqli->error);
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* check if server is alive */
if (mysqli_ping($link)) {
    printf ("Our connection is ok!\n");
} else {
    printf ("Error: %s\n", mysqli_error($link));
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Our connection is ok!
```

3.9.38 mysqli::poll, mysqli_poll

- mysqli::poll

  mysqli_poll

  Poll connections

Description

Object oriented style

```
  public static int mysqli::poll(
    array read,
    array error,
    array reject,
```

```
    int sec,
    int usec);
```

Procedural style

```
  int mysqli_poll(
    array read,
    array error,
    array reject,
    int sec,
    int usec);
```

Poll connections. Available only with mysqlnd. The method can be used as static.

Parameters

read	List of connections to check for outstanding results that can be read.
error	List of connections on which an error occured, for example, query failure or lost connection.
reject	List of connections rejected because no asynchronous query has been run on for which the function could poll results.
sec	Maximum number of seconds to wait, must be non-negative.
usec	Maximum number of microseconds to wait, must be non-negative.

Return Values

Returns number of ready connections upon success, FALSE otherwise.

Examples

Example 3.61 A `mysqli_poll` example

```
<?php
$link1 = mysqli_connect();
$link1->query("SELECT 'test'", MYSQLI_ASYNC);
$all_links = array($link1);
$processed = 0;
do {
    $links = $errors = $reject = array();
    foreach ($all_links as $link) {
        $links[] = $errors[] = $reject[] = $link;
    }
    if (!mysqli_poll($links, $errors, $reject, 1)) {
        continue;
    }
    foreach ($links as $link) {
        if ($result = $link->reap_async_query()) {
            print_r($result->fetch_row());
            if (is_object($result))
                mysqli_free_result($result);
        } else die(sprintf("MySQLi Error: %s", mysqli_error($link)));
        $processed++;
    }
} while ($processed < count($all_links));
?>
```

The above example will output:

```
Array
(
    [0] => test
)
```

See Also

mysqli_query
mysqli_reap_async_query

3.9.39 mysqli::prepare, mysqli_prepare

- mysqli::prepare

  mysqli_prepare

  Prepare an SQL statement for execution

Description

Object oriented style

```
mysqli_stmt mysqli::prepare(
  string query);
```

Procedural style

```
mysqli_stmt mysqli_prepare(
  mysqli link,
  string query);
```

Prepares the SQL query, and returns a statement handle to be used for further operations on the statement. The query must consist of a single SQL statement.

The parameter markers must be bound to application variables using mysqli_stmt_bind_param and/or mysqli_stmt_bind_result before executing the statement or fetching rows.

Parameters

link — Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

query — The query, as a string.

> **Note**
>
> You should not add a terminating semicolon or \g to the statement.

This parameter can include one or more parameter markers in the SQL statement by embedding question mark (?) characters at the appropriate positions.

Note

The markers are legal only in certain places in SQL statements. For example, they are allowed in the `VALUES()` list of an `INSERT` statement (to specify column values for a row), or in a comparison with a column in a `WHERE` clause to specify a comparison value.

However, they are not allowed for identifiers (such as table or column names), in the select list that names the columns to be returned by a `SELECT` statement, or to specify both operands of a binary operator such as the `=` equal sign. The latter restriction is necessary because it would be impossible to determine the parameter type. It's not allowed to compare marker with `NULL` by `? IS NULL` too. In general, parameters are legal only in Data Manipulation Language (DML) statements, and not in Data Definition Language (DDL) statements.

Return Values

`mysqli_prepare` returns a statement object or `FALSE` if an error occurred.

Examples

Example 3.62 `mysqli::prepare` example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$city = "Amersfoort";

/* create a prepared statement */
if ($stmt = $mysqli->prepare("SELECT District FROM City WHERE Name=?")) {

    /* bind parameters for markers */
    $stmt->bind_param("s", $city);

    /* execute query */
    $stmt->execute();

    /* bind result variables */
    $stmt->bind_result($district);

    /* fetch value */
    $stmt->fetch();
```

```
    printf("%s is in district %s\n", $city, $district);

    /* close statement */
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$city = "Amersfoort";

/* create a prepared statement */
if ($stmt = mysqli_prepare($link, "SELECT District FROM City WHERE Name=?")) {

    /* bind parameters for markers */
    mysqli_stmt_bind_param($stmt, "s", $city);

    /* execute query */
    mysqli_stmt_execute($stmt);

    /* bind result variables */
    mysqli_stmt_bind_result($stmt, $district);

    /* fetch value */
    mysqli_stmt_fetch($stmt);

    printf("%s is in district %s\n", $city, $district);

    /* close statement */
    mysqli_stmt_close($stmt);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Amersfoort is in district Utrecht
```

See Also

mysqli_stmt_execute
mysqli_stmt_fetch

mysqli_stmt_bind_param
mysqli_stmt_bind_result
mysqli_stmt_close

3.9.40 mysqli::query, mysqli_query

- mysqli::query

  mysqli_query

  Performs a query on the database

Description

Object oriented style

```
mixed mysqli::query(
  string query,
  int resultmode
      =  =MYSQLI_STORE_RESULT);
```

Procedural style

```
mixed mysqli_query(
  mysqli link,
  string query,
  int resultmode
      =  =MYSQLI_STORE_RESULT);
```

Performs a *query* against the database.

For non-DML queries (not INSERT, UPDATE or DELETE), this function is similar to calling mysqli_real_query followed by either mysqli_use_result or mysqli_store_result.

> **Note**
>
> In the case where you pass a statement to mysqli_query that is longer than max_allowed_packet of the server, the returned error codes are different depending on whether you are using MySQL Native Driver (mysqlnd) or MySQL Client Library (libmysqlclient). The behavior is as follows:
>
> - mysqlnd on Linux returns an error code of 1153. The error message means “got a packet bigger than max_allowed_packet bytes”.
> - mysqlnd on Windows returns an error code 2006. This error message means “server has gone away”.
> - libmysqlclient on all platforms returns an error code 2006. This error message means “server has gone away”.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
query	The query string. Data inside the query should be properly escaped.

resultmode

Either the constant MYSQLI_USE_RESULT or MYSQLI_STORE_RESULT depending on the desired behavior. By default, MYSQLI_STORE_RESULT is used.

If you use MYSQLI_USE_RESULT all subsequent calls will return error Commands out of sync unless you call mysqli_free_result

With MYSQLI_ASYNC (available with mysqlnd), it is possible to perform query asynchronously. mysqli_poll is then used to get results from such queries.

Return Values

Returns FALSE on failure. For successful SELECT, SHOW, DESCRIBE or EXPLAIN queries mysqli_query will return a mysqli_result object. For other successful queries mysqli_query will return TRUE.

Changelog

Version	Description
5.3.0	Added the ability of async queries.

Examples

Example 3.63 mysqli::query example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if ($mysqli->connect_errno) {
    printf("Connect failed: %s\n", $mysqli->connect_error);
    exit();
}

/* Create table doesn't return a resultset */
if ($mysqli->query("CREATE TEMPORARY TABLE myCity LIKE City") === TRUE) {
    printf("Table myCity successfully created.\n");
}

/* Select queries return a resultset */
if ($result = $mysqli->query("SELECT Name FROM City LIMIT 10")) {
    printf("Select returned %d rows.\n", $result->num_rows);

    /* free result set */
    $result->close();
}

/* If we have to retrieve large amount of data we use MYSQLI_USE_RESULT */
if ($result = $mysqli->query("SELECT * FROM City", MYSQLI_USE_RESULT)) {

    /* Note, that we can't execute any functions which interact with the
       server until result set was closed. All calls will return an
       'out of sync' error */
    if (!$mysqli->query("SET @a:='this will not work'")) {
        printf("Error: %s\n", $mysqli->error);
    }
```

```
    $result->close();
}

$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* Create table doesn't return a resultset */
if (mysqli_query($link, "CREATE TEMPORARY TABLE myCity LIKE City") === TRUE) {
    printf("Table myCity successfully created.\n");
}

/* Select queries return a resultset */
if ($result = mysqli_query($link, "SELECT Name FROM City LIMIT 10")) {
    printf("Select returned %d rows.\n", mysqli_num_rows($result));

    /* free result set */
    mysqli_free_result($result);
}

/* If we have to retrieve large amount of data we use MYSQLI_USE_RESULT */
if ($result = mysqli_query($link, "SELECT * FROM City", MYSQLI_USE_RESULT)) {

    /* Note, that we can't execute any functions which interact with the
       server until result set was closed. All calls will return an
       'out of sync' error */
    if (!mysqli_query($link, "SET @a:='this will not work'")) {
        printf("Error: %s\n", mysqli_error($link));
    }
    mysqli_free_result($result);
}

mysqli_close($link);
?>
```

The above examples will output:

```
Table myCity successfully created.
Select returned 10 rows.
Error: Commands out of sync;  You can't run this command now
```

See Also

mysqli_real_query
mysqli_multi_query
mysqli_free_result

3.9.41 mysqli::real_connect, mysqli_real_connect

- mysqli::real_connect

  mysqli_real_connect

  Opens a connection to a mysql server

Description

Object oriented style

```
bool mysqli::real_connect(
  string host,
  string username,
  string passwd,
  string dbname,
  int port,
  string socket,
  int flags);
```

Procedural style

```
bool mysqli_real_connect(
  mysqli link,
  string host,
  string username,
  string passwd,
  string dbname,
  int port,
  string socket,
  int flags);
```

Establish a connection to a MySQL database engine.

This function differs from mysqli_connect:

- mysqli_real_connect needs a valid object which has to be created by function mysqli_init.
- With the mysqli_options function you can set various options for connection.
- There is a *flags* parameter.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
host	Can be either a host name or an IP address. Passing the NULL value or the string "localhost" to this parameter, the local host is assumed. When possible, pipes will be used instead of the TCP/IP protocol.
username	The MySQL user name.
passwd	If provided or NULL, the MySQL server will attempt to authenticate the user against those user records which have no password only. This allows one username to be used with different permissions (depending on if a password as provided or not).

dbname If provided will specify the default database to be used when performing queries.

port Specifies the port number to attempt to connect to the MySQL server.

socket Specifies the socket or named pipe that should be used.

> **Note**
>
> Specifying the *socket* parameter will not explicitly determine the type of connection to be used when connecting to the MySQL server. How the connection is made to the MySQL database is determined by the *host* parameter.

flags With the parameter *flags* you can set different connection options:

Table 3.11 Supported flags

Name	Description
MYSQLI_CLIENT_COMPRESS	Use compression protocol
MYSQLI_CLIENT_FOUND_ROWS	return number of matched rows, not the number of affected rows
MYSQLI_CLIENT_IGNORE_SPACE	Allow spaces after function names. Makes all function names reserved words.
MYSQLI_CLIENT_INTERACTIVE	Allow interactive_timeout seconds (instead of wait_timeout seconds) of inactivity before closing the connection
MYSQLI_CLIENT_SSL	Use SSL (encryption)
MYSQLI_CLIENT_SSL_DONT_VERIFY_SERVER_CERT	Like MYSQLI_CLIENT_SSL, but disables validation of the provided SSL certificate. This is only for installations using MySQL Native Driver and MySQL 5.6 or later.

> **Note**
>
> For security reasons the MULTI_STATEMENT flag is not supported in PHP. If you want to execute multiple queries use the mysqli_multi_query function.

Changelog

Version	Description
5.6.16	Added the MYSQLI_CLIENT_SSL_DONT_VERIFY_SERVER_CERT flag for MySQL Native Driver

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.64 `mysqli::real_connect` example

Object oriented style

```
<?php

$mysqli = mysqli_init();
if (!$mysqli) {
    die('mysqli_init failed');
}

if (!$mysqli->options(MYSQLI_INIT_COMMAND, 'SET AUTOCOMMIT = 0')) {
    die('Setting MYSQLI_INIT_COMMAND failed');
}

if (!$mysqli->options(MYSQLI_OPT_CONNECT_TIMEOUT, 5)) {
    die('Setting MYSQLI_OPT_CONNECT_TIMEOUT failed');
}

if (!$mysqli->real_connect('localhost', 'my_user', 'my_password', 'my_db')) {
    die('Connect Error (' . mysqli_connect_errno() . ') '
            . mysqli_connect_error());
}

echo 'Success... ' . $mysqli->host_info . "\n";

$mysqli->close();
?>
```

Object oriented style when extending mysqli class

```
<?php

class foo_mysqli extends mysqli {
    public function __construct($host, $user, $pass, $db) {
        parent::init();

        if (!parent::options(MYSQLI_INIT_COMMAND, 'SET AUTOCOMMIT = 0')) {
            die('Setting MYSQLI_INIT_COMMAND failed');
        }

        if (!parent::options(MYSQLI_OPT_CONNECT_TIMEOUT, 5)) {
            die('Setting MYSQLI_OPT_CONNECT_TIMEOUT failed');
        }

        if (!parent::real_connect($host, $user, $pass, $db)) {
            die('Connect Error (' . mysqli_connect_errno() . ') '
                    . mysqli_connect_error());
        }
    }
}

$db = new foo_mysqli('localhost', 'my_user', 'my_password', 'my_db');

echo 'Success... ' . $db->host_info . "\n";
```

```
$db->close();
?>
```

Procedural style

```
<?php

$link = mysqli_init();
if (!$link) {
    die('mysqli_init failed');
}

if (!mysqli_options($link, MYSQLI_INIT_COMMAND, 'SET AUTOCOMMIT = 0')) {
    die('Setting MYSQLI_INIT_COMMAND failed');
}

if (!mysqli_options($link, MYSQLI_OPT_CONNECT_TIMEOUT, 5)) {
    die('Setting MYSQLI_OPT_CONNECT_TIMEOUT failed');
}

if (!mysqli_real_connect($link, 'localhost', 'my_user', 'my_password', 'my_db')) {
    die('Connect Error (' . mysqli_connect_errno() . ') '
            . mysqli_connect_error());
}

echo 'Success... ' . mysqli_get_host_info($link) . "\n";

mysqli_close($link);
?>
```

The above examples will output:

```
Success... MySQL host info: localhost via TCP/IP
```

Notes

> **Note**
>
> MySQLnd always assumes the server default charset. This charset is sent during connection hand-shake/authentication, which mysqlnd will use.
>
> Libmysqlclient uses the default charset set in the `my.cnf` or by an explicit call to `mysqli_options` prior to calling `mysqli_real_connect`, but after `mysqli_init`.

See Also

`mysqli_connect`
`mysqli_init`
`mysqli_options`
`mysqli_ssl_set`
`mysqli_close`

3.9.42 mysqli::real_escape_string, mysqli_real_escape_string

- mysqli::real_escape_string

  mysqli_real_escape_string

  Escapes special characters in a string for use in an SQL statement, taking into account the current charset of the connection

Description

Object oriented style

```
string mysqli::escape_string(
  string escapestr);
```

```
string mysqli::real_escape_string(
  string escapestr);
```

Procedural style

```
string mysqli_real_escape_string(
  mysqli link,
  string escapestr);
```

This function is used to create a legal SQL string that you can use in an SQL statement. The given string is encoded to an escaped SQL string, taking into account the current character set of the connection.

> **Security: the default character set**
>
> The character set must be set either at the server level, or with the API function mysqli_set_charset for it to affect mysqli_real_escape_string. See the concepts section on character sets for more information.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
escapestr	The string to be escaped. Characters encoded are NUL (ASCII 0), \n, \r, \, ', ", and Control-Z.

Return Values

Returns an escaped string.

Examples

Example 3.65 mysqli::real_escape_string example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");
```

```
/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$mysqli->query("CREATE TEMPORARY TABLE myCity LIKE City");

$city = "'s Hertogenbosch";

/* this query will fail, cause we didn't escape $city */
if (!$mysqli->query("INSERT into myCity (Name) VALUES ('$city')")) {
    printf("Error: %s\n", $mysqli->sqlstate);
}

$city = $mysqli->real_escape_string($city);

/* this query with escaped $city will work */
if ($mysqli->query("INSERT into myCity (Name) VALUES ('$city')")) {
    printf("%d Row inserted.\n", $mysqli->affected_rows);
}

$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

mysqli_query($link, "CREATE TEMPORARY TABLE myCity LIKE City");

$city = "'s Hertogenbosch";

/* this query will fail, cause we didn't escape $city */
if (!mysqli_query($link, "INSERT into myCity (Name) VALUES ('$city')")) {
    printf("Error: %s\n", mysqli_sqlstate($link));
}

$city = mysqli_real_escape_string($link, $city);

/* this query with escaped $city will work */
if (mysqli_query($link, "INSERT into myCity (Name) VALUES ('$city')")) {
    printf("%d Row inserted.\n", mysqli_affected_rows($link));
}

mysqli_close($link);
?>
```

The above examples will output:

```
Error: 42000
```

```
1 Row inserted.
```

Notes

> **Note**
>
> For those accustomed to using mysql_real_escape_string, note that the arguments of mysqli_real_escape_string differ from what mysql_real_escape_string expects. The *link* identifier comes first in mysqli_real_escape_string, whereas the string to be escaped comes first in mysql_real_escape_string.

See Also

mysqli_set_charset
mysqli_character_set_name

3.9.43 mysqli::real_query, mysqli_real_query

- mysqli::real_query

  mysqli_real_query

  Execute an SQL query

Description

Object oriented style

```
bool mysqli::real_query(
  string query);
```

Procedural style

```
bool mysqli_real_query(
  mysqli link,
  string query);
```

Executes a single query against the database whose result can then be retrieved or stored using the mysqli_store_result or mysqli_use_result functions.

In order to determine if a given query should return a result set or not, see mysqli_field_count.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
query	The query, as a string. Data inside the query should be properly escaped.

Return Values

Returns TRUE on success or FALSE on failure.

See Also

mysqli_query
mysqli_store_result
mysqli_use_result

3.9.44 mysqli::reap_async_query, mysqli_reap_async_query

- mysqli::reap_async_query

  mysqli_reap_async_query

  Get result from async query

Description

Object oriented style

```
public mysqli_result mysqli::reap_async_query();
```

Procedural style

```
mysqli_result mysqli_reap_async_query(
  mysqli link);
```

Get result from async query. Available only with mysqlnd.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

Returns mysqli_result in success, FALSE otherwise.

See Also

mysqli_poll

3.9.45 mysqli::refresh, mysqli_refresh

- mysqli::refresh

  mysqli_refresh

  Refreshes

Description

Object oriented style

```
public bool mysqli::refresh(
  int options);
```

Procedural style

```
int mysqli_refresh(
  resource link,
  int options);
```

Flushes tables or caches, or resets the replication server information.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
options	The options to refresh, using the MYSQLI_REFRESH_* constants as documented within the MySQLi constants documentation. See also the official MySQL Refresh documentation.

Return Values

TRUE if the refresh was a success, otherwise FALSE

See Also

mysqli_poll

3.9.46 mysqli::release_savepoint, mysqli_release_savepoint

- mysqli::release_savepoint

  mysqli_release_savepoint

  Removes the named savepoint from the set of savepoints of the current transaction

Description

Object oriented style (method):

```
public bool mysqli::release_savepoint(
  string name);
```

Procedural style:

```
bool mysqli_release_savepoint(
  mysqli link,
  string name);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
name	

Return Values

Returns TRUE on success or FALSE on failure.

See Also

mysqli_rollback

3.9.47 mysqli::rollback, mysqli_rollback

- mysqli::rollback

  mysqli_rollback

  Rolls back current transaction

Description

Object oriented style

```
bool mysqli::rollback(
  int flags,
  string name);
```

Procedural style

```
bool mysqli_rollback(
  mysqli link,
  int flags,
  string name);
```

Rollbacks the current transaction for the database.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
flags	A bitmask of MYSQLI_TRANS_COR_* constants.
name	If provided then ROLLBACK/*name*/ is executed.

Return Values

Returns TRUE on success or FALSE on failure.

Changelog

Version	Description
5.5.0	Added *flags* and *name* parameters.

Examples

Example 3.66 mysqli::rollback example

Object oriented style

```
```

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* disable autocommit */
$mysqli->autocommit(FALSE);

$mysqli->query("CREATE TABLE myCity LIKE City");
$mysqli->query("ALTER TABLE myCity Type=InnoDB");
$mysqli->query("INSERT INTO myCity SELECT * FROM City LIMIT 50");

/* commit insert */
$mysqli->commit();

/* delete all rows */
$mysqli->query("DELETE FROM myCity");

if ($result = $mysqli->query("SELECT COUNT(*) FROM myCity")) {
    $row = $result->fetch_row();
    printf("%d rows in table myCity.\n", $row[0]);
    /* Free result */
    $result->close();
}

/* Rollback */
$mysqli->rollback();

if ($result = $mysqli->query("SELECT COUNT(*) FROM myCity")) {
    $row = $result->fetch_row();
    printf("%d rows in table myCity (after rollback).\n", $row[0]);
    /* Free result */
    $result->close();
}

/* Drop table myCity */
$mysqli->query("DROP TABLE myCity");

$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* disable autocommit */
mysqli_autocommit($link, FALSE);

mysqli_query($link, "CREATE TABLE myCity LIKE City");
mysqli_query($link, "ALTER TABLE myCity Type=InnoDB");
mysqli_query($link, "INSERT INTO myCity SELECT * FROM City LIMIT 50");
```

```
/* commit insert */
mysqli_commit($link);

/* delete all rows */
mysqli_query($link, "DELETE FROM myCity");

if ($result = mysqli_query($link, "SELECT COUNT(*) FROM myCity")) {
    $row = mysqli_fetch_row($result);
    printf("%d rows in table myCity.\n", $row[0]);
    /* Free result */
    mysqli_free_result($result);
}

/* Rollback */
mysqli_rollback($link);

if ($result = mysqli_query($link, "SELECT COUNT(*) FROM myCity")) {
    $row = mysqli_fetch_row($result);
    printf("%d rows in table myCity (after rollback).\n", $row[0]);
    /* Free result */
    mysqli_free_result($result);
}

/* Drop table myCity */
mysqli_query($link, "DROP TABLE myCity");

mysqli_close($link);
?>
```

The above examples will output:

```


0 rows in table myCity.
50 rows in table myCity (after rollback).
```

See Also

mysqli_begin_transaction
mysqli_commit
mysqli_autocommit
mysqli_release_savepoint

3.9.48 mysqli::rpl_query_type, mysqli_rpl_query_type

- mysqli::rpl_query_type

  mysqli_rpl_query_type

  Returns RPL query type

Description

Object oriented style

```
  int mysqli::rpl_query_type(
    string query);
```

Procedural style

```
int mysqli_rpl_query_type(
  mysqli link,
  string query);
```

Returns MYSQLI_RPL_MASTER, MYSQLI_RPL_SLAVE or MYSQLI_RPL_ADMIN depending on a query type. INSERT, UPDATE and similar are *master* queries, SELECT is *slave*, and FLUSH, REPAIR and similar are *admin*.

> **Warning**
>
> This function is currently not documented; only its argument list is available.

> **Warning**
>
> This function has been *DEPRECATED* and *REMOVED* as of PHP 5.3.0.

3.9.49 mysqli::savepoint, mysqli_savepoint

- mysqli::savepoint

  mysqli_savepoint

  Set a named transaction savepoint

Description

Object oriented style (method):

```
public bool mysqli::savepoint(
  string name);
```

Procedural style:

```
bool mysqli_savepoint(
  mysqli link,
  string name);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
name	

Return Values

Returns TRUE on success or FALSE on failure.

See Also

mysqli_commit

3.9.50 mysqli::select_db, mysqli_select_db

- mysqli::select_db

  mysqli_select_db

  Selects the default database for database queries

Description

Object oriented style

```
bool mysqli::select_db(
  string dbname);
```

Procedural style

```
bool mysqli_select_db(
  mysqli link,
  string dbname);
```

Selects the default database to be used when performing queries against the database connection.

> **Note**
>
> This function should only be used to change the default database for the connection. You can select the default database with 4th parameter in mysqli_connect.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
dbname	The database name.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.67 mysqli::select_db example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "test");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* return name of current default database */
if ($result = $mysqli->query("SELECT DATABASE()")) {
    $row = $result->fetch_row();
```

```
    printf("Default database is %s.\n", $row[0]);
    $result->close();
}

/* change db to world db */
$mysqli->select_db("world");

/* return name of current default database */
if ($result = $mysqli->query("SELECT DATABASE()")) {
    $row = $result->fetch_row();
    printf("Default database is %s.\n", $row[0]);
    $result->close();
}

$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "test");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* return name of current default database */
if ($result = mysqli_query($link, "SELECT DATABASE()")) {
    $row = mysqli_fetch_row($result);
    printf("Default database is %s.\n", $row[0]);
    mysqli_free_result($result);
}

/* change db to world db */
mysqli_select_db($link, "world");

/* return name of current default database */
if ($result = mysqli_query($link, "SELECT DATABASE()")) {
    $row = mysqli_fetch_row($result);
    printf("Default database is %s.\n", $row[0]);
    mysqli_free_result($result);
}

mysqli_close($link);
?>
```

The above examples will output:

```
Default database is test.
Default database is world.
```

See Also

mysqli_connect

mysqli_real_connect

3.9.51 mysqli::send_query, mysqli_send_query

- mysqli::send_query

  mysqli_send_query

  Send the query and return

Description

Object oriented style

```
bool mysqli::send_query(
  string query);
```

Procedural style

```
bool mysqli_send_query(
  mysqli link,
  string query);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

> **Warning**
>
> This function has been *DEPRECATED* and *REMOVED* as of PHP 5.3.0.

3.9.52 mysqli::set_charset, mysqli_set_charset

- mysqli::set_charset

  mysqli_set_charset

  Sets the default client character set

Description

Object oriented style

```
bool mysqli::set_charset(
  string charset);
```

Procedural style

```
bool mysqli_set_charset(
  mysqli link,
  string charset);
```

Sets the default character set to be used when sending data from and to the database server.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
charset	The charset to be set as default.

Return Values

Returns TRUE on success or FALSE on failure.

Notes

> **Note**
>
> To use this function on a Windows platform you need MySQL client library version 4.1.11 or above (for MySQL 5.0 you need 5.0.6 or above).

> **Note**
>
> This is the preferred way to change the charset. Using mysqli_query to set it (such as SET NAMES utf8) is not recommended. See the MySQL character set concepts section for more information.

Examples

Example 3.68 mysqli::set_charset example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "test");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

printf("Initial character set: %s\n", $mysqli->character_set_name());

/* change character set to utf8 */
if (!$mysqli->set_charset("utf8")) {
    printf("Error loading character set utf8: %s\n", $mysqli->error);
    exit();
} else {
    printf("Current character set: %s\n", $mysqli->character_set_name());
}

$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect('localhost', 'my_user', 'my_password', 'test');

/* check connection */
if (mysqli_connect_errno()) {
```

```
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

printf("Initial character set: %s\n", mysqli_character_set_name($link));

/* change character set to utf8 */
if (!mysqli_set_charset($link, "utf8")) {
    printf("Error loading character set utf8: %s\n", mysqli_error($link));
    exit();
} else {
    printf("Current character set: %s\n", mysqli_character_set_name($link));
}

mysqli_close($link);
?>
```

The above examples will output something similar to:

```
Initial character set: latin1
Current character set: utf8
```

See Also

mysqli_character_set_name
mysqli_real_escape_string
MySQL character set concepts
List of character sets that MySQL supports

3.9.53 mysqli::set_local_infile_default, mysqli_set_local_infile_default

- mysqli::set_local_infile_default

  mysqli_set_local_infile_default

  Unsets user defined handler for load local infile command

Description

```
void mysqli_set_local_infile_default(
  mysqli link);
```

Deactivates a LOAD DATA INFILE LOCAL handler previously set with mysqli_set_local_infile_handler.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

No value is returned.

Examples

See mysqli_set_local_infile_handler examples

See Also

mysqli_set_local_infile_handler

3.9.54 mysqli::set_local_infile_handler, mysqli_set_local_infile_handler

- mysqli::set_local_infile_handler

  mysqli_set_local_infile_handler

  Set callback function for LOAD DATA LOCAL INFILE command

Description

Object oriented style

```
bool mysqli::set_local_infile_handler(
  mysqli link,
  callable read_func);
```

Procedural style

```
bool mysqli_set_local_infile_handler(
  mysqli link,
  callable read_func);
```

Set callback function for LOAD DATA LOCAL INFILE command

The callbacks task is to read input from the file specified in the LOAD DATA LOCAL INFILE and to reformat it into the format understood by LOAD DATA INFILE.

The returned data needs to match the format specified in the LOAD DATA

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
read_func	A callback function or object method taking the following parameters:
stream	A PHP stream associated with the SQL commands INFILE
&buffer	A string buffer to store the rewritten input into
buflen	The maximum number of characters to be stored in the buffer
&errormsg	If an error occurs you can store an error message in here

The callback function should return the number of characters stored in the *buffer* or a negative value if an error occurred.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.69 mysqli::set_local_infile_handler example

Object oriented style

```
<?php
  $db = mysqli_init();
  $db->real_connect("localhost","root","","test");

  function callme($stream, &$buffer, $buflen, &$errmsg)
  {
    $buffer = fgets($stream);

    echo $buffer;

    // convert to upper case and replace "," delimiter with [TAB]
    $buffer = strtoupper(str_replace(",", "\t", $buffer));

    return strlen($buffer);
  }


  echo "Input:\n";

  $db->set_local_infile_handler("callme");
  $db->query("LOAD DATA LOCAL INFILE 'input.txt' INTO TABLE t1");
  $db->set_local_infile_default();

  $res = $db->query("SELECT * FROM t1");

  echo "\nResult:\n";
  while ($row = $res->fetch_assoc()) {
    echo join(",", $row)."\n";
  }
?>
```

Procedural style

```
<?php
  $db = mysqli_init();
  mysqli_real_connect($db, "localhost","root","","test");

  function callme($stream, &$buffer, $buflen, &$errmsg)
  {
    $buffer = fgets($stream);

    echo $buffer;

    // convert to upper case and replace "," delimiter with [TAB]
    $buffer = strtoupper(str_replace(",", "\t", $buffer));

    return strlen($buffer);
```

```
  }


  echo "Input:\n";

  mysqli_set_local_infile_handler($db, "callme");
  mysqli_query($db, "LOAD DATA LOCAL INFILE 'input.txt' INTO TABLE t1");
  mysqli_set_local_infile_default($db);

  $res = mysqli_query($db, "SELECT * FROM t1");


  echo "\nResult:\n";
  while ($row = mysqli_fetch_assoc($res)) {
    echo join(",", $row)."\n";
  }
?>
```

The above examples will output:

```


Input:
23,foo
42,bar

Output:
23,FOO
42,BAR
```

See Also

mysqli_set_local_infile_default

3.9.55 mysqli::$sqlstate, mysqli_sqlstate

- mysqli::$sqlstate

  mysqli_sqlstate

  Returns the SQLSTATE error from previous MySQL operation

Description

Object oriented style

```
  string
    mysqli->sqlstate ;
```

Procedural style

```
  string mysqli_sqlstate(
    mysqli link);
```

Returns a string containing the SQLSTATE error code for the last error. The error code consists of five characters. '00000' means no error. The values are specified by ANSI SQL and ODBC. For a list of possible values, see http://dev.mysql.com/doc/mysql/en/error-handling.html.

> **Note**
>
> Note that not all MySQL errors are yet mapped to SQLSTATE's. The value `HY000` (general error) is used for unmapped errors.

Parameters

link	Procedural style only: A link identifier returned by `mysqli_connect` or `mysqli_init`

Return Values

Returns a string containing the SQLSTATE error code for the last error. The error code consists of five characters. `'00000'` means no error.

Examples

Example 3.70 `$mysqli->sqlstate` example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* Table City already exists, so we should get an error */
if (!$mysqli->query("CREATE TABLE City (ID INT, Name VARCHAR(30))")) {
    printf("Error - SQLSTATE %s.\n", $mysqli->sqlstate);
}

$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* Table City already exists, so we should get an error */
if (!mysqli_query($link, "CREATE TABLE City (ID INT, Name VARCHAR(30))")) {
    printf("Error - SQLSTATE %s.\n", mysqli_sqlstate($link));
}

mysqli_close($link);
?>
```

The above examples will output:

```
Error - SQLSTATE 42S01.
```

See Also

mysqli_errno
mysqli_error

3.9.56 mysqli::ssl_set, mysqli_ssl_set

- mysqli::ssl_set

  mysqli_ssl_set

  Used for establishing secure connections using SSL

Description

Object oriented style

```
bool mysqli::ssl_set(
  string key,
  string cert,
  string ca,
  string capath,
  string cipher);
```

Procedural style

```
bool mysqli_ssl_set(
  mysqli link,
  string key,
  string cert,
  string ca,
  string capath,
  string cipher);
```

Used for establishing secure connections using SSL. It must be called before mysqli_real_connect. This function does nothing unless OpenSSL support is enabled.

Note that MySQL Native Driver does not support SSL before PHP 5.3.3, so calling this function when using MySQL Native Driver will result in an error. MySQL Native Driver is enabled by default on Microsoft Windows from PHP version 5.3 onwards.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
key	The path name to the key file.
cert	The path name to the certificate file.
ca	The path name to the certificate authority file.

capath	The pathname to a directory that contains trusted SSL CA certificates in PEM format.
cipher	A list of allowable ciphers to use for SSL encryption.

Any unused SSL parameters may be given as NULL

Return Values

This function always returns TRUE value. If SSL setup is incorrect mysqli_real_connect will return an error when you attempt to connect.

See Also

mysqli_options
mysqli_real_connect

3.9.57 mysqli::stat, mysqli_stat

- mysqli::stat

  mysqli_stat

  Gets the current system status

Description

Object oriented style

```
string mysqli::stat();
```

Procedural style

```
string mysqli_stat(
  mysqli link);
```

mysqli_stat returns a string containing information similar to that provided by the 'mysqladmin status' command. This includes uptime in seconds and the number of running threads, questions, reloads, and open tables.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

A string describing the server status. FALSE if an error occurred.

Examples

Example 3.71 mysqli::stat example

Object oriented style

```
```

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

printf ("System status: %s\n", $mysqli->stat());

$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

printf("System status: %s\n", mysqli_stat($link));

mysqli_close($link);
?>
```

The above examples will output:

```
System status: Uptime: 272  Threads: 1  Questions: 5340  Slow queries: 0
Opens: 13  Flush tables: 1  Open tables: 0  Queries per second avg: 19.632
Memory in use: 8496K  Max memory used: 8560K
```

See Also

mysqli_get_server_info

3.9.58 mysqli::stmt_init, mysqli_stmt_init

- mysqli::stmt_init

  mysqli_stmt_init

  Initializes a statement and returns an object for use with mysqli_stmt_prepare

Description

Object oriented style

```
mysqli_stmt mysqli::stmt_init();
```

Procedural style

```
mysqli_stmt mysqli_stmt_init(
  mysqli link);
```

Allocates and initializes a statement object suitable for mysqli_stmt_prepare.

> **Note**
>
> Any subsequent calls to any mysqli_stmt function will fail until mysqli_stmt_prepare was called.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

Returns an object.

See Also

mysqli_stmt_prepare

3.9.59 mysqli::store_result, mysqli_store_result

- mysqli::store_result

  mysqli_store_result

  Transfers a result set from the last query

Description

Object oriented style

```
mysqli_result mysqli::store_result(
  int option);
```

Procedural style

```
mysqli_result mysqli_store_result(
  mysqli link,
  int option);
```

Transfers the result set from the last query on the database connection represented by the *link* parameter to be used with the mysqli_data_seek function.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init
option	The option that you want to set. It can be one of the following values:

Table 3.12 Valid options

Name	Description
MYSQLI_STORE_RESULT_COPY_DATA	Copy results from the internal mysqlnd buffer into the PHP variables fetched. By default, mysqlnd will use a reference logic to avoid copying and duplicating results held in memory. For certain result sets, for example, result sets with many small rows, the copy approach can reduce the overall memory usage because PHP variables holding results may be released earlier (available with mysqlnd only, since PHP 5.6.0)

Return Values

Returns a buffered result object or FALSE if an error occurred.

> **Note**
>
> mysqli_store_result returns FALSE in case the query didn't return a result set (if the query was, for example an INSERT statement). This function also returns FALSE if the reading of the result set failed. You can check if you have got an error by checking if mysqli_error doesn't return an empty string, if mysqli_errno returns a non zero value, or if mysqli_field_count returns a non zero value. Also possible reason for this function returning FALSE after successful call to mysqli_query can be too large result set (memory for it cannot be allocated). If mysqli_field_count returns a non-zero value, the statement should have produced a non-empty result set.

Notes

> **Note**
>
> Although it is always good practice to free the memory used by the result of a query using the mysqli_free_result function, when transferring large result sets using the mysqli_store_result this becomes particularly important.

Examples

See mysqli_multi_query.

See Also

mysqli_real_query
mysqli_use_result

3.9.60 mysqli::$thread_id, mysqli_thread_id

- mysqli::$thread_id

mysqli_thread_id

Returns the thread ID for the current connection

Description

Object oriented style

```
int
  mysqli->thread_id ;
```

Procedural style

```
int mysqli_thread_id(
  mysqli link);
```

The mysqli_thread_id function returns the thread ID for the current connection which can then be killed using the mysqli_kill function. If the connection is lost and you reconnect with mysqli_ping, the thread ID will be other. Therefore you should get the thread ID only when you need it.

> **Note**
>
> The thread ID is assigned on a connection-by-connection basis. Hence, if the connection is broken and then re-established a new thread ID will be assigned.
>
> To kill a running query you can use the SQL command KILL QUERY processid.

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

Returns the Thread ID for the current connection.

Examples

Example 3.72 $mysqli->thread_id example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* determine our thread id */
$thread_id = $mysqli->thread_id;

/* Kill connection */
$mysqli->kill($thread_id);

/* This should produce an error */
if (!$mysqli->query("CREATE TABLE myCity LIKE City")) {
```

```
    printf("Error: %s\n", $mysqli->error);
    exit;
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* determine our thread id */
$thread_id = mysqli_thread_id($link);

/* Kill connection */
mysqli_kill($link, $thread_id);

/* This should produce an error */
if (!mysqli_query($link, "CREATE TABLE myCity LIKE City")) {
    printf("Error: %s\n", mysqli_error($link));
    exit;
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Error: MySQL server has gone away
```

See Also

mysqli_kill

3.9.61 mysqli::thread_safe, mysqli_thread_safe

- mysqli::thread_safe

  mysqli_thread_safe

  Returns whether thread safety is given or not

Description

Procedural style

```
bool mysqli_thread_safe();
```

Tells whether the client library is compiled as thread-safe.

Return Values

TRUE if the client library is thread-safe, otherwise FALSE.

3.9.62 mysqli::use_result, mysqli_use_result

- mysqli::use_result

  mysqli_use_result

  Initiate a result set retrieval

Description

Object oriented style

```
mysqli_result mysqli::use_result();
```

Procedural style

```
mysqli_result mysqli_use_result(
  mysqli link);
```

Used to initiate the retrieval of a result set from the last query executed using the mysqli_real_query function on the database connection.

Either this or the mysqli_store_result function must be called before the results of a query can be retrieved, and one or the other must be called to prevent the next query on that database connection from failing.

> **Note**
>
> The mysqli_use_result function does not transfer the entire result set from the database and hence cannot be used functions such as mysqli_data_seek to move to a particular row within the set. To use this functionality, the result set must be stored using mysqli_store_result. One should not use mysqli_use_result if a lot of processing on the client side is performed, since this will tie up the server and prevent other threads from updating any tables from which the data is being fetched.

Return Values

Returns an unbuffered result object or FALSE if an error occurred.

Examples

Example 3.73 mysqli::use_result example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query  = "SELECT CURRENT_USER();";
$query .= "SELECT Name FROM City ORDER BY ID LIMIT 20, 5";

/* execute multi query */
if ($mysqli->multi_query($query)) {
    do {
        /* store first result set */
        if ($result = $mysqli->use_result()) {
            while ($row = $result->fetch_row()) {
                printf("%s\n", $row[0]);
            }
            $result->close();
        }
        /* print divider */
        if ($mysqli->more_results()) {
            printf("-----------------\n");
        }
    } while ($mysqli->next_result());
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query  = "SELECT CURRENT_USER();";
$query .= "SELECT Name FROM City ORDER BY ID LIMIT 20, 5";

/* execute multi query */
if (mysqli_multi_query($link, $query)) {
    do {
        /* store first result set */
        if ($result = mysqli_use_result($link)) {
            while ($row = mysqli_fetch_row($result)) {
                printf("%s\n", $row[0]);
            }
            mysqli_free_result($result);
        }
        /* print divider */
        if (mysqli_more_results($link)) {
            printf("-----------------\n");
        }
```

```
    } while (mysqli_next_result($link));
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
my_user@localhost
-----------------
Amersfoort
Maastricht
Dordrecht
Leiden
Haarlemmermeer
```

See Also

mysqli_real_query
mysqli_store_result

3.9.63 mysqli::$warning_count, mysqli_warning_count

- mysqli::$warning_count

  mysqli_warning_count

  Returns the number of warnings from the last query for the given link

Description

Object oriented style

```
  int
    mysqli->warning_count ;
```

Procedural style

```
  int mysqli_warning_count(
    mysqli link);
```

Returns the number of warnings from the last query in the connection.

> **Note**
>
> For retrieving warning messages you can use the SQL command SHOW WARNINGS [limit row_count].

Parameters

link	Procedural style only: A link identifier returned by mysqli_connect or mysqli_init

Return Values

Number of warnings or zero if there are no warnings.

Examples

Example 3.74 `$mysqli->warning_count` example

Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$mysqli->query("CREATE TABLE myCity LIKE City");

/* a remarkable city in Wales */
$query = "INSERT INTO myCity (CountryCode, Name) VALUES('GBR',
        'Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch')";

$mysqli->query($query);

if ($mysqli->warning_count) {
    if ($result = $mysqli->query("SHOW WARNINGS")) {
        $row = $result->fetch_row();
        printf("%s (%d): %s\n", $row[0], $row[1], $row[2]);
        $result->close();
    }
}

/* close connection */
$mysqli->close();
?>
```

Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

mysqli_query($link, "CREATE TABLE myCity LIKE City");

/* a remarkable long city name in Wales */
$query = "INSERT INTO myCity (CountryCode, Name) VALUES('GBR',
        'Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch')";

mysqli_query($link, $query);

if (mysqli_warning_count($link)) {
    if ($result = mysqli_query($link, "SHOW WARNINGS")) {
```

```
        $row = mysqli_fetch_row($result);
        printf("%s (%d): %s\n", $row[0], $row[1], $row[2]);
        mysqli_free_result($result);
    }
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Warning (1264): Data truncated for column 'Name' at row 1
```

See Also

mysqli_errno
mysqli_error
mysqli_sqlstate

3.10 The mysqli_stmt class

Represents a prepared statement.

```
mysqli_stmt {
mysqli_stmt

      Properties

  int
    mysqli_stmt->affected_rows ;

  int
    mysqli_stmt->errno ;

  array
    mysqli_stmt->error_list ;

  string
    mysqli_stmt->error ;

  int
    mysqli_stmt->field_count ;

  int
    mysqli_stmt->insert_id ;

  int
    mysqli_stmt->num_rows ;

  int
    mysqli_stmt->param_count ;

  string
    mysqli_stmt->sqlstate ;
```

```
Methods

  mysqli_stmt::__construct(
    mysqli link,
    string query);

  int mysqli_stmt::attr_get(
    int attr);

  bool mysqli_stmt::attr_set(
    int attr,
    int mode);

  bool mysqli_stmt::bind_param(
    string types,
    mixed var1,
    mixed ...);

  bool mysqli_stmt::bind_result(
    mixed var1,
    mixed ...);

  bool mysqli_stmt::close();

  void mysqli_stmt::data_seek(
    int offset);

  bool mysqli_stmt::execute();

  bool mysqli_stmt::fetch();

  void mysqli_stmt::free_result();

  mysqli_result mysqli_stmt::get_result();

  object mysqli_stmt::get_warnings(
    mysqli_stmt stmt);

  mixed mysqli_stmt::prepare(
    string query);

  bool mysqli_stmt::reset();

  mysqli_result mysqli_stmt::result_metadata();

  bool mysqli_stmt::send_long_data(
    int param_nr,
    string data);

  bool mysqli_stmt::store_result();

}
```

3.10.1 mysqli_stmt::$affected_rows, mysqli_stmt_affected_rows

- mysqli_stmt::$affected_rows

  mysqli_stmt_affected_rows

  Returns the total number of rows changed, deleted, or inserted by the last executed statement

Description

Object oriented style

```
int
  mysqli_stmt->affected_rows ;
```

Procedural style

```
int mysqli_stmt_affected_rows(
  mysqli_stmt stmt);
```

Returns the number of rows affected by INSERT, UPDATE, or DELETE query.

This function only works with queries which update a table. In order to get the number of rows from a SELECT query, use mysqli_stmt_num_rows instead.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

An integer greater than zero indicates the number of rows affected or retrieved. Zero indicates that no records where updated for an UPDATE/DELETE statement, no rows matched the WHERE clause in the query or that no query has yet been executed. -1 indicates that the query has returned an error. NULL indicates an invalid argument was supplied to the function.

> **Note**
>
> If the number of affected rows is greater than maximal PHP int value, the number of affected rows will be returned as a string value.

Examples

Example 3.75 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* create temp table */
$mysqli->query("CREATE TEMPORARY TABLE myCountry LIKE Country");

$query = "INSERT INTO myCountry SELECT * FROM Country WHERE Code LIKE ?";

/* prepare statement */
if ($stmt = $mysqli->prepare($query)) {

    /* Bind variable for placeholder */
    $code = 'A%';
    $stmt->bind_param("s", $code);

    /* execute statement */
    $stmt->execute();

    printf("rows inserted: %d\n", $stmt->affected_rows);

    /* close statement */
```

```
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.76 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* create temp table */
mysqli_query($link, "CREATE TEMPORARY TABLE myCountry LIKE Country");

$query = "INSERT INTO myCountry SELECT * FROM Country WHERE Code LIKE ?";

/* prepare statement */
if ($stmt = mysqli_prepare($link, $query)) {

    /* Bind variable for placeholder */
    $code = 'A%';
    mysqli_stmt_bind_param($stmt, "s", $code);

    /* execute statement */
    mysqli_stmt_execute($stmt);

    printf("rows inserted: %d\n", mysqli_stmt_affected_rows($stmt));

    /* close statement */
    mysqli_stmt_close($stmt);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
rows inserted: 17
```

See Also

mysqli_stmt_num_rows
mysqli_prepare

3.10.2 mysqli_stmt::attr_get, mysqli_stmt_attr_get

- mysqli_stmt::attr_get

  mysqli_stmt_attr_get

  Used to get the current value of a statement attribute

Description

Object oriented style

```
int mysqli_stmt::attr_get(
  int attr);
```

Procedural style

```
int mysqli_stmt_attr_get(
  mysqli_stmt stmt,
  int attr);
```

Gets the current value of a statement attribute.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.
attr	The attribute that you want to get.

Return Values

Returns FALSE if the attribute is not found, otherwise returns the value of the attribute.

3.10.3 mysqli_stmt::attr_set, mysqli_stmt_attr_set

- mysqli_stmt::attr_set

  mysqli_stmt_attr_set

  Used to modify the behavior of a prepared statement

Description

Object oriented style

```
bool mysqli_stmt::attr_set(
  int attr,
  int mode);
```

Procedural style

```
bool mysqli_stmt_attr_set(
  mysqli_stmt stmt,
  int attr,
  int mode);
```

Used to modify the behavior of a prepared statement. This function may be called multiple times to set several attributes.

Parameters

stmt — Procedural style only: A statement identifier returned by mysqli_stmt_init.

attr — The attribute that you want to set. It can have one of the following values:

Table 3.13 Attribute values

Character	Description
MYSQLI_STMT_ATTR_UPDATE_MAX_LENGTH	Setting to TRUE causes mysqli_stmt_store_result to update the metadata MYSQL_FIELD->max_length value.
MYSQLI_STMT_ATTR_CURSOR_TYPE	Type of cursor to open for statement when mysqli_stmt_execute is invoked. *mode* can be MYSQLI_CURSOR_TYPE_NO_CURSOR (the default) or MYSQLI_CURSOR_TYPE_READ_ONLY.
MYSQLI_STMT_ATTR_PREFETCH_ROWS	Number of rows to fetch from server at a time when using a cursor. *mode* can be in the range from 1 to the maximum value of unsigned long. The default is 1.

If you use the MYSQLI_STMT_ATTR_CURSOR_TYPE option with MYSQLI_CURSOR_TYPE_READ_ONLY, a cursor is opened for the statement when you invoke mysqli_stmt_execute. If there is already an open cursor from a previous mysqli_stmt_execute call, it closes the cursor before opening a new one. mysqli_stmt_reset also closes any open cursor before preparing the statement for re-execution. mysqli_stmt_free_result closes any open cursor.

If you open a cursor for a prepared statement, mysqli_stmt_store_result is unnecessary.

mode — The value to assign to the attribute.

See Also

Connector/MySQL mysql_stmt_attr_set()

3.10.4 mysqli_stmt::bind_param, mysqli_stmt_bind_param

- mysqli_stmt::bind_param

  mysqli_stmt_bind_param

  Binds variables to a prepared statement as parameters

Description

Object oriented style

```
bool mysqli_stmt::bind_param(
  string types,
  mixed var1,
  mixed ...);
```

Procedural style

```
bool mysqli_stmt_bind_param(
  mysqli_stmt stmt,
  string types,
  mixed var1,
  mixed ...);
```

Bind variables for the parameter markers in the SQL statement that was passed to `mysqli_prepare`.

> **Note**
>
> If data size of a variable exceeds max. allowed packet size (max_allowed_packet), you have to specify `b` in *`types`* and use `mysqli_stmt_send_long_data` to send the data in packets.

> **Note**
>
> Care must be taken when using `mysqli_stmt_bind_param` in conjunction with `call_user_func_array`. Note that `mysqli_stmt_bind_param` requires parameters to be passed by reference, whereas `call_user_func_array` can accept as a parameter a list of variables that can represent references or values.

Parameters

`stmt`
Procedural style only: A statement identifier returned by `mysqli_stmt_init`.

`types`
A string that contains one or more characters which specify the types for the corresponding bind variables:

Table 3.14 Type specification chars

Character	Description
i	corresponding variable has type integer
d	corresponding variable has type double
s	corresponding variable has type string
b	corresponding variable is a blob and will be sent in packets

`var1`
The number of variables and length of string *`types`* must match the parameters in the statement.

Return Values

Returns `TRUE` on success or `FALSE` on failure.

Examples

Example 3.77 Object oriented style

```
<?php
$mysqli = new mysqli('localhost', 'my_user', 'my_password', 'world');

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$stmt = $mysqli->prepare("INSERT INTO CountryLanguage VALUES (?, ?, ?, ?)");
$stmt->bind_param('sssd', $code, $language, $official, $percent);

$code = 'DEU';
$language = 'Bavarian';
$official = "F";
$percent = 11.2;

/* execute prepared statement */
$stmt->execute();

printf("%d Row inserted.\n", $stmt->affected_rows);

/* close statement and connection */
$stmt->close();

/* Clean up table CountryLanguage */
$mysqli->query("DELETE FROM CountryLanguage WHERE Language='Bavarian'");
printf("%d Row deleted.\n", $mysqli->affected_rows);

/* close connection */
$mysqli->close();
?>
```

Example 3.78 Procedural style

```
<?php
$link = mysqli_connect('localhost', 'my_user', 'my_password', 'world');

/* check connection */
if (!$link) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$stmt = mysqli_prepare($link, "INSERT INTO CountryLanguage VALUES (?, ?, ?, ?)");
mysqli_stmt_bind_param($stmt, 'sssd', $code, $language, $official, $percent);

$code = 'DEU';
$language = 'Bavarian';
$official = "F";
$percent = 11.2;

/* execute prepared statement */
mysqli_stmt_execute($stmt);

printf("%d Row inserted.\n", mysqli_stmt_affected_rows($stmt));
```

```
/* close statement and connection */
mysqli_stmt_close($stmt);

/* Clean up table CountryLanguage */
mysqli_query($link, "DELETE FROM CountryLanguage WHERE Language='Bavarian'");
printf("%d Row deleted.\n", mysqli_affected_rows($link));

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
1 Row inserted.
1 Row deleted.
```

See Also

mysqli_stmt_bind_result
mysqli_stmt_execute
mysqli_stmt_fetch
mysqli_prepare
mysqli_stmt_send_long_data
mysqli_stmt_errno
mysqli_stmt_error

3.10.5 mysqli_stmt::bind_result, mysqli_stmt_bind_result

- mysqli_stmt::bind_result

  mysqli_stmt_bind_result

  Binds variables to a prepared statement for result storage

Description

Object oriented style

```
bool mysqli_stmt::bind_result(
  mixed var1,
  mixed ...);
```

Procedural style

```
bool mysqli_stmt_bind_result(
  mysqli_stmt stmt,
  mixed var1,
  mixed ...);
```

Binds columns in the result set to variables.

When mysqli_stmt_fetch is called to fetch data, the MySQL client/server protocol places the data for the bound columns into the specified variables *var1, ...*.

> **Note**
>
> Note that all columns must be bound after mysqli_stmt_execute and prior to calling mysqli_stmt_fetch. Depending on column types bound variables can silently change to the corresponding PHP type.
>
> A column can be bound or rebound at any time, even after a result set has been partially retrieved. The new binding takes effect the next time mysqli_stmt_fetch is called.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.
var1	The variable to be bound.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.79 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* prepare statement */
if ($stmt = $mysqli->prepare("SELECT Code, Name FROM Country ORDER BY Name LIMIT 5")) {
    $stmt->execute();

    /* bind variables to prepared statement */
    $stmt->bind_result($col1, $col2);

    /* fetch values */
    while ($stmt->fetch()) {
        printf("%s %s\n", $col1, $col2);
    }

    /* close statement */
    $stmt->close();
}
/* close connection */
$mysqli->close();

?>
```

Example 3.80 Procedural style

```
<?php
```

```
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (!$link) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* prepare statement */
if ($stmt = mysqli_prepare($link, "SELECT Code, Name FROM Country ORDER BY Name LIMIT 5")) {
    mysqli_stmt_execute($stmt);

    /* bind variables to prepared statement */
    mysqli_stmt_bind_result($stmt, $col1, $col2);

    /* fetch values */
    while (mysqli_stmt_fetch($stmt)) {
        printf("%s %s\n", $col1, $col2);
    }

    /* close statement */
    mysqli_stmt_close($stmt);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
AFG Afghanistan
ALB Albania
DZA Algeria
ASM American Samoa
AND Andorra
```

See Also

mysqli_stmt_get_result
mysqli_stmt_bind_param
mysqli_stmt_execute
mysqli_stmt_fetch
mysqli_prepare
mysqli_stmt_prepare
mysqli_stmt_init
mysqli_stmt_errno
mysqli_stmt_error

3.10.6 mysqli_stmt::close, mysqli_stmt_close

- mysqli_stmt::close

  mysqli_stmt_close

  Closes a prepared statement

Description

Object oriented style

```
bool mysqli_stmt::close();
```

Procedural style

```
bool mysqli_stmt_close(
  mysqli_stmt stmt);
```

Closes a prepared statement. `mysqli_stmt_close` also deallocates the statement handle. If the current statement has pending or unread results, this function cancels them so that the next query can be executed.

Parameters

`stmt`	Procedural style only: A statement identifier returned by `mysqli_stmt_init`.

Return Values

Returns `TRUE` on success or `FALSE` on failure.

See Also

`mysqli_prepare`

3.10.7 `mysqli_stmt::__construct`

- `mysqli_stmt::__construct`

  Constructs a new `mysqli_stmt` object

Description

```
mysqli_stmt::__construct(
  mysqli link,
  string query);
```

This method constructs a new `mysqli_stmt` object.

> **Note**
>
> In general, you should use either `mysqli_prepare` or `mysqli_stmt_init` to create a `mysqli_stmt` object, rather than directly instantiating the object with `new mysqli_stmt`. This method (and the ability to directly instantiate `mysqli_stmt` objects) may be deprecated and removed in the future.

Parameters

`link`	Procedural style only: A link identifier returned by `mysqli_connect` or `mysqli_init`
`query`	The query, as a string. If this parameter is omitted, then the constructor behaves identically to `mysqli_stmt_init`, if provided, then it behaves as per `mysqli_prepare`.

See Also

mysqli_prepare
mysqli_stmt_init

3.10.8 mysqli_stmt::data_seek, mysqli_stmt_data_seek

- mysqli_stmt::data_seek

  mysqli_stmt_data_seek

  Seeks to an arbitrary row in statement result set

Description

Object oriented style

```
void mysqli_stmt::data_seek(
  int offset);
```

Procedural style

```
void mysqli_stmt_data_seek(
  mysqli_stmt stmt,
  int offset);
```

Seeks to an arbitrary result pointer in the statement result set.

mysqli_stmt_store_result must be called prior to mysqli_stmt_data_seek.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.
offset	Must be between zero and the total number of rows minus one (0.. mysqli_stmt_num_rows - 1).

Return Values

No value is returned.

Examples

Example 3.81 Object oriented style

```
<?php
/* Open a connection */
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER BY Name";
```

```
if ($stmt = $mysqli->prepare($query)) {

    /* execute query */
    $stmt->execute();

    /* bind result variables */
    $stmt->bind_result($name, $code);

    /* store result */
    $stmt->store_result();

    /* seek to row no. 400 */
    $stmt->data_seek(399);

    /* fetch values */
    $stmt->fetch();

    printf ("City: %s  Countrycode: %s\n", $name, $code);

    /* close statement */
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.82 Procedural style

```
<?php
/* Open a connection */
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER BY Name";
if ($stmt = mysqli_prepare($link, $query)) {

    /* execute query */
    mysqli_stmt_execute($stmt);

    /* bind result variables */
    mysqli_stmt_bind_result($stmt, $name, $code);

    /* store result */
    mysqli_stmt_store_result($stmt);

    /* seek to row no. 400 */
    mysqli_stmt_data_seek($stmt, 399);

    /* fetch values */
    mysqli_stmt_fetch($stmt);

    printf ("City: %s  Countrycode: %s\n", $name, $code);

    /* close statement */
    mysqli_stmt_close($stmt);
}
```

```
/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
City: Benin City  Countrycode: NGA
```

See Also

mysqli_prepare

3.10.9 mysqli_stmt::$errno, mysqli_stmt_errno

- mysqli_stmt::$errno

  mysqli_stmt_errno

  Returns the error code for the most recent statement call

Description

Object oriented style

```
int
  mysqli_stmt->errno ;
```

Procedural style

```
int mysqli_stmt_errno(
  mysqli_stmt stmt);
```

Returns the error code for the most recently invoked statement function that can succeed or fail.

Client error message numbers are listed in the MySQL errmsg.h header file, server error message numbers are listed in mysqld_error.h. In the MySQL source distribution you can find a complete list of error messages and error numbers in the file Docs/mysqld_error.txt.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

An error code value. Zero means no error occurred.

Examples

Example 3.83 Object oriented style

```
```

```
<?php
/* Open a connection */
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$mysqli->query("CREATE TABLE myCountry LIKE Country");
$mysqli->query("INSERT INTO myCountry SELECT * FROM Country");


$query = "SELECT Name, Code FROM myCountry ORDER BY Name";
if ($stmt = $mysqli->prepare($query)) {

    /* drop table */
    $mysqli->query("DROP TABLE myCountry");

    /* execute query */
    $stmt->execute();

    printf("Error: %d.\n", $stmt->errno);

    /* close statement */
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.84 Procedural style

```
<?php
/* Open a connection */
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

mysqli_query($link, "CREATE TABLE myCountry LIKE Country");
mysqli_query($link, "INSERT INTO myCountry SELECT * FROM Country");


$query = "SELECT Name, Code FROM myCountry ORDER BY Name";
if ($stmt = mysqli_prepare($link, $query)) {

    /* drop table */
    mysqli_query($link, "DROP TABLE myCountry");

    /* execute query */
    mysqli_stmt_execute($stmt);

    printf("Error: %d.\n", mysqli_stmt_errno($stmt));

    /* close statement */
    mysqli_stmt_close($stmt);
}
```

```
/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Error: 1146.
```

See Also

mysqli_stmt_error
mysqli_stmt_sqlstate

3.10.10 mysqli_stmt::$error_list, mysqli_stmt_error_list

- mysqli_stmt::$error_list

  mysqli_stmt_error_list

  Returns a list of errors from the last statement executed

Description

Object oriented style

```
array
  mysqli_stmt->error_list ;
```

Procedural style

```
array mysqli_stmt_error_list(
  mysqli_stmt stmt);
```

Returns an array of errors for the most recently invoked statement function that can succeed or fail.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

A list of errors, each as an associative array containing the errno, error, and sqlstate.

Examples

Example 3.85 Object oriented style

```
<?php
/* Open a connection */
```

```
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$mysqli->query("CREATE TABLE myCountry LIKE Country");
$mysqli->query("INSERT INTO myCountry SELECT * FROM Country");


$query = "SELECT Name, Code FROM myCountry ORDER BY Name";
if ($stmt = $mysqli->prepare($query)) {

    /* drop table */
    $mysqli->query("DROP TABLE myCountry");

    /* execute query */
    $stmt->execute();

    echo "Error:\n";
    print_r($stmt->error_list);

    /* close statement */
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.86 Procedural style

```


<?php
/* Open a connection */
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

mysqli_query($link, "CREATE TABLE myCountry LIKE Country");
mysqli_query($link, "INSERT INTO myCountry SELECT * FROM Country");


$query = "SELECT Name, Code FROM myCountry ORDER BY Name";
if ($stmt = mysqli_prepare($link, $query)) {

    /* drop table */
    mysqli_query($link, "DROP TABLE myCountry");

    /* execute query */
    mysqli_stmt_execute($stmt);

    echo "Error:\n";
    print_r(mysql_stmt_error_list($stmt));

    /* close statement */
    mysqli_stmt_close($stmt);
}
```

```
/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Array
(
    [0] => Array
        (
            [errno] => 1146
            [sqlstate] => 42S02
            [error] => Table 'world.myCountry' doesn't exist
        )

)
```

See Also

mysqli_stmt_error
mysqli_stmt_errno
mysqli_stmt_sqlstate

3.10.11 mysqli_stmt::$error, mysqli_stmt_error

- mysqli_stmt::$error

  mysqli_stmt_error

  Returns a string description for last statement error

Description

Object oriented style

```
  string
    mysqli_stmt->error ;
```

Procedural style

```
  string mysqli_stmt_error(
    mysqli_stmt stmt);
```

Returns a string containing the error message for the most recently invoked statement function that can succeed or fail.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

A string that describes the error. An empty string if no error occurred.

Examples

Example 3.87 Object oriented style

```
<?php
/* Open a connection */
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$mysqli->query("CREATE TABLE myCountry LIKE Country");
$mysqli->query("INSERT INTO myCountry SELECT * FROM Country");


$query = "SELECT Name, Code FROM myCountry ORDER BY Name";
if ($stmt = $mysqli->prepare($query)) {

    /* drop table */
    $mysqli->query("DROP TABLE myCountry");

    /* execute query */
    $stmt->execute();

    printf("Error: %s.\n", $stmt->error);

    /* close statement */
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.88 Procedural style

```
<?php
/* Open a connection */
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

mysqli_query($link, "CREATE TABLE myCountry LIKE Country");
mysqli_query($link, "INSERT INTO myCountry SELECT * FROM Country");


$query = "SELECT Name, Code FROM myCountry ORDER BY Name";
if ($stmt = mysqli_prepare($link, $query)) {

    /* drop table */
    mysqli_query($link, "DROP TABLE myCountry");
```

```
    /* execute query */
    mysqli_stmt_execute($stmt);

    printf("Error: %s.\n", mysqli_stmt_error($stmt));

    /* close statement */
    mysqli_stmt_close($stmt);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Error: Table 'world.myCountry' doesn't exist.
```

See Also

mysqli_stmt_errno
mysqli_stmt_sqlstate

3.10.12 mysqli_stmt::execute, mysqli_stmt_execute

- mysqli_stmt::execute

  mysqli_stmt_execute

  Executes a prepared Query

Description

Object oriented style

```
  bool mysqli_stmt::execute();
```

Procedural style

```
  bool mysqli_stmt_execute(
    mysqli_stmt stmt);
```

Executes a query that has been previously prepared using the mysqli_prepare function. When executed any parameter markers which exist will automatically be replaced with the appropriate data.

If the statement is UPDATE, DELETE, or INSERT, the total number of affected rows can be determined by using the mysqli_stmt_affected_rows function. Likewise, if the query yields a result set the mysqli_stmt_fetch function is used.

> **Note**
>
> When using mysqli_stmt_execute, the mysqli_stmt_fetch function must be used to fetch the data prior to performing any additional queries.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.89 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$mysqli->query("CREATE TABLE myCity LIKE City");

/* Prepare an insert statement */
$query = "INSERT INTO myCity (Name, CountryCode, District) VALUES (?,?,?)";
$stmt = $mysqli->prepare($query);

$stmt->bind_param("sss", $val1, $val2, $val3);

$val1 = 'Stuttgart';
$val2 = 'DEU';
$val3 = 'Baden-Wuerttemberg';

/* Execute the statement */
$stmt->execute();

$val1 = 'Bordeaux';
$val2 = 'FRA';
$val3 = 'Aquitaine';

/* Execute the statement */
$stmt->execute();

/* close statement */
$stmt->close();

/* retrieve all rows from myCity */
$query = "SELECT Name, CountryCode, District FROM myCity";
if ($result = $mysqli->query($query)) {
    while ($row = $result->fetch_row()) {
        printf("%s (%s,%s)\n", $row[0], $row[1], $row[2]);
    }
    /* free result set */
    $result->close();
}

/* remove table */
$mysqli->query("DROP TABLE myCity");

/* close connection */
$mysqli->close();
?>
```

Example 3.90 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

mysqli_query($link, "CREATE TABLE myCity LIKE City");

/* Prepare an insert statement */
$query = "INSERT INTO myCity (Name, CountryCode, District) VALUES (?,?,?)";
$stmt = mysqli_prepare($link, $query);

mysqli_stmt_bind_param($stmt, "sss", $val1, $val2, $val3);

$val1 = 'Stuttgart';
$val2 = 'DEU';
$val3 = 'Baden-Wuerttemberg';

/* Execute the statement */
mysqli_stmt_execute($stmt);

$val1 = 'Bordeaux';
$val2 = 'FRA';
$val3 = 'Aquitaine';

/* Execute the statement */
mysqli_stmt_execute($stmt);

/* close statement */
mysqli_stmt_close($stmt);

/* retrieve all rows from myCity */
$query = "SELECT Name, CountryCode, District FROM myCity";
if ($result = mysqli_query($link, $query)) {
    while ($row = mysqli_fetch_row($result)) {
        printf("%s (%s,%s)\n", $row[0], $row[1], $row[2]);
    }
    /* free result set */
    mysqli_free_result($result);
}

/* remove table */
mysqli_query($link, "DROP TABLE myCity");

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Stuttgart (DEU,Baden-Wuerttemberg)
Bordeaux (FRA,Aquitaine)
```

See Also

mysqli_prepare
mysqli_stmt_bind_param
mysqli_stmt_get_result

3.10.13 mysqli_stmt::fetch, mysqli_stmt_fetch

- mysqli_stmt::fetch

  mysqli_stmt_fetch

  Fetch results from a prepared statement into the bound variables

Description

Object oriented style

```
bool mysqli_stmt::fetch();
```

Procedural style

```
bool mysqli_stmt_fetch(
  mysqli_stmt stmt);
```

Fetch the result from a prepared statement into the variables bound by mysqli_stmt_bind_result.

> **Note**
>
> Note that all columns must be bound by the application before calling mysqli_stmt_fetch.

> **Note**
>
> Data are transferred unbuffered without calling mysqli_stmt_store_result which can decrease performance (but reduces memory cost).

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

Table 3.15 Return Values

Value	Description
TRUE	Success. Data has been fetched
FALSE	Error occurred
NULL	No more rows/data exists or data truncation occurred

Examples

Example 3.91 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER by ID DESC LIMIT 150,5";

if ($stmt = $mysqli->prepare($query)) {

    /* execute statement */
    $stmt->execute();

    /* bind result variables */
    $stmt->bind_result($name, $code);

    /* fetch values */
    while ($stmt->fetch()) {
        printf ("%s (%s)\n", $name, $code);
    }

    /* close statement */
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.92 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER by ID DESC LIMIT 150,5";

if ($stmt = mysqli_prepare($link, $query)) {

    /* execute statement */
    mysqli_stmt_execute($stmt);

    /* bind result variables */
    mysqli_stmt_bind_result($stmt, $name, $code);

    /* fetch values */
    while (mysqli_stmt_fetch($stmt)) {
        printf ("%s (%s)\n", $name, $code);
    }

    /* close statement */
```

```
    mysqli_stmt_close($stmt);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Rockford (USA)
Tallahassee (USA)
Salinas (USA)
Santa Clarita (USA)
Springfield (USA)
```

See Also

mysqli_prepare
mysqli_stmt_errno
mysqli_stmt_error
mysqli_stmt_bind_result

3.10.14 mysqli_stmt::$field_count, mysqli_stmt_field_count

- mysqli_stmt::$field_count

  mysqli_stmt_field_count

  Returns the number of field in the given statement

Description

Object oriented style

```
  int
    mysqli_stmt->field_count ;
```

Procedural style

```
  int mysqli_stmt_field_count(
    mysqli_stmt stmt);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

3.10.15 mysqli_stmt::free_result, mysqli_stmt_free_result

- mysqli_stmt::free_result

  mysqli_stmt_free_result

Frees stored result memory for the given statement handle

Description

Object oriented style

```
void mysqli_stmt::free_result();
```

Procedural style

```
void mysqli_stmt_free_result(
  mysqli_stmt stmt);
```

Frees the result memory associated with the statement, which was allocated by mysqli_stmt_store_result.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

No value is returned.

See Also

mysqli_stmt_store_result

3.10.16 mysqli_stmt::get_result, mysqli_stmt_get_result

- mysqli_stmt::get_result

  mysqli_stmt_get_result

  Gets a result set from a prepared statement

Description

Object oriented style

```
mysqli_result mysqli_stmt::get_result();
```

Procedural style

```
mysqli_result mysqli_stmt_get_result(
  mysqli_stmt stmt);
```

Call to return a result set from a prepared statement query.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

Returns a resultset for successful SELECT queries, or FALSE for other DML queries or on failure. The mysqli_errno function can be used to distinguish between the two types of failure.

MySQL Native Driver Only

Available only with mysqlnd.

Examples

Example 3.93 Object oriented style

```
<?php

$mysqli = new mysqli("127.0.0.1", "user", "password", "world");

if($mysqli->connect_error)
{
    die("$mysqli->connect_errno: $mysqli->connect_error");
}

$query = "SELECT Name, Population, Continent FROM Country WHERE Continent=? ORDER BY Name LIMIT 1";

$stmt = $mysqli->stmt_init();
if(!$stmt->prepare($query))
{
    print "Failed to prepare statement\n";
}
else
{
    $stmt->bind_param("s", $continent);

    $continent_array = array('Europe','Africa','Asia','North America');

    foreach($continent_array as $continent)
    {
        $stmt->execute();
        $result = $stmt->get_result();
        while ($row = $result->fetch_array(MYSQLI_NUM))
        {
            foreach ($row as $r)
            {
                print "$r ";
            }
            print "\n";
        }
    }
}

$stmt->close();
$mysqli->close();
?>
```

Example 3.94 Procedural style

```
<?php

$link = mysqli_connect("127.0.0.1", "user", "password", "world");

if (!$link)
{
```

```
    $error = mysqli_connect_error();
    $errno = mysqli_connect_errno();
    print "$errno: $error\n";
    exit();
}

$query = "SELECT Name, Population, Continent FROM Country WHERE Continent=? ORDER BY Name LIMIT 1";

$stmt = mysqli_stmt_init($link);
if(!mysqli_stmt_prepare($stmt, $query))
{
    print "Failed to prepare statement\n";
}
else
{
    mysqli_stmt_bind_param($stmt, "s", $continent);

    $continent_array = array('Europe','Africa','Asia','North America');

    foreach($continent_array as $continent)
    {
        mysqli_stmt_execute($stmt);
        $result = mysqli_stmt_get_result($stmt);
        while ($row = mysqli_fetch_array($result, MYSQLI_NUM))
        {
            foreach ($row as $r)
            {
                print "$r ";
            }
            print "\n";
        }
    }
}
mysqli_stmt_close($stmt);
mysqli_close($link);
?>
```

The above examples will output:

```
Albania 3401200 Europe
Algeria 31471000 Africa
Afghanistan 22720000 Asia
Anguilla 8000 North America
```

See Also

mysqli_prepare
mysqli_stmt_result_metadata
mysqli_stmt_fetch
mysqli_fetch_array
mysqli_stmt_store_result
mysqli_errno

3.10.17 mysqli_stmt::get_warnings, mysqli_stmt_get_warnings

- mysqli_stmt::get_warnings

mysqli_stmt_get_warnings

Get result of SHOW WARNINGS

Description

Object oriented style

```
object mysqli_stmt::get_warnings(
  mysqli_stmt stmt);
```

Procedural style

```
object mysqli_stmt_get_warnings(
  mysqli_stmt stmt);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

3.10.18 mysqli_stmt::$insert_id, mysqli_stmt_insert_id

- mysqli_stmt::$insert_id

  mysqli_stmt_insert_id

  Get the ID generated from the previous INSERT operation

Description

Object oriented style

```
int
  mysqli_stmt->insert_id ;
```

Procedural style

```
mixed mysqli_stmt_insert_id(
  mysqli_stmt stmt);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

3.10.19 mysqli_stmt::more_results, mysqli_stmt_more_results

- mysqli_stmt::more_results

  mysqli_stmt_more_results

  Check if there are more query results from a multiple query

Description

Object oriented style (method):

```
public bool mysqli_stmt::more_results();
```

Procedural style:

```
bool mysqli_stmt_more_results(
  mysql_stmt stmt);
```

Checks if there are more query results from a multiple query.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

Returns TRUE if more results exist, otherwise FALSE.

MySQL Native Driver Only

Available only with mysqlnd.

See Also

mysqli_stmt::next_result
mysqli::multi_query

3.10.20 mysqli_stmt::next_result, mysqli_stmt_next_result

- mysqli_stmt::next_result

  mysqli_stmt_next_result

  Reads the next result from a multiple query

Description

Object oriented style (method):

```
public bool mysqli_stmt::next_result();
```

Procedural style:

```
bool mysqli_stmt_next_result(
  mysql_stmt stmt);
```

Reads the next result from a multiple query.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

Returns TRUE on success or FALSE on failure.

Errors/Exceptions

Emits an E_STRICT level error if a result set does not exist, and suggests using mysqli_stmt::more_results in these cases, before calling mysqli_stmt::next_result.

MySQL Native Driver Only

Available only with mysqlnd.

See Also

mysqli_stmt::more_results
mysqli::multi_query

3.10.21 mysqli_stmt::$num_rows, mysqli_stmt_num_rows

- mysqli_stmt::$num_rows

  mysqli_stmt_num_rows

  Return the number of rows in statements result set

Description

Object oriented style

```
int
  mysqli_stmt->num_rows ;
```

Procedural style

```
int mysqli_stmt_num_rows(
  mysqli_stmt stmt);
```

Returns the number of rows in the result set. The use of mysqli_stmt_num_rows depends on whether or not you used mysqli_stmt_store_result to buffer the entire result set in the statement handle.

If you use mysqli_stmt_store_result, mysqli_stmt_num_rows may be called immediately.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

An integer representing the number of rows in result set.

Examples

Example 3.95 Object oriented style

```
<?php
/* Open a connection */
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");
```

```
/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER BY Name LIMIT 20";
if ($stmt = $mysqli->prepare($query)) {

    /* execute query */
    $stmt->execute();

    /* store result */
    $stmt->store_result();

    printf("Number of rows: %d.\n", $stmt->num_rows);

    /* close statement */
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.96 Procedural style

```
<?php
/* Open a connection */
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER BY Name LIMIT 20";
if ($stmt = mysqli_prepare($link, $query)) {

    /* execute query */
    mysqli_stmt_execute($stmt);

    /* store result */
    mysqli_stmt_store_result($stmt);

    printf("Number of rows: %d.\n", mysqli_stmt_num_rows($stmt));

    /* close statement */
    mysqli_stmt_close($stmt);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
```

```
Number of rows: 20.
```

See Also

mysqli_stmt_affected_rows
mysqli_prepare
mysqli_stmt_store_result

3.10.22 mysqli_stmt::$param_count, mysqli_stmt_param_count

- mysqli_stmt::$param_count

  mysqli_stmt_param_count

  Returns the number of parameter for the given statement

Description

Object oriented style

```
  int
    mysqli_stmt->param_count ;
```

Procedural style

```
  int mysqli_stmt_param_count(
    mysqli_stmt stmt);
```

Returns the number of parameter markers present in the prepared statement.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

Returns an integer representing the number of parameters.

Examples

Example 3.97 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

if ($stmt = $mysqli->prepare("SELECT Name FROM Country WHERE Name=? OR Code=?")) {
```

```
    $marker = $stmt->param_count;
    printf("Statement has %d markers.\n", $marker);

    /* close statement */
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.98 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

if ($stmt = mysqli_prepare($link, "SELECT Name FROM Country WHERE Name=? OR Code=?")) {

    $marker = mysqli_stmt_param_count($stmt);
    printf("Statement has %d markers.\n", $marker);

    /* close statement */
    mysqli_stmt_close($stmt);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Statement has 2 markers.
```

See Also

mysqli_prepare

3.10.23 mysqli_stmt::prepare, mysqli_stmt_prepare

- mysqli_stmt::prepare

  mysqli_stmt_prepare

  Prepare an SQL statement for execution

Description

Object oriented style

```
mixed mysqli_stmt::prepare(
  string query);
```

Procedural style

```
bool mysqli_stmt_prepare(
  mysqli_stmt stmt,
  string query);
```

Prepares the SQL query pointed to by the null-terminated string query.

The parameter markers must be bound to application variables using `mysqli_stmt_bind_param` and/or `mysqli_stmt_bind_result` before executing the statement or fetching rows.

> **Note**
>
> In the case where you pass a statement to `mysqli_stmt_prepare` that is longer than `max_allowed_packet` of the server, the returned error codes are different depending on whether you are using MySQL Native Driver (`mysqlnd`) or MySQL Client Library (`libmysqlclient`). The behavior is as follows:
>
> - `mysqlnd` on Linux returns an error code of 1153. The error message means “got a packet bigger than `max_allowed_packet` bytes”.
> - `mysqlnd` on Windows returns an error code 2006. This error message means “server has gone away”.
> - `libmysqlclient` on all platforms returns an error code 2006. This error message means “server has gone away”.

Parameters

`stmt`
: Procedural style only: A statement identifier returned by `mysqli_stmt_init`.

`query`
: The query, as a string. It must consist of a single SQL statement.

  You can include one or more parameter markers in the SQL statement by embedding question mark (`?`) characters at the appropriate positions.

  > **Note**
  >
  > You should not add a terminating semicolon or `\g` to the statement.

  > **Note**
  >
  > The markers are legal only in certain places in SQL statements. For example, they are allowed in the VALUES() list of an INSERT statement (to specify column values for a row), or in a comparison with a column in a WHERE clause to specify a comparison value.
  >
  > However, they are not allowed for identifiers (such as table or column names), in the select

list that names the columns to be returned by a SELECT statement), or to specify both operands of a binary operator such as the = equal sign. The latter restriction is necessary because it would be impossible to determine the parameter type. In general, parameters are legal only in Data Manipulation Language (DML) statements, and not in Data Definition Language (DDL) statements.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.99 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$city = "Amersfoort";

/* create a prepared statement */
$stmt =  $mysqli->stmt_init();
if ($stmt->prepare("SELECT District FROM City WHERE Name=?")) {

    /* bind parameters for markers */
    $stmt->bind_param("s", $city);

    /* execute query */
    $stmt->execute();

    /* bind result variables */
    $stmt->bind_result($district);

    /* fetch value */
    $stmt->fetch();

    printf("%s is in district %s\n", $city, $district);

    /* close statement */
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.100 Procedural style

```
<?php
```

```
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$city = "Amersfoort";

/* create a prepared statement */
$stmt = mysqli_stmt_init($link);
if (mysqli_stmt_prepare($stmt, 'SELECT District FROM City WHERE Name=?')) {

    /* bind parameters for markers */
    mysqli_stmt_bind_param($stmt, "s", $city);

    /* execute query */
    mysqli_stmt_execute($stmt);

    /* bind result variables */
    mysqli_stmt_bind_result($stmt, $district);

    /* fetch value */
    mysqli_stmt_fetch($stmt);

    printf("%s is in district %s\n", $city, $district);

    /* close statement */
    mysqli_stmt_close($stmt);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Amersfoort is in district Utrecht
```

See Also

mysqli_stmt_init
mysqli_stmt_execute
mysqli_stmt_fetch
mysqli_stmt_bind_param
mysqli_stmt_bind_result
mysqli_stmt_get_result
mysqli_stmt_close

3.10.24 mysqli_stmt::reset, mysqli_stmt_reset

- mysqli_stmt::reset

  mysqli_stmt_reset

Resets a prepared statement

Description

Object oriented style

```
bool mysqli_stmt::reset();
```

Procedural style

```
bool mysqli_stmt_reset(
  mysqli_stmt stmt);
```

Resets a prepared statement on client and server to state after prepare.

It resets the statement on the server, data sent using mysqli_stmt_send_long_data, unbuffered result sets and current errors. It does not clear bindings or stored result sets. Stored result sets will be cleared when executing the prepared statement (or closing it).

To prepare a statement with another query use function mysqli_stmt_prepare.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

Returns TRUE on success or FALSE on failure.

See Also

mysqli_prepare

3.10.25 mysqli_stmt::result_metadata, mysqli_stmt_result_metadata

- mysqli_stmt::result_metadata

  mysqli_stmt_result_metadata

  Returns result set metadata from a prepared statement

Description

Object oriented style

```
mysqli_result mysqli_stmt::result_metadata();
```

Procedural style

```
mysqli_result mysqli_stmt_result_metadata(
  mysqli_stmt stmt);
```

If a statement passed to mysqli_prepare is one that produces a result set, mysqli_stmt_result_metadata returns the result object that can be used to process the meta information such as total number of fields and individual field information.

> **Note**
>
> This result set pointer can be passed as an argument to any of the field-based functions that process result set metadata, such as:
>
> - mysqli_num_fields
> - mysqli_fetch_field
> - mysqli_fetch_field_direct
> - mysqli_fetch_fields
> - mysqli_field_count
> - mysqli_field_seek
> - mysqli_field_tell
> - mysqli_free_result

The result set structure should be freed when you are done with it, which you can do by passing it to mysqli_free_result

> **Note**
>
> The result set returned by mysqli_stmt_result_metadata contains only metadata. It does not contain any row results. The rows are obtained by using the statement handle with mysqli_stmt_fetch.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

Returns a result object or FALSE if an error occurred.

Examples

Example 3.101 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "test");

$mysqli->query("DROP TABLE IF EXISTS friends");
$mysqli->query("CREATE TABLE friends (id int, name varchar(20))");

$mysqli->query("INSERT INTO friends VALUES (1,'Hartmut'), (2, 'Ulf')");

$stmt = $mysqli->prepare("SELECT id, name FROM friends");
$stmt->execute();

/* get resultset for metadata */
$result = $stmt->result_metadata();

/* retrieve field information from metadata result set */
$field = $result->fetch_field();
```

```
printf("Fieldname: %s\n", $field->name);

/* close resultset */
$result->close();

/* close connection */
$mysqli->close();
?>
```

Example 3.102 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "test");

mysqli_query($link, "DROP TABLE IF EXISTS friends");
mysqli_query($link, "CREATE TABLE friends (id int, name varchar(20))");

mysqli_query($link, "INSERT INTO friends VALUES (1,'Hartmut'), (2, 'Ulf')");

$stmt = mysqli_prepare($link, "SELECT id, name FROM friends");
mysqli_stmt_execute($stmt);

/* get resultset for metadata */
$result = mysqli_stmt_result_metadata($stmt);

/* retrieve field information from metadata result set */
$field = mysqli_fetch_field($result);

printf("Fieldname: %s\n", $field->name);

/* close resultset */
mysqli_free_result($result);

/* close connection */
mysqli_close($link);
?>
```

See Also

mysqli_prepare
mysqli_free_result

3.10.26 mysqli_stmt::send_long_data, mysqli_stmt_send_long_data

- mysqli_stmt::send_long_data

  mysqli_stmt_send_long_data

  Send data in blocks

Description

Object oriented style

```
  bool mysqli_stmt::send_long_data(
```

```
    int param_nr,
    string data);
```

Procedural style

```
  bool mysqli_stmt_send_long_data(
    mysqli_stmt stmt,
    int param_nr,
    string data);
```

Allows to send parameter data to the server in pieces (or chunks), e.g. if the size of a blob exceeds the size of max_allowed_packet. This function can be called multiple times to send the parts of a character or binary data value for a column, which must be one of the TEXT or BLOB datatypes.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.
param_nr	Indicates which parameter to associate the data with. Parameters are numbered beginning with 0.
data	A string containing data to be sent.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.103 Object oriented style

```
<?php
$stmt = $mysqli->prepare("INSERT INTO messages (message) VALUES (?)");
$null = NULL;
$stmt->bind_param("b", $null);
$fp = fopen("messages.txt", "r");
while (!feof($fp)) {
    $stmt->send_long_data(0, fread($fp, 8192));
}
fclose($fp);
$stmt->execute();
?>
```

See Also

mysqli_prepare
mysqli_stmt_bind_param

3.10.27 mysqli_stmt::$sqlstate, mysqli_stmt_sqlstate

- mysqli_stmt::$sqlstate

  mysqli_stmt_sqlstate

Returns SQLSTATE error from previous statement operation

Description

Object oriented style

```
  string
    mysqli_stmt->sqlstate ;
```

Procedural style

```
  string mysqli_stmt_sqlstate(
    mysqli_stmt stmt);
```

Returns a string containing the SQLSTATE error code for the most recently invoked prepared statement function that can succeed or fail. The error code consists of five characters. `'00000'` means no error. The values are specified by ANSI SQL and ODBC. For a list of possible values, see http://dev.mysql.com/doc/mysql/en/error-handling.html.

Parameters

`stmt`	Procedural style only: A statement identifier returned by `mysqli_stmt_init`.

Return Values

Returns a string containing the SQLSTATE error code for the last error. The error code consists of five characters. `'00000'` means no error.

Notes

> **Note**
>
> Note that not all MySQL errors are yet mapped to SQLSTATE's. The value `HY000` (general error) is used for unmapped errors.

Examples

Example 3.104 Object oriented style

```
<?php
/* Open a connection */
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$mysqli->query("CREATE TABLE myCountry LIKE Country");
$mysqli->query("INSERT INTO myCountry SELECT * FROM Country");


$query = "SELECT Name, Code FROM myCountry ORDER BY Name";
if ($stmt = $mysqli->prepare($query)) {

    /* drop table */
    $mysqli->query("DROP TABLE myCountry");
```

```
    /* execute query */
    $stmt->execute();

    printf("Error: %s.\n", $stmt->sqlstate);

    /* close statement */
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.105 Procedural style

```
<?php
/* Open a connection */
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

mysqli_query($link, "CREATE TABLE myCountry LIKE Country");
mysqli_query($link, "INSERT INTO myCountry SELECT * FROM Country");


$query = "SELECT Name, Code FROM myCountry ORDER BY Name";
if ($stmt = mysqli_prepare($link, $query)) {

    /* drop table */
    mysqli_query($link, "DROP TABLE myCountry");

    /* execute query */
    mysqli_stmt_execute($stmt);

    printf("Error: %s.\n", mysqli_stmt_sqlstate($stmt));

    /* close statement */
    mysqli_stmt_close($stmt);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Error: 42S02.
```

See Also

mysqli_stmt_errno

mysqli_stmt_error

3.10.28 mysqli_stmt::store_result, mysqli_stmt_store_result

- mysqli_stmt::store_result

  mysqli_stmt_store_result

  Transfers a result set from a prepared statement

Description

Object oriented style

```
bool mysqli_stmt::store_result();
```

Procedural style

```
bool mysqli_stmt_store_result(
  mysqli_stmt stmt);
```

You must call mysqli_stmt_store_result for every query that successfully produces a result set (SELECT, SHOW, DESCRIBE, EXPLAIN), if and only if you want to buffer the complete result set by the client, so that the subsequent mysqli_stmt_fetch call returns buffered data.

> **Note**
>
> It is unnecessary to call mysqli_stmt_store_result for other queries, but if you do, it will not harm or cause any notable performance loss in all cases. You can detect whether the query produced a result set by checking if mysqli_stmt_result_metadata returns NULL.

Parameters

stmt	Procedural style only: A statement identifier returned by mysqli_stmt_init.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.106 Object oriented style

```
<?php
/* Open a connection */
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER BY Name LIMIT 20";
```

```
if ($stmt = $mysqli->prepare($query)) {

    /* execute query */
    $stmt->execute();

    /* store result */
    $stmt->store_result();

    printf("Number of rows: %d.\n", $stmt->num_rows);

    /* free result */
    $stmt->free_result();

    /* close statement */
    $stmt->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.107 Procedural style

```
<?php
/* Open a connection */
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER BY Name LIMIT 20";
if ($stmt = mysqli_prepare($link, $query)) {

    /* execute query */
    mysqli_stmt_execute($stmt);

    /* store result */
    mysqli_stmt_store_result($stmt);

    printf("Number of rows: %d.\n", mysqli_stmt_num_rows($stmt));

    /* free result */
    mysqli_stmt_free_result($stmt);

    /* close statement */
    mysqli_stmt_close($stmt);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Number of rows: 20.
```

See Also

mysqli_prepare
mysqli_stmt_result_metadata
mysqli_stmt_fetch

3.11 The mysqli_result class

Represents the result set obtained from a query against the database.

Changelog

Table 3.16 Changelog

Version	Description
5.4.0	Iterator support was added, as mysqli_result now implements Traversable.

```
mysqli_result {
mysqli_result

        Traversable

      Properties

  int
    mysqli_result->current_field ;

  int
    mysqli_result->field_count ;

  array
    mysqli_result->lengths ;

  int
    mysqli_result->num_rows ;

Methods

  bool mysqli_result::data_seek(
    int offset);

  mixed mysqli_result::fetch_all(
    int resulttype
        =  =MYSQLI_NUM);

  mixed mysqli_result::fetch_array(
    int resulttype
        =  =MYSQLI_BOTH);

  array mysqli_result::fetch_assoc();

  object mysqli_result::fetch_field_direct(
    int fieldnr);

  object mysqli_result::fetch_field();
```

```
  array mysqli_result::fetch_fields();

  object mysqli_result::fetch_object(
    string class_name
        =  ="stdClass",
    array params);

  mixed mysqli_result::fetch_row();

  bool mysqli_result::field_seek(
    int fieldnr);

  void mysqli_result::free();

}
```

3.11.1 mysqli_result::$current_field, mysqli_field_tell

- mysqli_result::$current_field

  mysqli_field_tell

  Get current field offset of a result pointer

Description

Object oriented style

```
  int
    mysqli_result->current_field ;
```

Procedural style

```
  int mysqli_field_tell(
    mysqli_result result);
```

Returns the position of the field cursor used for the last mysqli_fetch_field call. This value can be used as an argument to mysqli_field_seek.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.

Return Values

Returns current offset of field cursor.

Examples

Example 3.108 Object oriented style

```


<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
```

```
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, SurfaceArea from Country ORDER BY Code LIMIT 5";

if ($result = $mysqli->query($query)) {

    /* Get field information for all columns */
    while ($finfo = $result->fetch_field()) {

        /* get fieldpointer offset */
        $currentfield = $result->current_field;

        printf("Column %d:\n", $currentfield);
        printf("Name:     %s\n", $finfo->name);
        printf("Table:    %s\n", $finfo->table);
        printf("max. Len: %d\n", $finfo->max_length);
        printf("Flags:    %d\n", $finfo->flags);
        printf("Type:     %d\n\n", $finfo->type);
    }
    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.109 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, SurfaceArea from Country ORDER BY Code LIMIT 5";

if ($result = mysqli_query($link, $query)) {

    /* Get field information for all fields */
    while ($finfo = mysqli_fetch_field($result)) {

        /* get fieldpointer offset */
        $currentfield = mysqli_field_tell($result);

        printf("Column %d:\n", $currentfield);
        printf("Name:     %s\n", $finfo->name);
        printf("Table:    %s\n", $finfo->table);
        printf("max. Len: %d\n", $finfo->max_length);
        printf("Flags:    %d\n", $finfo->flags);
        printf("Type:     %d\n\n", $finfo->type);
    }
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

```
```

The above examples will output:

```
Column 1:
Name:     Name
Table:    Country
max. Len: 11
Flags:    1
Type:     254

Column 2:
Name:     SurfaceArea
Table:    Country
max. Len: 10
Flags:    32769
Type:     4
```

See Also

mysqli_fetch_field
mysqli_field_seek

3.11.2 mysqli_result::data_seek, mysqli_data_seek

- mysqli_result::data_seek

  mysqli_data_seek

  Adjusts the result pointer to an arbitrary row in the result

Description

Object oriented style

```
bool mysqli_result::data_seek(
  int offset);
```

Procedural style

```
bool mysqli_data_seek(
  mysqli_result result,
  int offset);
```

The mysqli_data_seek function seeks to an arbitrary result pointer specified by the *offset* in the result set.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.
offset	The field offset. Must be between zero and the total number of rows minus one (0..mysqli_num_rows - 1).

Return Values

Returns TRUE on success or FALSE on failure.

Notes

> **Note**
>
> This function can only be used with buffered results attained from the use of the mysqli_store_result or mysqli_query functions.

Examples

Example 3.110 Object oriented style

```
<?php
/* Open a connection */
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER BY Name";
if ($result = $mysqli->query($query)) {

    /* seek to row no. 400 */
    $result->data_seek(399);

    /* fetch row */
    $row = $result->fetch_row();

    printf ("City: %s  Countrycode: %s\n", $row[0], $row[1]);

    /* free result set*/
    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.111 Procedural style

```
<?php
/* Open a connection */
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (!$link) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER BY Name";
```

```
if ($result = mysqli_query($link, $query)) {

    /* seek to row no. 400 */
    mysqli_data_seek($result, 399);

    /* fetch row */
    $row = mysqli_fetch_row($result);

    printf ("City: %s  Countrycode: %s\n", $row[0], $row[1]);

    /* free result set*/
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
City: Benin City  Countrycode: NGA
```

See Also

mysqli_store_result
mysqli_fetch_row
mysqli_fetch_array
mysqli_fetch_assoc
mysqli_fetch_object
mysqli_query
mysqli_num_rows

3.11.3 mysqli_result::fetch_all, mysqli_fetch_all

- mysqli_result::fetch_all

  mysqli_fetch_all

  Fetches all result rows as an associative array, a numeric array, or both

Description

Object oriented style

```
  mixed mysqli_result::fetch_all(
    int resulttype
        =  =MYSQLI_NUM);
```

Procedural style

```
  mixed mysqli_fetch_all(
    mysqli_result result,
    int resulttype
        =  =MYSQLI_NUM);
```

mysqli_fetch_all fetches all result rows and returns the result set as an associative array, a numeric array, or both.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.
resulttype	This optional parameter is a constant indicating what type of array should be produced from the current row data. The possible values for this parameter are the constants MYSQLI_ASSOC, MYSQLI_NUM, or MYSQLI_BOTH.

Return Values

Returns an array of associative or numeric arrays holding result rows.

MySQL Native Driver Only

Available only with mysqlnd.

As mysqli_fetch_all returns all the rows as an array in a single step, it may consume more memory than some similar functions such as mysqli_fetch_array, which only returns one row at a time from the result set. Further, if you need to iterate over the result set, you will need a looping construct that will further impact performance. For these reasons mysqli_fetch_all should only be used in those situations where the fetched result set will be sent to another layer for processing.

See Also

mysqli_fetch_array
mysqli_query

3.11.4 mysqli_result::fetch_array, mysqli_fetch_array

- mysqli_result::fetch_array

  mysqli_fetch_array

  Fetch a result row as an associative, a numeric array, or both

Description

Object oriented style

```
mixed mysqli_result::fetch_array(
  int resulttype
      =  =MYSQLI_BOTH);
```

Procedural style

```
mixed mysqli_fetch_array(
  mysqli_result result,
  int resulttype
      =  =MYSQLI_BOTH);
```

Returns an array that corresponds to the fetched row or NULL if there are no more rows for the resultset represented by the *result* parameter.

mysqli_fetch_array is an extended version of the mysqli_fetch_row function. In addition to storing the data in the numeric indices of the result array, the mysqli_fetch_array function can also store the data in associative indices, using the field names of the result set as keys.

> **Note**
>
> Field names returned by this function are *case-sensitive*.

> **Note**
>
> This function sets NULL fields to the PHP NULL value.

If two or more columns of the result have the same field names, the last column will take precedence and overwrite the earlier data. In order to access multiple columns with the same name, the numerically indexed version of the row must be used.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.
resulttype	This optional parameter is a constant indicating what type of array should be produced from the current row data. The possible values for this parameter are the constants MYSQLI_ASSOC, MYSQLI_NUM, or MYSQLI_BOTH. By using the MYSQLI_ASSOC constant this function will behave identically to the mysqli_fetch_assoc, while MYSQLI_NUM will behave identically to the mysqli_fetch_row function. The final option MYSQLI_BOTH will create a single array with the attributes of both.

Return Values

Returns an array of strings that corresponds to the fetched row or NULL if there are no more rows in resultset.

Examples

Example 3.112 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if ($mysqli->connect_errno) {
    printf("Connect failed: %s\n", $mysqli->connect_error);
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER by ID LIMIT 3";
$result = $mysqli->query($query);

/* numeric array */
$row = $result->fetch_array(MYSQLI_NUM);
printf ("%s (%s)\n", $row[0], $row[1]);

/* associative array */
$row = $result->fetch_array(MYSQLI_ASSOC);
```

```
printf ("%s (%s)\n", $row["Name"], $row["CountryCode"]);

/* associative and numeric array */
$row = $result->fetch_array(MYSQLI_BOTH);
printf ("%s (%s)\n", $row[0], $row["CountryCode"]);

/* free result set */
$result->free();

/* close connection */
$mysqli->close();
?>
```

Example 3.113 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER by ID LIMIT 3";
$result = mysqli_query($link, $query);

/* numeric array */
$row = mysqli_fetch_array($result, MYSQLI_NUM);
printf ("%s (%s)\n", $row[0], $row[1]);

/* associative array */
$row = mysqli_fetch_array($result, MYSQLI_ASSOC);
printf ("%s (%s)\n", $row["Name"], $row["CountryCode"]);

/* associative and numeric array */
$row = mysqli_fetch_array($result, MYSQLI_BOTH);
printf ("%s (%s)\n", $row[0], $row["CountryCode"]);

/* free result set */
mysqli_free_result($result);

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Kabul (AFG)
Qandahar (AFG)
Herat (AFG)
```

See Also

mysqli_fetch_assoc
mysqli_fetch_row

mysqli_fetch_object
mysqli_query
mysqli_data_seek

3.11.5 mysqli_result::fetch_assoc, mysqli_fetch_assoc

- mysqli_result::fetch_assoc

  mysqli_fetch_assoc

  Fetch a result row as an associative array

Description

Object oriented style

```
array mysqli_result::fetch_assoc();
```

Procedural style

```
array mysqli_fetch_assoc(
  mysqli_result result);
```

Returns an associative array that corresponds to the fetched row or NULL if there are no more rows.

> **Note**
>
> Field names returned by this function are *case-sensitive*.

> **Note**
>
> This function sets NULL fields to the PHP NULL value.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.

Return Values

Returns an associative array of strings representing the fetched row in the result set, where each key in the array represents the name of one of the result set's columns or NULL if there are no more rows in resultset.

If two or more columns of the result have the same field names, the last column will take precedence. To access the other column(s) of the same name, you either need to access the result with numeric indices by using mysqli_fetch_row or add alias names.

Examples

Example 3.114 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
```

```
if ($mysqli->connect_errno) {
    printf("Connect failed: %s\n", $mysqli->connect_error);
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER by ID DESC LIMIT 50,5";

if ($result = $mysqli->query($query)) {

    /* fetch associative array */
    while ($row = $result->fetch_assoc()) {
        printf ("%s (%s)\n", $row["Name"], $row["CountryCode"]);
    }

    /* free result set */
    $result->free();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.115 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER by ID DESC LIMIT 50,5";

if ($result = mysqli_query($link, $query)) {

    /* fetch associative array */
    while ($row = mysqli_fetch_assoc($result)) {
        printf ("%s (%s)\n", $row["Name"], $row["CountryCode"]);
    }

    /* free result set */
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Pueblo (USA)
Arvada (USA)
Cape Coral (USA)
Green Bay (USA)
Santa Clara (USA)
```

Example 3.116 A `mysqli_result` example comparing `iterator` usage

```
<?php
$c = mysqli_connect('127.0.0.1','user', 'pass');

// Using iterators (support was added with PHP 5.4)
foreach ( $c->query('SELECT user,host FROM mysql.user') as $row ) {
    printf("'%s'@'%s'\n", $row['user'], $row['host']);
}

echo "\n==================\n";

// Not using iterators
$result = $c->query('SELECT user,host FROM mysql.user');
while ($row = $result->fetch_assoc()) {
    printf("'%s'@'%s'\n", $row['user'], $row['host']);
}

?>
```

The above example will output something similar to:

```
'root'@'192.168.1.1'
'root'@'127.0.0.1'
'dude'@'localhost'
'lebowski'@'localhost'

==================

'root'@'192.168.1.1'
'root'@'127.0.0.1'
'dude'@'localhost'
'lebowski'@'localhost'
```

See Also

mysqli_fetch_array
mysqli_fetch_row
mysqli_fetch_object
mysqli_query
mysqli_data_seek

3.11.6 `mysqli_result::fetch_field_direct`, `mysqli_fetch_field_direct`

- `mysqli_result::fetch_field_direct`

  `mysqli_fetch_field_direct`

  Fetch meta-data for a single field

Description

Object oriented style

```
object mysqli_result::fetch_field_direct(
  int fieldnr);
```

Procedural style

```
object mysqli_fetch_field_direct(
  mysqli_result result,
  int fieldnr);
```

Returns an object which contains field definition information from the specified result set.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.
fieldnr	The field number. This value must be in the range from 0 to number of fields - 1.

Return Values

Returns an object which contains field definition information or FALSE if no field information for specified fieldnr is available.

Table 3.17 Object attributes

Attribute	Description
name	The name of the column
orgname	Original column name if an alias was specified
table	The name of the table this field belongs to (if not calculated)
orgtable	Original table name if an alias was specified
def	The default value for this field, represented as a string
max_length	The maximum width of the field for the result set.
length	The width of the field, as specified in the table definition.
charsetnr	The character set number for the field.
flags	An integer representing the bit-flags for the field.
type	The data type used for this field
decimals	The number of decimals used (for numeric fields)

Examples

Example 3.117 Object oriented style

```

<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
```

```
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, SurfaceArea from Country ORDER BY Name LIMIT 5";

if ($result = $mysqli->query($query)) {

    /* Get field information for column 'SurfaceArea' */
    $finfo = $result->fetch_field_direct(1);

    printf("Name:     %s\n", $finfo->name);
    printf("Table:    %s\n", $finfo->table);
    printf("max. Len: %d\n", $finfo->max_length);
    printf("Flags:    %d\n", $finfo->flags);
    printf("Type:     %d\n", $finfo->type);

    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.118 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, SurfaceArea from Country ORDER BY Name LIMIT 5";

if ($result = mysqli_query($link, $query)) {

    /* Get field information for column 'SurfaceArea' */
    $finfo = mysqli_fetch_field_direct($result, 1);

    printf("Name:     %s\n", $finfo->name);
    printf("Table:    %s\n", $finfo->table);
    printf("max. Len: %d\n", $finfo->max_length);
    printf("Flags:    %d\n", $finfo->flags);
    printf("Type:     %d\n", $finfo->type);

    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Name:     SurfaceArea
Table:    Country
```

```
max. Len: 10
Flags:    32769
Type:     4
```

See Also

mysqli_num_fields
mysqli_fetch_field
mysqli_fetch_fields

3.11.7 mysqli_result::fetch_field, mysqli_fetch_field

- mysqli_result::fetch_field

  mysqli_fetch_field

  Returns the next field in the result set

Description

Object oriented style

```
object mysqli_result::fetch_field();
```

Procedural style

```
object mysqli_fetch_field(
  mysqli_result result);
```

Returns the definition of one column of a result set as an object. Call this function repeatedly to retrieve information about all columns in the result set.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.

Return Values

Returns an object which contains field definition information or FALSE if no field information is available.

Table 3.18 Object properties

Property	Description
name	The name of the column
orgname	Original column name if an alias was specified
table	The name of the table this field belongs to (if not calculated)
orgtable	Original table name if an alias was specified
def	Reserved for default value, currently always ""
db	Database (since PHP 5.3.6)
catalog	The catalog name, always "def" (since PHP 5.3.6)

Property	Description
max_length	The maximum width of the field for the result set.
length	The width of the field, as specified in the table definition.
charsetnr	The character set number for the field.
flags	An integer representing the bit-flags for the field.
type	The data type used for this field
decimals	The number of decimals used (for integer fields)

Examples

Example 3.119 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, SurfaceArea from Country ORDER BY Code LIMIT 5";

if ($result = $mysqli->query($query)) {

    /* Get field information for all columns */
    while ($finfo = $result->fetch_field()) {

        printf("Name:     %s\n", $finfo->name);
        printf("Table:    %s\n", $finfo->table);
        printf("max. Len: %d\n", $finfo->max_length);
        printf("Flags:    %d\n", $finfo->flags);
        printf("Type:     %d\n\n", $finfo->type);
    }
    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.120 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, SurfaceArea from Country ORDER BY Code LIMIT 5";
```

```
if ($result = mysqli_query($link, $query)) {

    /* Get field information for all fields */
    while ($finfo = mysqli_fetch_field($result)) {

        printf("Name:     %s\n", $finfo->name);
        printf("Table:    %s\n", $finfo->table);
        printf("max. Len: %d\n", $finfo->max_length);
        printf("Flags:    %d\n", $finfo->flags);
        printf("Type:     %d\n\n", $finfo->type);
    }
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```


Name:     Name
Table:    Country
max. Len: 11
Flags:    1
Type:     254

Name:     SurfaceArea
Table:    Country
max. Len: 10
Flags:    32769
Type:     4
```

See Also

mysqli_num_fields
mysqli_fetch_field_direct
mysqli_fetch_fields
mysqli_field_seek

3.11.8 mysqli_result::fetch_fields, mysqli_fetch_fields

- mysqli_result::fetch_fields

  mysqli_fetch_fields

  Returns an array of objects representing the fields in a result set

Description

Object oriented style

```
  array mysqli_result::fetch_fields();
```

Procedural style

```
  array mysqli_fetch_fields(
    mysqli_result result);
```

This function serves an identical purpose to the mysqli_fetch_field function with the single difference that, instead of returning one object at a time for each field, the columns are returned as an array of objects.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.

Return Values

Returns an array of objects which contains field definition information or FALSE if no field information is available.

Table 3.19 Object properties

Property	Description
name	The name of the column
orgname	Original column name if an alias was specified
table	The name of the table this field belongs to (if not calculated)
orgtable	Original table name if an alias was specified
max_length	The maximum width of the field for the result set.
length	The width of the field, in bytes, as specified in the table definition. Note that this number (bytes) might differ from your table definition value (characters), depending on the character set you use. For example, the character set utf8 has 3 bytes per character, so varchar(10) will return a length of 30 for utf8 (10*3), but return 10 for latin1 (10*1).
charsetnr	The character set number (id) for the field.
flags	An integer representing the bit-flags for the field.
type	The data type used for this field
decimals	The number of decimals used (for integer fields)

Examples

Example 3.121 Object oriented style

```
<?php
$mysqli = new mysqli("127.0.0.1", "root", "foofoo", "sakila");

/* check connection */
if ($mysqli->connect_errno) {
    printf("Connect failed: %s\n", $mysqli->connect_error);
    exit();
}

foreach (array('latin1', 'utf8') as $charset) {
```

```
    // Set character set, to show its impact on some values (e.g., length in bytes)
    $mysqli->set_charset($charset);

    $query = "SELECT actor_id, last_name from actor ORDER BY actor_id";

    echo "=====================\n";
    echo "Character Set: $charset\n";
    echo "=====================\n";

    if ($result = $mysqli->query($query)) {

        /* Get field information for all columns */
        $finfo = $result->fetch_fields();

        foreach ($finfo as $val) {
            printf("Name:      %s\n",   $val->name);
            printf("Table:     %s\n",   $val->table);
            printf("Max. Len:  %d\n",   $val->max_length);
            printf("Length:    %d\n",   $val->length);
            printf("charsetnr: %d\n",   $val->charsetnr);
            printf("Flags:     %d\n",   $val->flags);
            printf("Type:      %d\n\n", $val->type);
        }
        $result->free();
    }
}
$mysqli->close();
?>
```

Example 3.122 Procedural style

```


<?php
$link = mysqli_connect("127.0.0.1", "my_user", "my_password", "sakila");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

foreach (array('latin1', 'utf8') as $charset) {

    // Set character set, to show its impact on some values (e.g., length in bytes)
    mysqli_set_charset($link, $charset);

    $query = "SELECT actor_id, last_name from actor ORDER BY actor_id";

    echo "=====================\n";
    echo "Character Set: $charset\n";
    echo "=====================\n";

    if ($result = mysqli_query($link, $query)) {

        /* Get field information for all columns */
        $finfo = mysqli_fetch_fields($result);

        foreach ($finfo as $val) {
            printf("Name:      %s\n",   $val->name);
            printf("Table:     %s\n",   $val->table);
            printf("Max. Len:  %d\n",   $val->max_length);
            printf("Length:    %d\n",   $val->length);
            printf("charsetnr: %d\n",   $val->charsetnr);
            printf("Flags:     %d\n",   $val->flags);
```

```
            printf("Type:      %d\n\n", $val->type);
        }
        mysqli_free_result($result);
    }
}

mysqli_close($link);
?>
```

The above examples will output:

```
======================
Character Set: latin1
======================
Name:      actor_id
Table:     actor
Max. Len:  3
Length:    5
charsetnr: 63
Flags:     49699
Type:      2

Name:      last_name
Table:     actor
Max. Len:  12
Length:    45
charsetnr: 8
Flags:     20489
Type:      253

======================
Character Set: utf8
======================
Name:      actor_id
Table:     actor
Max. Len:  3
Length:    5
charsetnr: 63
Flags:     49699
Type:      2

Name:      last_name
Table:     actor
Max. Len:  12
Length:    135
charsetnr: 33
Flags:     20489
```

See Also

mysqli_num_fields
mysqli_fetch_field_direct
mysqli_fetch_field

3.11.9 mysqli_result::fetch_object, mysqli_fetch_object

- mysqli_result::fetch_object

mysqli_fetch_object

Returns the current row of a result set as an object

Description

Object oriented style

```
object mysqli_result::fetch_object(
  string class_name
      =  ="stdClass",
  array params);
```

Procedural style

```
object mysqli_fetch_object(
  mysqli_result result,
  string class_name
      =  ="stdClass",
  array params);
```

The mysqli_fetch_object will return the current row result set as an object where the attributes of the object represent the names of the fields found within the result set.

Note that mysqli_fetch_object sets the properties of the object before calling the object constructor.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.
class_name	The name of the class to instantiate, set the properties of and return. If not specified, a stdClass object is returned.
params	An optional array of parameters to pass to the constructor for *class_name* objects.

Return Values

Returns an object with string properties that corresponds to the fetched row or NULL if there are no more rows in resultset.

> **Note**
>
> Field names returned by this function are *case-sensitive*.

> **Note**
>
> This function sets NULL fields to the PHP NULL value.

Examples

Example 3.123 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
```

```
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER by ID DESC LIMIT 50,5";

if ($result = $mysqli->query($query)) {

    /* fetch object array */
    while ($obj = $result->fetch_object()) {
        printf ("%s (%s)\n", $obj->Name, $obj->CountryCode);
    }

    /* free result set */
    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.124 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER by ID DESC LIMIT 50,5";

if ($result = mysqli_query($link, $query)) {

    /* fetch associative array */
    while ($obj = mysqli_fetch_object($result)) {
        printf ("%s (%s)\n", $obj->Name, $obj->CountryCode);
    }

    /* free result set */
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Pueblo (USA)
Arvada (USA)
Cape Coral (USA)
Green Bay (USA)
Santa Clara (USA)
```

See Also

mysqli_fetch_array
mysqli_fetch_assoc
mysqli_fetch_row
mysqli_query
mysqli_data_seek

3.11.10 mysqli_result::fetch_row, mysqli_fetch_row

- mysqli_result::fetch_row

  mysqli_fetch_row

  Get a result row as an enumerated array

Description

Object oriented style

```
mixed mysqli_result::fetch_row();
```

Procedural style

```
mixed mysqli_fetch_row(
  mysqli_result result);
```

Fetches one row of data from the result set and returns it as an enumerated array, where each column is stored in an array offset starting from 0 (zero). Each subsequent call to this function will return the next row within the result set, or NULL if there are no more rows.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.

Return Values

mysqli_fetch_row returns an array of strings that corresponds to the fetched row or NULL if there are no more rows in result set.

> **Note**
>
> This function sets NULL fields to the PHP NULL value.

Examples

Example 3.125 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
```

```
}

$query = "SELECT Name, CountryCode FROM City ORDER by ID DESC LIMIT 50,5";

if ($result = $mysqli->query($query)) {

    /* fetch object array */
    while ($row = $result->fetch_row()) {
        printf ("%s (%s)\n", $row[0], $row[1]);
    }

    /* free result set */
    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.126 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, CountryCode FROM City ORDER by ID DESC LIMIT 50,5";

if ($result = mysqli_query($link, $query)) {

    /* fetch associative array */
    while ($row = mysqli_fetch_row($result)) {
        printf ("%s (%s)\n", $row[0], $row[1]);
    }

    /* free result set */
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Pueblo (USA)
Arvada (USA)
Cape Coral (USA)
Green Bay (USA)
Santa Clara (USA)
```

See Also

mysqli_fetch_array
mysqli_fetch_assoc
mysqli_fetch_object
mysqli_query
mysqli_data_seek

3.11.11 mysqli_result::$field_count, mysqli_num_fields

- mysqli_result::$field_count

  mysqli_num_fields

  Get the number of fields in a result

Description

Object oriented style

```
int
  mysqli_result->field_count ;
```

Procedural style

```
int mysqli_num_fields(
  mysqli_result result);
```

Returns the number of fields from specified result set.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.

Return Values

The number of fields from a result set.

Examples

Example 3.127 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

if ($result = $mysqli->query("SELECT * FROM City ORDER BY ID LIMIT 1")) {

    /* determine number of fields in result set */
    $field_cnt = $result->field_count;

    printf("Result set has %d fields.\n", $field_cnt);

    /* close result set */
```

```
    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.128 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

if ($result = mysqli_query($link, "SELECT * FROM City ORDER BY ID LIMIT 1")) {

    /* determine number of fields in result set */
    $field_cnt = mysqli_num_fields($result);

    printf("Result set has %d fields.\n", $field_cnt);

    /* close result set */
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Result set has 5 fields.
```

See Also

mysqli_fetch_field

3.11.12 mysqli_result::field_seek, mysqli_field_seek

- mysqli_result::field_seek

  mysqli_field_seek

  Set result pointer to a specified field offset

Description

Object oriented style

```
bool mysqli_result::field_seek(
  int fieldnr);
```

Procedural style

```
bool mysqli_field_seek(
  mysqli_result result,
  int fieldnr);
```

Sets the field cursor to the given offset. The next call to mysqli_fetch_field will retrieve the field definition of the column associated with that offset.

> **Note**
>
> To seek to the beginning of a row, pass an offset value of zero.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.
fieldnr	The field number. This value must be in the range from 0 to number of fields - 1.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 3.129 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, SurfaceArea from Country ORDER BY Code LIMIT 5";

if ($result = $mysqli->query($query)) {

    /* Get field information for 2nd column */
    $result->field_seek(1);
    $finfo = $result->fetch_field();

    printf("Name:     %s\n", $finfo->name);
    printf("Table:    %s\n", $finfo->table);
    printf("max. Len: %d\n", $finfo->max_length);
    printf("Flags:    %d\n", $finfo->flags);
    printf("Type:     %d\n\n", $finfo->type);

    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.130 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT Name, SurfaceArea from Country ORDER BY Code LIMIT 5";

if ($result = mysqli_query($link, $query)) {

    /* Get field information for 2nd column */
    mysqli_field_seek($result, 1);
    $finfo = mysqli_fetch_field($result);

    printf("Name:     %s\n", $finfo->name);
    printf("Table:    %s\n", $finfo->table);
    printf("max. Len: %d\n", $finfo->max_length);
    printf("Flags:    %d\n", $finfo->flags);
    printf("Type:     %d\n\n", $finfo->type);

    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Name:     SurfaceArea
Table:    Country
max. Len: 10
Flags:    32769
Type:     4
```

See Also

mysqli_fetch_field

3.11.13 mysqli_result::free, mysqli_free_result

- mysqli_result::free

  mysqli_free_result

  Frees the memory associated with a result

Description

Object oriented style

```
void mysqli_result::free();
```

```
void mysqli_result::close();
```

```
void mysqli_result::free_result();
```

Procedural style

```
void mysqli_free_result(
  mysqli_result result);
```

Frees the memory associated with the result.

> **Note**
>
> You should always free your result with mysqli_free_result, when your result object is not needed anymore.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.

Return Values

No value is returned.

See Also

mysqli_query
mysqli_stmt_store_result
mysqli_store_result
mysqli_use_result

3.11.14 mysqli_result::$lengths, mysqli_fetch_lengths

- mysqli_result::$lengths

  mysqli_fetch_lengths

  Returns the lengths of the columns of the current row in the result set

Description

Object oriented style

```
array
  mysqli_result->lengths ;
```

Procedural style

```
array mysqli_fetch_lengths(
  mysqli_result result);
```

The mysqli_fetch_lengths function returns an array containing the lengths of every column of the current row within the result set.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.

Return Values

An array of integers representing the size of each column (not including any terminating null characters). FALSE if an error occurred.

mysqli_fetch_lengths is valid only for the current row of the result set. It returns FALSE if you call it before calling mysqli_fetch_row/array/object or after retrieving all rows in the result.

Examples

Example 3.131 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

$query = "SELECT * from Country ORDER BY Code LIMIT 1";

if ($result = $mysqli->query($query)) {

    $row = $result->fetch_row();

    /* display column lengths */
    foreach ($result->lengths as $i => $val) {
        printf("Field %2d has Length %2d\n", $i+1, $val);
    }
    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.132 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}
```

```
$query = "SELECT * from Country ORDER BY Code LIMIT 1";

if ($result = mysqli_query($link, $query)) {

    $row = mysqli_fetch_row($result);

    /* display column lengths */
    foreach (mysqli_fetch_lengths($result) as $i => $val) {
        printf("Field %2d has Length %2d\n", $i+1, $val);
    }
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Field  1 has Length  3
Field  2 has Length  5
Field  3 has Length 13
Field  4 has Length  9
Field  5 has Length  6
Field  6 has Length  1
Field  7 has Length  6
Field  8 has Length  4
Field  9 has Length  6
Field 10 has Length  6
Field 11 has Length  5
Field 12 has Length 44
Field 13 has Length  7
Field 14 has Length  3
Field 15 has Length  2
```

3.11.15 mysqli_result::$num_rows, mysqli_num_rows

- mysqli_result::$num_rows

  mysqli_num_rows

  Gets the number of rows in a result

Description

Object oriented style

```
  int
    mysqli_result->num_rows ;
```

Procedural style

```
  int mysqli_num_rows(
    mysqli_result result);
```

Returns the number of rows in the result set.

The behaviour of mysqli_num_rows depends on whether buffered or unbuffered result sets are being used. For unbuffered result sets, mysqli_num_rows will not return the correct number of rows until all the rows in the result have been retrieved.

Parameters

result	Procedural style only: A result set identifier returned by mysqli_query, mysqli_store_result or mysqli_use_result.

Return Values

Returns number of rows in the result set.

> **Note**
>
> If the number of rows is greater than PHP_INT_MAX, the number will be returned as a string.

Examples

Example 3.133 Object oriented style

```
<?php
$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

if ($result = $mysqli->query("SELECT Code, Name FROM Country ORDER BY Name")) {

    /* determine number of rows result set */
    $row_cnt = $result->num_rows;

    printf("Result set has %d rows.\n", $row_cnt);

    /* close result set */
    $result->close();
}

/* close connection */
$mysqli->close();
?>
```

Example 3.134 Procedural style

```
<?php
$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}
```

```
if ($result = mysqli_query($link, "SELECT Code, Name FROM Country ORDER BY Name")) {

    /* determine number of rows result set */
    $row_cnt = mysqli_num_rows($result);

    printf("Result set has %d rows.\n", $row_cnt);

    /* close result set */
    mysqli_free_result($result);
}

/* close connection */
mysqli_close($link);
?>
```

The above examples will output:

```
Result set has 239 rows.
```

See Also

mysqli_affected_rows
mysqli_store_result
mysqli_use_result
mysqli_query

3.12 The mysqli_driver class

MySQLi Driver.

```
mysqli_driver {
mysqli_driver

      Properties

  public readonly string
    client_info ;

  public readonly string
    client_version ;

  public readonly string
    driver_version ;

  public readonly string
    embedded ;

  public bool
    reconnect ;

  public int
    report_mode ;

Methods
```

```
  void mysqli_driver::embedded_server_end();

  bool mysqli_driver::embedded_server_start(
    bool start,
    array arguments,
    array groups);

}
```

client_info	The Client API header version
client_version	The Client version
driver_version	The MySQLi Driver version
embedded	Whether MySQLi Embedded support is enabled
reconnect	Allow or prevent reconnect (see the mysqli.reconnect INI directive)
report_mode	Set to MYSQLI_REPORT_OFF, MYSQLI_REPORT_ALL or any combination of MYSQLI_REPORT_STRICT (throw Exceptions for errors), MYSQLI_REPORT_ERROR (report errors) and MYSQLI_REPORT_INDEX (errors regarding indexes). See also mysqli_report.

3.12.1 mysqli_driver::embedded_server_end, mysqli_embedded_server_end

- mysqli_driver::embedded_server_end

  mysqli_embedded_server_end

  Stop embedded server

Description

Object oriented style

```
  void mysqli_driver::embedded_server_end();
```

Procedural style

```
  void mysqli_embedded_server_end();
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

3.12.2 mysqli_driver::embedded_server_start, mysqli_embedded_server_start

- mysqli_driver::embedded_server_start

  mysqli_embedded_server_start

Initialize and start embedded server

Description

Object oriented style

```
bool mysqli_driver::embedded_server_start(
  bool start,
  array arguments,
  array groups);
```

Procedural style

```
bool mysqli_embedded_server_start(
  bool start,
  array arguments,
  array groups);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

3.12.3 mysqli_driver::$report_mode, mysqli_report

- `mysqli_driver::$report_mode`

  `mysqli_report`

  Enables or disables internal report functions

Description

Object oriented style

```
int
  mysqli_driver->report_mode ;
```

Procedural style

```
bool mysqli_report(
  int flags);
```

A function helpful in improving queries during code development and testing. Depending on the flags, it reports errors from mysqli function calls or queries that don't use an index (or use a bad index).

Parameters

`flags`

Table 3.20 Supported flags

Name	Description
`MYSQLI_REPORT_OFF`	Turns reporting off
`MYSQLI_REPORT_ERROR`	Report errors from mysqli function calls
`MYSQLI_REPORT_STRICT`	Throw `mysqli_sql_exception` for errors instead of warnings

Name	Description
MYSQLI_REPORT_INDEX	Report if no index or bad index was used in a query
MYSQLI_REPORT_ALL	Set all options (report all)

Return Values

Returns TRUE on success or FALSE on failure.

Changelog

Version	Description
5.3.4	Changing the reporting mode is now be per-request, rather than per-process.
5.2.15	Changing the reporting mode is now be per-request, rather than per-process.

Examples

Example 3.135 Object oriented style

```
<?php

$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* activate reporting */
$driver = new mysqli_driver();
$driver->report_mode = MYSQLI_REPORT_ALL;

try {

    /* this query should report an error */
    $result = $mysqli->query("SELECT Name FROM Nonexistingtable WHERE population > 50000");

    /* this query should report a bad index */
    $result = $mysqli->query("SELECT Name FROM City WHERE population > 50000");

    $result->close();

    $mysqli->close();

} catch (mysqli_sql_exception $e) {

    echo $e->__toString();
}
?>
```

Example 3.136 Procedural style

```
<?php
/* activate reporting */
mysqli_report(MYSQLI_REPORT_ALL);

$link = mysqli_connect("localhost", "my_user", "my_password", "world");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}

/* this query should report an error */
$result = mysqli_query("SELECT Name FROM Nonexistingtable WHERE population > 50000");

/* this query should report a bad index */
$result = mysqli_query("SELECT Name FROM City WHERE population > 50000");

mysqli_free_result($result);

mysqli_close($link);
?>
```

See Also

mysqli_debug
mysqli_dump_debug_info
mysqli_sql_exception
set_exception_handler
error_reporting

3.13 The mysqli_warning class

Represents a MySQL warning.

```
mysqli_warning {
mysqli_warning

      Properties

  public
    message ;

  public
    sqlstate ;

  public
    errno ;

Methods

  protected mysqli_warning::__construct();

  public void mysqli_warning::next();

}
```

message	Message string

sqlstate	SQL state
errno	Error number

3.13.1 mysqli_warning::__construct

- mysqli_warning::__construct

  The __construct purpose

Description

```
protected mysqli_warning::__construct();
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

This function has no parameters.

Return Values

3.13.2 mysqli_warning::next

- mysqli_warning::next

  The next purpose

Description

```
public void mysqli_warning::next();
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

This function has no parameters.

Return Values

3.14 The mysqli_sql_exception class

The mysqli exception handling class.

```
mysqli_sql_exception {
mysqli_sql_exceptionextends RuntimeException

      Properties

  protected string
    sqlstate ;

Inherited properties

  protected string
    message ;

  protected int
    code ;

  protected string
    file ;

  protected int
    line ;

}
```

`sqlstate`	The sql state with the error.

3.15 Aliases and deprecated Mysqli Functions

3.15.1 `mysqli_bind_param`

- `mysqli_bind_param`

  Alias for `mysqli_stmt_bind_param`

Description

This function is an alias of `mysqli_stmt_bind_param`.

> **Warning**
>
> This function has been *DEPRECATED* as of PHP 5.3.0 and *REMOVED* as of PHP 5.4.0.

See Also

`mysqli_stmt_bind_param`

3.15.2 `mysqli_bind_result`

- `mysqli_bind_result`

  Alias for `mysqli_stmt_bind_result`

Description

This function is an alias of mysqli_stmt_bind_result.

> **Warning**
>
> This function has been *DEPRECATED* as of PHP 5.3.0 and *REMOVED* as of PHP 5.4.0.

See Also

mysqli_stmt_bind_result

3.15.3 mysqli_client_encoding

- mysqli_client_encoding

  Alias of mysqli_character_set_name

Description

This function is an alias of mysqli_character_set_name.

> **Warning**
>
> This function has been *DEPRECATED* as of PHP 5.3.0 and *REMOVED* as of PHP 5.4.0.

See Also

mysqli_real_escape_string

3.15.4 mysqli_connect

- mysqli_connect

  Alias of mysqli::__construct

Description

This function is an alias of: mysqli::__construct

Although the mysqli::__construct documentation also includes procedural examples that use the mysqli_connect function, here is a short example:

Examples

Example 3.137 mysqli_connect example

```
<?php
$link = mysqli_connect("127.0.0.1", "my_user", "my_password", "my_db");

if (!$link) {
    echo "Error: Unable to connect to MySQL." . PHP_EOL;
    echo "Debugging errno: " . mysqli_connect_errno() . PHP_EOL;
```

```
    echo "Debugging error: " . mysqli_connect_error() . PHP_EOL;
    exit;
}

echo "Success: A proper connection to MySQL was made! The my_db database is great." . PHP_EOL;
echo "Host information: " . mysqli_get_host_info($link) . PHP_EOL;

mysqli_close($link);
?>
```

The above examples will output:

```
Success: A proper connection to MySQL was made! The my_db database is great.
Host information: localhost via TCP/IP
```

3.15.5 mysqli::disable_reads_from_master, mysqli_disable_reads_from_master

- mysqli::disable_reads_from_master

  mysqli_disable_reads_from_master

  Disable reads from master

Description

Object oriented style

```
void mysqli::disable_reads_from_master();
```

Procedural style

```
bool mysqli_disable_reads_from_master(
  mysqli link);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

> **Warning**
>
> This function has been *DEPRECATED* and *REMOVED* as of PHP 5.3.0.

3.15.6 mysqli_disable_rpl_parse

- mysqli_disable_rpl_parse

  Disable RPL parse

Description

```
bool mysqli_disable_rpl_parse(
  mysqli link);
```

Warning

This function is currently not documented; only its argument list is available.

Warning

This function has been *DEPRECATED* and *REMOVED* as of PHP 5.3.0.

3.15.7 mysqli_enable_reads_from_master

- `mysqli_enable_reads_from_master`

  Enable reads from master

Description

```
bool mysqli_enable_reads_from_master(
  mysqli link);
```

Warning

This function is currently not documented; only its argument list is available.

Warning

This function has been *DEPRECATED* and *REMOVED* as of PHP 5.3.0.

3.15.8 mysqli_enable_rpl_parse

- `mysqli_enable_rpl_parse`

  Enable RPL parse

Description

```
bool mysqli_enable_rpl_parse(
  mysqli link);
```

Warning

This function is currently not documented; only its argument list is available.

Warning

This function has been *DEPRECATED* and *REMOVED* as of PHP 5.3.0.

3.15.9 mysqli_escape_string

- `mysqli_escape_string`

Alias of mysqli_real_escape_string

Description

This function is an alias of: mysqli_real_escape_string.

3.15.10 mysqli_execute

- mysqli_execute

  Alias for mysqli_stmt_execute

Description

This function is an alias of mysqli_stmt_execute.

Notes

> **Note**
>
> mysqli_execute is deprecated and will be removed.

See Also

mysqli_stmt_execute

3.15.11 mysqli_fetch

- mysqli_fetch

  Alias for mysqli_stmt_fetch

Description

This function is an alias of mysqli_stmt_fetch.

> **Warning**
>
> This function has been *DEPRECATED* as of PHP 5.3.0 and *REMOVED* as of PHP 5.4.0.

See Also

mysqli_stmt_fetch

3.15.12 mysqli_get_cache_stats

- mysqli_get_cache_stats

  Returns client Zval cache statistics

> **Warning**
>
> This function has been *REMOVED* as of PHP 5.4.0.

Description

```
array mysqli_get_cache_stats();
```

Returns an empty array. Available only with mysqlnd.

Parameters

Return Values

Returns an empty array on success, FALSE otherwise.

Changelog

Version	Description
5.4.0	The `mysqli_get_cache_stats` was removed.
5.3.0	The `mysqli_get_cache_stats` was added as stub.

3.15.13 `mysqli_get_links_stats`

- `mysqli_get_links_stats`

  Return information about open and cached links

Description

```
array mysqli_get_links_stats();
```

`mysqli_get_links_stats` returns information about open and cached MySQL links.

Parameters

This function has no parameters.

Return Values

`mysqli_get_links_stats` returns an associative array with three elements, keyed as follows:

`total`	An integer indicating the total number of open links in any state.
`active_plinks`	An integer representing the number of active persistent connections.
`cached_plinks`	An integer representing the number of inactive persistent connections.

3.15.14 `mysqli_get_metadata`

- `mysqli_get_metadata`

Alias for mysqli_stmt_result_metadata

Description

This function is an alias of mysqli_stmt_result_metadata.

> **Warning**
>
> This function has been *DEPRECATED* as of PHP 5.3.0 and *REMOVED* as of PHP 5.4.0.

See Also

mysqli_stmt_result_metadata

3.15.15 mysqli_master_query

- mysqli_master_query

  Enforce execution of a query on the master in a master/slave setup

Description

```
bool mysqli_master_query(
  mysqli link,
  string query);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

> **Warning**
>
> This function has been *DEPRECATED* and *REMOVED* as of PHP 5.3.0.

3.15.16 mysqli_param_count

- mysqli_param_count

  Alias for mysqli_stmt_param_count

Description

This function is an alias of mysqli_stmt_param_count.

> **Warning**
>
> This function has been *DEPRECATED* as of PHP 5.3.0 and *REMOVED* as of PHP 5.4.0.

See Also

mysqli_stmt_param_count

3.15.17 mysqli_report

- mysqli_report

  Alias of mysqli_driver->report_mode

Description

This function is an alias of: mysqli_driver->report_mode

3.15.18 mysqli_rpl_parse_enabled

- mysqli_rpl_parse_enabled

  Check if RPL parse is enabled

Description

```
int mysqli_rpl_parse_enabled(
  mysqli link);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

> **Warning**
>
> This function has been *DEPRECATED* and *REMOVED* as of PHP 5.3.0.

3.15.19 mysqli_rpl_probe

- mysqli_rpl_probe

  RPL probe

Description

```
bool mysqli_rpl_probe(
  mysqli link);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

> **Warning**
>
> This function has been *DEPRECATED* and *REMOVED* as of PHP 5.3.0.

3.15.20 mysqli_send_long_data

- mysqli_send_long_data

  Alias for mysqli_stmt_send_long_data

Description

This function is an alias of mysqli_stmt_send_long_data.

> **Warning**
>
> This function has been *DEPRECATED* as of PHP 5.3.0 and *REMOVED* as of PHP 5.4.0.

See Also

mysqli_stmt_send_long_data

3.15.21 mysqli::set_opt, mysqli_set_opt

- mysqli::set_opt

  mysqli_set_opt

  Alias of mysqli_options

Description

This function is an alias of mysqli_options.

3.15.22 mysqli_slave_query

- mysqli_slave_query

  Force execution of a query on a slave in a master/slave setup

Description

```
bool mysqli_slave_query(
  mysqli link,
  string query);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

> **Warning**
>
> This function has been *DEPRECATED* and *REMOVED* as of PHP 5.3.0.

3.16 Changelog

The following changes have been made to classes/functions/methods of this extension.

Chapter 4 MySQL Functions (PDO_MYSQL)

Table of Contents

PDO_MYSQL is a driver that implements the PHP Data Objects (PDO) interface to enable access from PHP to MySQL 3.x, 4.x and 5.x databases.

PDO_MYSQL will take advantage of native prepared statement support present in MySQL 4.1 and higher. If you're using an older version of the mysql client libraries, PDO will emulate them for you.

> **Warning**
>
> Beware: Some MySQL table types (storage engines) do not support transactions. When writing transactional database code using a table type that does not support transactions, MySQL will pretend that a transaction was initiated successfully. In addition, any DDL queries issued will implicitly commit any pending transactions.

The common Unix distributions include binary versions of PHP that can be installed. Although these binary versions are typically built with support for the MySQL extensions, the extension libraries themselves may need to be installed using an additional package. Check the package manager than comes with your chosen distribution for availability.

For example, on Ubuntu the `php5-mysql` package installs the ext/mysql, ext/mysqli, and PDO_MYSQL PHP extensions. On CentOS, the `php-mysql` package also installs these three PHP extensions.

Alternatively, you can compile this extension yourself. Building PHP from source allows you to specify the MySQL extensions you want to use, as well as your choice of client library for each extension.

When compiling, use `--with-pdo-mysql[=DIR]` to install the PDO MySQL extension, where the optional `[=DIR]` is the MySQL base library. As of PHP 5.4, mysqlnd is the default library. For details about choosing a library, see Choosing a MySQL library.

Optionally, the `--with-mysql-sock[=DIR]` sets to location to the MySQL unix socket pointer for all MySQL extensions, including PDO_MYSQL. If unspecified, the default locations are searched.

Optionally, the `--with-zlib-dir[=DIR]` is used to set the path to the libz install prefix.

```
$ ./configure --with-pdo-mysql --with-mysql-sock=/var/mysql/mysql.sock
```

SSL support is enabled using the appropriate PDO_MySQL constants, which is equivalent to calling the MySQL C API function mysql_ssl_set(). Also, SSL cannot be enabled with `PDO::setAttribute` because the connection already exists. See also the MySQL documentation about connecting to MySQL with SSL.

Table 4.1 Changelog

Version	Description
5.4.0	mysqlnd became the default MySQL library when compiling PDO_MYSQL. Previously, libmysqlclient was the default MySQL library.
5.4.0	MySQL client libraries 4.1 and below are no longer supported.
5.3.9	Added SSL support with mysqlnd and OpenSSL.
5.3.7	Added SSL support with libmysqlclient and OpenSSL.

The constants below are defined by this driver, and will only be available when the extension has been either compiled into PHP or dynamically loaded at runtime. In addition, these driver-specific constants should only be used if you are using this driver. Using driver-specific attributes with another driver may result in unexpected behaviour. `PDO::getAttribute` may be used to obtain the `PDO_ATTR_DRIVER_NAME` attribute to check the driver, if your code can run against multiple drivers.

`PDO::MYSQL_ATTR_USE_BUFFERED_QUERY` (integer)

If this attribute is set to `TRUE` on a `PDOStatement`, the MySQL driver will use the buffered versions of the MySQL API. If you're writing portable code, you should use `PDOStatement::fetchAll` instead.

Example 4.1 Forcing queries to be buffered in mysql

```
<?php
if ($db->getAttribute(PDO::ATTR_DRIVER_NAME) == 'mysql') {
    $stmt = $db->prepare('select * from foo',
        array(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY => true));
} else {
    die("my application only works with mysql; I should use \$stmt->fetchAll()
}
?>
```

`PDO::MYSQL_ATTR_LOCAL_INFILE` (integer)

Enable `LOAD LOCAL INFILE`.

Note, this constant can only be used in the *`driver_options`* array when constructing a new database handle.

`PDO::MYSQL_ATTR_INIT_COMMAND` (integer)

Command to execute when connecting to the MySQL server. Will automatically be re-executed when reconnecting.

Note, this constant can only be used in the *`driver_options`* array when constructing a new database handle.

`PDO::MYSQL_ATTR_READ_DEFAULT_FILE` (integer)

Read options from the named option file instead of from `my.cnf`. This option is not available if mysqlnd is used, because mysqlnd does not read the mysql configuration files.

`PDO::MYSQL_ATTR_READ_DEFAULT_GROUP` (integer)

Read options from the named group from `my.cnf` or the file specified with `MYSQL_READ_DEFAULT_FILE`. This option is not available if mysqlnd is used, because mysqlnd does not read the mysql configuration files.

`PDO::MYSQL_ATTR_MAX_BUFFER_SIZE` (integer)	Maximum buffer size. Defaults to 1 MiB. This constant is not supported when compiled against mysqlnd.
`PDO::MYSQL_ATTR_DIRECT_QUERY` (integer)	Perform direct queries, don't use prepared statements.
`PDO::MYSQL_ATTR_FOUND_ROWS` (integer)	Return the number of found (matched) rows, not the number of changed rows.
`PDO::MYSQL_ATTR_IGNORE_SPACE` (integer)	Permit spaces after function names. Makes all functions names reserved words.
`PDO::MYSQL_ATTR_COMPRESS` (integer)	Enable network communication compression. This is also supported when compiled against mysqlnd as of PHP 5.3.11.
`PDO::MYSQL_ATTR_SSL_CA` (integer)	The file path to the SSL certificate authority. This exists as of PHP 5.3.7.
`PDO::MYSQL_ATTR_SSL_CAPATH` (integer)	The file path to the directory that contains the trusted SSL CA certificates, which are stored in PEM format. This exists as of PHP 5.3.7.
`PDO::MYSQL_ATTR_SSL_CERT` (integer)	The file path to the SSL certificate. This exists as of PHP 5.3.7.
`PDO::MYSQL_ATTR_SSL_CIPHER` (integer)	A list of one or more permissible ciphers to use for SSL encryption, in a format understood by OpenSSL. For example: `DHE-RSA-AES256-SHA:AES128-SHA` This exists as of PHP 5.3.7.
`PDO::MYSQL_ATTR_SSL_KEY` (integer)	The file path to the SSL key. This exists as of PHP 5.3.7.
`PDO::MYSQL_ATTR_MULTI_STATEMENTS` (integer)	Disables multi query execution in both `PDO::prepare` and `PDO::query` when set to `FALSE`. Note, this constant can only be used in the *`driver_options`* array when constructing a new database handle. This exists as of PHP 5.5.21 and PHP 5.6.5.

The behaviour of these functions is affected by settings in `php.ini`.

Table 4.2 PDO_MYSQL Configuration Options

Name	Default	Changeable
pdo_mysql.default_socket	"/tmp/mysql.sock"	PHP_INI_SYSTEM
pdo_mysql.debug	NULL	PHP_INI_SYSTEM

For further details and definitions of the PHP_INI_* modes, see the http://www.php.net/manual/en/configuration.changes.modes.

Here's a short explanation of the configuration directives.

pdo_mysql.default_socket string	Sets a Unix domain socket. This value can either be set at compile time if a domain socket is found at configure. This ini setting is Unix only.
pdo_mysql.debug boolean	Enables debugging for PDO_MYSQL. This setting is only available when PDO_MYSQL is compiled against mysqlnd and in PDO debug mode.

4.1 PDO_MYSQL DSN

- PDO_MYSQL DSN

  Connecting to MySQL databases

Description

The PDO_MYSQL Data Source Name (DSN) is composed of the following elements:

DSN prefix	The DSN prefix is **mysql:**.
host	The hostname on which the database server resides.
port	The port number where the database server is listening.
dbname	The name of the database.
unix_socket	The MySQL Unix socket (shouldn't be used with host or port).
charset	The character set. See the character set concepts documentation for more information. Prior to PHP 5.3.6, this element was silently ignored. The same behaviour can be partly replicated with the PDO::MYSQL_ATTR_INIT_COMMAND driver option, as the following example shows.

> **Warning**
>
> The method in the below example can only be used with character sets that share the same lower 7 bit representation as ASCII, such as ISO-8859-1 and UTF-8. Users using character sets that have different representations (such as UTF-16 or Big5) *must* use the charset option provided in PHP 5.3.6 and later versions.

Example 4.2 Setting the connection character set to UTF-8 prior to PHP 5.3.6

```
<?php
$dsn = 'mysql:host=localhost;dbname=testdb';
$username = 'username';
$password = 'password';
```

```
$options = array(
    PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8',
);

$dbh = new PDO($dsn, $username, $password, $options);
?>
```

Changelog

Version	Description
5.3.6	Prior to version 5.3.6, `charset` was ignored.

Examples

Example 4.3 PDO_MYSQL DSN examples

The following example shows a PDO_MYSQL DSN for connecting to MySQL databases:

```
mysql:host=localhost;dbname=testdb
```

More complete examples:

```
mysql:host=localhost;port=3307;dbname=testdb
mysql:unix_socket=/tmp/mysql.sock;dbname=testdb
```

Notes

> **Unix only:**
>
> When the host name is set to `"localhost"`, then the connection to the server is made thru a domain socket. If PDO_MYSQL is compiled against libmysqlclient then the location of the socket file is at libmysqlclient's compiled in location. If PDO_MYSQL is compiled against mysqlnd a default socket can be set thru the pdo_mysql.default_socket setting.

Chapter 5 Original MySQL API

Table of Contents

This extension is deprecated as of PHP 5.5.0, and has been removed as of PHP 7.0.0. Instead, either the mysqli or PDO_MySQL extension should be used. See also the MySQL API Overview for further help while choosing a MySQL API.

These functions allow you to access MySQL database servers. More information about MySQL can be found at http://www.mysql.com/.

Documentation for MySQL can be found at http://dev.mysql.com/doc/.

5.1 Installing/Configuring

5.1.1 Requirements

In order to have these functions available, you must compile PHP with MySQL support.

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:

5.1.2 Installation

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:

For compiling, simply use the `--with-mysql[=DIR]` configuration option where the optional `[DIR]` points to the MySQL installation directory.

Although this MySQL extension is compatible with MySQL 4.1.0 and greater, it doesn't support the extra functionality that these versions provide. For that, use the MySQLi extension.

If you would like to install the mysql extension along with the mysqli extension you have to use the same client library to avoid any conflicts.

5.1.2.1 Installation on Linux Systems

Note: `[DIR]` is the path to the MySQL client library files (*headers and libraries*), which can be downloaded from MySQL.

Table 5.1 ext/mysql compile time support matrix

PHP Version	Default	Configure Options: mysqlnd	Configure Options: `libmysqlclient`	Changelog
4.x.x	libmysqlclient	Not Available	`--without-mysql` to disable	MySQL enabled by default, MySQL client libraries are bundled
5.0.x, 5.1.x, 5.2.x	libmysqlclient	Not Available	`--with-mysql=[DIR]`	MySQL is no longer enabled by default, and the MySQL client libraries are no longer bundled
5.3.x	libmysqlclient	`--with-mysql=mysqlnd`	`--with-mysql=[DIR]`	mysqlnd is now available
5.4.x	mysqlnd	`--with-mysql`	`--with-mysql=[DIR]`	mysqlnd is now the default

5.1.2.2 Installation on Windows Systems

PHP 5.0.x, 5.1.x, 5.2.x

MySQL is no longer enabled by default, so the `php_mysql.dll` DLL must be enabled inside of `php.ini`. Also, PHP needs access to the MySQL client library. A file named `libmysql.dll` is included in the Windows PHP distribution and in order for PHP to talk to MySQL this file needs to be available to the Windows systems `PATH`. See the FAQ titled "How do I add my PHP directory to the `PATH` on Windows" for information on how to do this. Although copying `libmysql.dll` to the Windows system directory also works (because the system directory is by default in the system's `PATH`), it's not recommended.

As with enabling any PHP extension (such as `php_mysql.dll`), the PHP directive extension_dir should be set to the directory where the PHP extensions are located. See also the Manual Windows Installation Instructions. An example extension_dir value for PHP 5 is `c:\php\ext`

> **Note**
>
> If when starting the web server an error similar to the following occurs: `"Unable to load dynamic library './php_mysql.dll'"`, this is because `php_mysql.dll` and/or `libmysql.dll` cannot be found by the system.

PHP 5.3.0+

The MySQL Native Driver is enabled by default. Include `php_mysql.dll`, but `libmysql.dll` is no longer required or used.

5.1.2.3 MySQL Installation Notes

> **Warning**
>
> Crashes and startup problems of PHP may be encountered when loading this extension in conjunction with the recode extension. See the recode extension for more information.

> **Note**
>
> If you need charsets other than *latin* (default), you have to install external (not bundled) libmysqlclient with compiled charset support.

5.1.3 Runtime Configuration

The behaviour of these functions is affected by settings in `php.ini`.

Table 5.2 MySQL Configuration Options

Name	Default	Changeable	Changelog
mysql.allow_local_infile	"1"	PHP_INI_SYSTEM	
mysql.allow_persistent	"1"	PHP_INI_SYSTEM	
mysql.max_persistent	"-1"	PHP_INI_SYSTEM	
mysql.max_links	"-1"	PHP_INI_SYSTEM	
mysql.trace_mode	"0"	PHP_INI_ALL	Available since PHP 4.3.0.
mysql.default_port	NULL	PHP_INI_ALL	
mysql.default_socket	NULL	PHP_INI_ALL	Available since PHP 4.0.1.
mysql.default_host	NULL	PHP_INI_ALL	
mysql.default_user	NULL	PHP_INI_ALL	
mysql.default_password	NULL	PHP_INI_ALL	
mysql.connect_timeout	"60"	PHP_INI_ALL	PHP_INI_SYSTEM in PHP <= 4.3.2. Available since PHP 4.3.0.

For further details and definitions of the PHP_INI_* modes, see the http://www.php.net/manual/en/configuration.changes.modes.

Here's a short explanation of the configuration directives.

mysql.allow_local_infile integer	Allow accessing, from PHP's perspective, local files with LOAD DATA statements
mysql.allow_persistent boolean	Whether to allow persistent connections to MySQL.
mysql.max_persistent integer	The maximum number of persistent MySQL connections per process.
mysql.max_links integer	The maximum number of MySQL connections per process, including persistent connections.
mysql.trace_mode boolean	Trace mode. When `mysql.trace_mode` is enabled, warnings for table/index scans, non free result sets, and SQL-Errors will be displayed. (Introduced in PHP 4.3.0)
mysql.default_port string	The default TCP port number to use when connecting to the database server if no other port is specified. If no default is specified, the port will be obtained from the `MYSQL_TCP_PORT` environment variable, the `mysql-tcp` entry in `/etc/services` or the compile-time `MYSQL_PORT` constant, in that order. Win32 will only use the `MYSQL_PORT` constant.
mysql.default_socket string	The default socket name to use when connecting to a local database server if no other socket name is specified.
mysql.default_host string	The default server host to use when connecting to the database server if no other host is specified. Doesn't apply in SQL safe mode.
mysql.default_user string	The default user name to use when connecting to the database server if no other name is specified. Doesn't apply in SQL safe mode.
mysql.default_password string	The default password to use when connecting to the database server if no other password is specified. Doesn't apply in SQL safe mode.
mysql.connect_timeout integer	Connect timeout in seconds. On Linux this timeout is also used for waiting for the first answer from the server.

5.1.4 Resource Types

There are two resource types used in the MySQL module. The first one is the link identifier for a database connection, the second a resource which holds the result of a query.

5.2 Changelog

The following changes have been made to classes/functions/methods of this extension.

General Changelog for the ext/mysql extension

This changelog references the ext/mysql extension.

Global ext/mysql changes

The following is a list of changes to the entire ext/mysql extension.

Version	Description
7.0.0	This extension was removed from PHP. For details, see Section 2.3, “Choosing an API”.
5.5.0	This extension has been deprecated. Connecting to a MySQL database via `mysql_connect`, `mysql_pconnect` or an implicit connection via any other `mysql_*` function will generate an `E_DEPRECATED` error.
5.5.0	All of the old deprecated functions and aliases now emit `E_DEPRECATED` errors. These functions are: mysql(), mysql_fieldname(), mysql_fieldtable(), mysql_fieldlen(), mysql_fieldtype(), mysql_fieldflags(), mysql_selectdb(), mysql_createdb(), mysql_dropdb(), mysql_freeresult(), mysql_numfields(), mysql_numrows(), mysql_listdbs(), mysql_listtables(), mysql_listfields(), mysql_db_name(), mysql_dbname(), mysql_tablename(), and mysql_table_name().

Changes to existing functions

The following list is a compilation of changelog entries from the ext/mysql functions.

5.3 Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

It is possible to specify additional client flags for the `mysql_connect` and `mysql_pconnect` functions. The following constants are defined:

Table 5.3 MySQL client constants

Constant	Description
`MYSQL_CLIENT_COMPRESS`	Use compression protocol
`MYSQL_CLIENT_IGNORE_SPACE`	Allow space after function names
`MYSQL_CLIENT_INTERACTIVE`	Allow interactive_timeout seconds (instead of `wait_timeout`) of inactivity before closing the connection.
`MYSQL_CLIENT_SSL`	Use SSL encryption. This flag is only available with version 4.x of the MySQL client library or newer. Version 3.23.x is bundled both with PHP 4 and Windows binaries of PHP 5.

The function `mysql_fetch_array` uses a constant for the different types of result arrays. The following constants are defined:

Table 5.4 MySQL fetch constants

Constant	Description
`MYSQL_ASSOC`	Columns are returned into the array having the fieldname as the array index.
`MYSQL_BOTH`	Columns are returned into the array having both a numerical index and the fieldname as the array index.
`MYSQL_NUM`	Columns are returned into the array having a numerical index to the fields. This index starts with 0, the first field in the result.

5.4 Examples

5.4.1 MySQL extension overview example

This simple example shows how to connect, execute a query, print resulting rows and disconnect from a MySQL database.

Example 5.1 MySQL extension overview example

```
<?php
// Connecting, selecting database
$link = mysql_connect('mysql_host', 'mysql_user', 'mysql_password')
    or die('Could not connect: ' . mysql_error());
echo 'Connected successfully';
mysql_select_db('my_database') or die('Could not select database');

// Performing SQL query
$query = 'SELECT * FROM my_table';
$result = mysql_query($query) or die('Query failed: ' . mysql_error());

// Printing results in HTML
echo "<table>\n";
while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
    echo "\t<tr>\n";
    foreach ($line as $col_value) {
        echo "\t\t<td>$col_value</td>\n";
    }
    echo "\t</tr>\n";
}
echo "</table>\n";

// Free resultset
mysql_free_result($result);

// Closing connection
mysql_close($link);
?>
```

5.5 MySQL Functions

> **Note**
>
> Most MySQL functions accept *`link_identifier`* as the last optional parameter. If it is not provided, last opened connection is used. If it doesn't exist, connection is tried to establish with default parameters defined in `php.ini`. If it is not successful, functions return `FALSE`.

5.5.1 `mysql_affected_rows`

- `mysql_affected_rows`

  Get number of affected rows in previous MySQL operation

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_affected_rows`
> `PDOStatement::rowCount`

Description

```
int mysql_affected_rows(
  resource link_identifier
      =  =NULL);
```

Get the number of affected rows by the last INSERT, UPDATE, REPLACE or DELETE query associated with *`link_identifier`*.

Parameters

`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

Returns the number of affected rows on success, and -1 if the last query failed.

If the last query was a DELETE query with no WHERE clause, all of the records will have been deleted from the table but this function will return zero with MySQL versions prior to 4.1.2.

When using UPDATE, MySQL will not update columns where the new value is the same as the old value. This creates the possibility that `mysql_affected_rows` may not actually equal the number of rows matched, only the number of rows that were literally affected by the query.

The REPLACE statement first deletes the record with the same primary key and then inserts the new record. This function returns the number of deleted records plus the number of inserted records.

In the case of "INSERT ... ON DUPLICATE KEY UPDATE" queries, the return value will be 1 if an insert was performed, or 2 for an update of an existing row.

Examples

Example 5.2 `mysql_affected_rows` example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
mysql_select_db('mydb');

/* this should return the correct numbers of deleted records */
mysql_query('DELETE FROM mytable WHERE id < 10');
printf("Records deleted: %d\n", mysql_affected_rows());

/* with a where clause that is never true, it should return 0 */
mysql_query('DELETE FROM mytable WHERE 0');
printf("Records deleted: %d\n", mysql_affected_rows());
?>
```

The above example will output something similar to:

```
Records deleted: 10
Records deleted: 0
```

Example 5.3 `mysql_affected_rows` example using transactions

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
mysql_select_db('mydb');

/* Update records */
mysql_query("UPDATE mytable SET used=1 WHERE id < 10");
printf ("Updated records: %d\n", mysql_affected_rows());
mysql_query("COMMIT");
?>
```

The above example will output something similar to:

```
Updated Records: 10
```

Notes

> **Transactions**
>
> If you are using transactions, you need to call `mysql_affected_rows` after your INSERT, UPDATE, or DELETE query, not after the COMMIT.

> **SELECT Statements**
>
> To retrieve the number of rows returned by a SELECT, it is possible to use `mysql_num_rows`.

> **Cascaded Foreign Keys**
>
> `mysql_affected_rows` does not count rows affected implicitly through the use of ON DELETE CASCADE and/or ON UPDATE CASCADE in foreign key constraints.

See Also

`mysql_num_rows`
`mysql_info`

5.5.2 `mysql_client_encoding`

- `mysql_client_encoding`

  Returns the name of the character set

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_character_set_name`

Description

```
string mysql_client_encoding(
  resource link_identifier
      =  =NULL);
```

Retrieves the `character_set` variable from MySQL.

Parameters

`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no

arguments. If no connection is found or established, an E_WARNING level error is generated.

Return Values

Returns the default character set name for the current connection.

Examples

Example 5.4 mysql_client_encoding example

```
<?php
$link    = mysql_connect('localhost', 'mysql_user', 'mysql_password');
$charset = mysql_client_encoding($link);

echo "The current character set is: $charset\n";
?>
```

The above example will output something similar to:

```
The current character set is: latin1
```

See Also

mysql_set_charset
mysql_real_escape_string

5.5.3 mysql_close

- mysql_close

  Close MySQL connection

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> mysqli_close
> PDO: Assign the value of NULL to the PDO object

Description

```
bool mysql_close(
  resource link_identifier
      =  =NULL);
```

mysql_close closes the non-persistent connection to the MySQL server that's associated with the specified link identifier. If *link_identifier* isn't specified, the last opened link is used.

Open non-persistent MySQL connections and result sets are automatically destroyed when a PHP script finishes its execution. So, while explicitly closing open connections and freeing result sets is optional, doing so is recommended. This will immediately return resources to PHP and MySQL, which can improve performance. For related information, see freeing resources

Parameters

link_identifier	The MySQL connection. If the link identifier is not specified, the last link opened by mysql_connect is assumed. If no connection is found or established, an E_WARNING level error is generated.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 5.5 mysql_close example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully';
mysql_close($link);
?>
```

The above example will output:

```
Connected successfully
```

Notes

> **Note**
>
> mysql_close will not close persistent links created by mysql_pconnect. For additional details, see the manual page on persistent connections.

See Also

mysql_connect
mysql_free_result

5.5.4 mysql_connect

- mysql_connect

  Open a connection to a MySQL Server

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> mysqli_connect
> PDO::__construct

Description

```
resource mysql_connect(
  string server
      =  =ini_get("mysql.default_host"),
  string username
      =  =ini_get("mysql.default_user"),
  string password
      =  =ini_get("mysql.default_password"),
  bool new_link
      =  =false,
  int client_flags
      =  =0);
```

Opens or reuses a connection to a MySQL server.

Parameters

server	The MySQL server. It can also include a port number. e.g. "hostname:port" or a path to a local socket e.g. ":/path/to/socket" for the localhost. If the PHP directive mysql.default_host is undefined (default), then the default value is 'localhost:3306'. In SQL safe mode, this parameter is ignored and value 'localhost:3306' is always used.
username	The username. Default value is defined by mysql.default_user. In SQL safe mode, this parameter is ignored and the name of the user that owns the server process is used.
password	The password. Default value is defined by mysql.default_password. In SQL safe mode, this parameter is ignored and empty password is used.
new_link	If a second call is made to mysql_connect with the same arguments, no new link will be established, but instead, the link identifier of the already opened link will be returned. The *new_link* parameter modifies this behavior and makes mysql_connect always open a new link, even if mysql_connect was called before with the same parameters. In SQL safe mode, this parameter is ignored.
client_flags	The *client_flags* parameter can be a combination of the following constants: 128 (enable LOAD DATA LOCAL handling), MYSQL_CLIENT_SSL, MYSQL_CLIENT_COMPRESS, MYSQL_CLIENT_IGNORE_SPACE or MYSQL_CLIENT_INTERACTIVE. Read the section about Table 5.3, "MySQL client constants" for further information. In SQL safe mode, this parameter is ignored.

Return Values

Returns a MySQL link identifier on success or FALSE on failure.

Changelog

Version	Description
5.5.0	This function will generate an E_DEPRECATED error.

Examples

Example 5.6 mysql_connect example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully';
mysql_close($link);
?>
```

Example 5.7 mysql_connect example using hostname:port syntax

```
<?php
// we connect to example.com and port 3307
$link = mysql_connect('example.com:3307', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully';
mysql_close($link);

// we connect to localhost at port 3307
$link = mysql_connect('127.0.0.1:3307', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully';
mysql_close($link);
?>
```

Example 5.8 mysql_connect example using ":/path/to/socket" syntax

```
<?php
// we connect to localhost and socket e.g. /tmp/mysql.sock

// variant 1: omit localhost
$link = mysql_connect(':/tmp/mysql', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully';
mysql_close($link);
```

```
// variant 2: with localhost
$link = mysql_connect('localhost:/tmp/mysql.sock', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully';
mysql_close($link);
?>
```

Notes

> **Note**
>
> Whenever you specify "localhost" or "localhost:port" as server, the MySQL client library will override this and try to connect to a local socket (named pipe on Windows). If you want to use TCP/IP, use "127.0.0.1" instead of "localhost". If the MySQL client library tries to connect to the wrong local socket, you should set the correct path as *mysql.default_host* string in your PHP configuration and leave the server field blank.

> **Note**
>
> The link to the server will be closed as soon as the execution of the script ends, unless it's closed earlier by explicitly calling mysql_close.

> **Note**
>
> You can suppress the error message on failure by prepending a @ to the function name.

> **Note**
>
> Error "Can't create TCP/IP socket (10106)" usually means that the variables_order configure directive doesn't contain character E. On Windows, if the environment is not copied the SYSTEMROOT environment variable won't be available and PHP will have problems loading Winsock.

See Also

mysql_pconnect
mysql_close

5.5.5 mysql_create_db

- mysql_create_db

  Create a MySQL database

> **Warning**
>
> This function was deprecated in PHP 4.3.0, and it and the entire original MySQL extension was removed in PHP 7.0.0. Instead, use either the actively developed MySQLi or PDO_MySQL extensions. See also the MySQL: choosing an API guide and its related FAQ entry for additional information. Alternatives to this function include:

> mysqli_query
> PDO::query

Description

```
bool mysql_create_db(
  string database_name,
  resource link_identifier
      =  =NULL);
```

mysql_create_db attempts to create a new database on the server associated with the specified link identifier.

Parameters

database_name	The name of the database being created.
link_identifier	The MySQL connection. If the link identifier is not specified, the last link opened by mysql_connect is assumed. If no such link is found, it will try to create one as if mysql_connect had been called with no arguments. If no connection is found or established, an E_WARNING level error is generated.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 5.9 mysql_create_db alternative example

The function mysql_create_db is deprecated. It is preferable to use mysql_query to issue an sql CREATE DATABASE statement instead.

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}

$sql = 'CREATE DATABASE my_db';
if (mysql_query($sql, $link)) {
    echo "Database my_db created successfully\n";
} else {
    echo 'Error creating database: ' . mysql_error() . "\n";
}
?>
```

The above example will output something similar to:

```
Database my_db created successfully
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used: `mysql_createdb`

> **Note**
>
> This function will not be available if the MySQL extension was built against a MySQL 4.x client library.

See Also

`mysql_query`
`mysql_select_db`

5.5.6 `mysql_data_seek`

- `mysql_data_seek`

  Move internal result pointer

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_data_seek`
> `PDO::FETCH_ORI_ABS`

Description

```
bool mysql_data_seek(
  resource result,
  int row_number);
```

`mysql_data_seek` moves the internal row pointer of the MySQL result associated with the specified result identifier to point to the specified row number. The next call to a MySQL fetch function, such as `mysql_fetch_assoc`, would return that row.

`row_number` starts at 0. The *`row_number`* should be a value in the range from 0 to `mysql_num_rows` - 1. However if the result set is empty (`mysql_num_rows` == 0), a seek to 0 will fail with a E_WARNING and `mysql_data_seek` will return `FALSE`.

Parameters

`result`	The result resource that is being evaluated. This result comes from a call to `mysql_query`.
`row_number`	The desired row number of the new result pointer.

Return Values

Returns `TRUE` on success or `FALSE` on failure.

Examples

Example 5.10 `mysql_data_seek` example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
$db_selected = mysql_select_db('sample_db');
if (!$db_selected) {
    die('Could not select database: ' . mysql_error());
}
$query = 'SELECT last_name, first_name FROM friends';
$result = mysql_query($query);
if (!$result) {
    die('Query failed: ' . mysql_error());
}
/* fetch rows in reverse order */
for ($i = mysql_num_rows($result) - 1; $i >= 0; $i--) {
    if (!mysql_data_seek($result, $i)) {
        echo "Cannot seek to row $i: " . mysql_error() . "\n";
        continue;
    }

    if (!($row = mysql_fetch_assoc($result))) {
        continue;
    }

    echo $row['last_name'] . ' ' . $row['first_name'] . "<br />\n";
}

mysql_free_result($result);
?>
```

Notes

> **Note**
>
> The function `mysql_data_seek` can be used in conjunction only with `mysql_query`, not with `mysql_unbuffered_query`.

See Also

`mysql_query`
`mysql_num_rows`
`mysql_fetch_row`
`mysql_fetch_assoc`
`mysql_fetch_array`
`mysql_fetch_object`

5.5.7 `mysql_db_name`

- `mysql_db_name`

  Retrieves database name from the call to `mysql_list_dbs`

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> Query: `SELECT DATABASE()`

Description

```
string mysql_db_name(
  resource result,
  int row,
  mixed field
      =  =NULL);
```

Retrieve the database name from a call to `mysql_list_dbs`.

Parameters

`result`	The result pointer from a call to `mysql_list_dbs`.
`row`	The index into the result set.
`field`	The field name.

Return Values

Returns the database name on success, and `FALSE` on failure. If `FALSE` is returned, use `mysql_error` to determine the nature of the error.

Changelog

Version	Description
5.5.0	The `mysql_list_dbs` function is deprecated, and emits an `E_DEPRECATED` level error.

Examples

Example 5.11 `mysql_db_name` example

```
<?php
error_reporting(E_ALL);

$link = mysql_connect('dbhost', 'username', 'password');
$db_list = mysql_list_dbs($link);

$i = 0;
$cnt = mysql_num_rows($db_list);
while ($i < $cnt) {
    echo mysql_db_name($db_list, $i) . "\n";
    $i++;
}
?>
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used: `mysql_dbname`

See Also

`mysql_list_dbs`
`mysql_tablename`

5.5.8 `mysql_db_query`

- `mysql_db_query`

  Selects a database and executes a query on it

> **Warning**
>
> This function was deprecated in PHP 5.3.0, and it and the entire original MySQL extension was removed in PHP 7.0.0. Instead, use either the actively developed MySQLi or PDO_MySQL extensions. See also the MySQL: choosing an API guide and its related FAQ entry for additional information. Alternatives to this function include:
>
> `mysqli_select_db` then the query
> `PDO::__construct`

Description

```
resource mysql_db_query(
  string database,
  string query,
  resource link_identifier
      =  =NULL);
```

`mysql_db_query` selects a database, and executes a query on it.

Parameters

`database`	The name of the database that will be selected.
`query`	The MySQL query. Data inside the query should be properly escaped.
`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

Returns a positive MySQL result resource to the query result, or `FALSE` on error. The function also returns `TRUE`/`FALSE` for `INSERT`/`UPDATE`/`DELETE` queries to indicate success/failure.

Changelog

Version	Description
5.3.0	This function now throws an E_DEPRECATED notice.

Examples

Example 5.12 `mysql_db_query` alternative example

```
<?php

if (!$link = mysql_connect('mysql_host', 'mysql_user', 'mysql_password')) {
    echo 'Could not connect to mysql';
    exit;
}

if (!mysql_select_db('mysql_dbname', $link)) {
    echo 'Could not select database';
    exit;
}

$sql    = 'SELECT foo FROM bar WHERE id = 42';
$result = mysql_query($sql, $link);

if (!$result) {
    echo "DB Error, could not query the database\n";
    echo 'MySQL Error: ' . mysql_error();
    exit;
}

while ($row = mysql_fetch_assoc($result)) {
    echo $row['foo'];
}

mysql_free_result($result);

?>
```

Notes

> **Note**
>
> Be aware that this function does *NOT* switch back to the database you were connected before. In other words, you can't use this function to *temporarily* run a sql query on another database, you would have to manually switch back. Users are strongly encouraged to use the `database.table` syntax in their sql queries or `mysql_select_db` instead of this function.

See Also

`mysql_query`
`mysql_select_db`

5.5.9 `mysql_drop_db`

- `mysql_drop_db`

Drop (delete) a MySQL database

> **Warning**
>
> This function was deprecated in PHP 4.3.0, and it and the entire original MySQL extension was removed in PHP 7.0.0. Instead, use either the actively developed MySQLi or PDO_MySQL extensions. See also the MySQL: choosing an API guide and its related FAQ entry for additional information. Alternatives to this function include:
>
> Execute a `DROP DATABASE` query

Description

```
bool mysql_drop_db(
  string database_name,
  resource link_identifier
      =  =NULL);
```

`mysql_drop_db` attempts to drop (remove) an entire database from the server associated with the specified link identifier. This function is deprecated, it is preferable to use `mysql_query` to issue an sql `DROP DATABASE` statement instead.

Parameters

`database_name`	The name of the database that will be deleted.
`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

Returns `TRUE` on success or `FALSE` on failure.

Examples

Example 5.13 `mysql_drop_db` alternative example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}

$sql = 'DROP DATABASE my_db';
if (mysql_query($sql, $link)) {
    echo "Database my_db was successfully dropped\n";
} else {
    echo 'Error dropping database: ' . mysql_error() . "\n";
}
?>
```

Notes

> **Warning**
>
> This function will not be available if the MySQL extension was built against a MySQL 4.x client library.

> **Note**
>
> For backward compatibility, the following deprecated alias may be used: `mysql_dropdb`

See Also

`mysql_query`

5.5.10 `mysql_errno`

- `mysql_errno`

  Returns the numerical value of the error message from previous MySQL operation

  > **Warning**
  >
  > This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
  >
  > `mysqli_errno`
  > `PDO::errorCode`

Description

```
int mysql_errno(
  resource link_identifier
      =  =NULL);
```

Returns the error number from the last MySQL function.

Errors coming back from the MySQL database backend no longer issue warnings. Instead, use `mysql_errno` to retrieve the error code. Note that this function only returns the error code from the most recently executed MySQL function (not including `mysql_error` and `mysql_errno`), so if you want to use it, make sure you check the value before calling another MySQL function.

Parameters

`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

Returns the error number from the last MySQL function, or `0` (zero) if no error occurred.

Examples

Example 5.14 `mysql_errno` example

```
<?php
$link = mysql_connect("localhost", "mysql_user", "mysql_password");

if (!mysql_select_db("nonexistentdb", $link)) {
    echo mysql_errno($link) . ": " . mysql_error($link). "\n";
}

mysql_select_db("kossu", $link);
if (!mysql_query("SELECT * FROM nonexistenttable", $link)) {
    echo mysql_errno($link) . ": " . mysql_error($link) . "\n";
}
?>
```

The above example will output something similar to:

```
1049: Unknown database 'nonexistentdb'
1146: Table 'kossu.nonexistenttable' doesn't exist
```

See Also

`mysql_error`
MySQL error codes

5.5.11 `mysql_error`

- `mysql_error`

  Returns the text of the error message from previous MySQL operation

  > **Warning**
  >
  > This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
  >
  > `mysqli_error`
  > `PDO::errorInfo`

Description

```
  string mysql_error(
    resource link_identifier
        =  =NULL);
```

Returns the error text from the last MySQL function. Errors coming back from the MySQL database backend no longer issue warnings. Instead, use `mysql_error` to retrieve the error text. Note that this function only returns the error text from the most recently executed MySQL function (not including `mysql_error` and `mysql_errno`), so if you want to use it, make sure you check the value before calling another MySQL function.

Parameters

link_identifier	The MySQL connection. If the link identifier is not specified, the last link opened by mysql_connect is assumed. If no such link is found, it will try to create one as if mysql_connect had been called with no arguments. If no connection is found or established, an E_WARNING level error is generated.

Return Values

Returns the error text from the last MySQL function, or '' (empty string) if no error occurred.

Examples

Example 5.15 mysql_error example

```
<?php
$link = mysql_connect("localhost", "mysql_user", "mysql_password");

mysql_select_db("nonexistentdb", $link);
echo mysql_errno($link) . ": " . mysql_error($link). "\n";

mysql_select_db("kossu", $link);
mysql_query("SELECT * FROM nonexistenttable", $link);
echo mysql_errno($link) . ": " . mysql_error($link) . "\n";
?>
```

The above example will output something similar to:

```
1049: Unknown database 'nonexistentdb'
1146: Table 'kossu.nonexistenttable' doesn't exist
```

See Also

mysql_errno
MySQL error codes

5.5.12 mysql_escape_string

- mysql_escape_string

  Escapes a string for use in a mysql_query

> **Warning**
>
> This function was deprecated in PHP 4.3.0, and it and the entire original MySQL extension was removed in PHP 7.0.0. Instead, use either the actively developed MySQLi or PDO_MySQL extensions. See also the MySQL: choosing an API guide and its related FAQ entry for additional information. Alternatives to this function include:

> mysqli_escape_string
> PDO::quote

Description

```
string mysql_escape_string(
  string unescaped_string);
```

This function will escape the *unescaped_string*, so that it is safe to place it in a mysql_query. This function is deprecated.

This function is identical to mysql_real_escape_string except that mysql_real_escape_string takes a connection handler and escapes the string according to the current character set. mysql_escape_string does not take a connection argument and does not respect the current charset setting.

Parameters

unescaped_string	The string that is to be escaped.

Return Values

Returns the escaped string.

Changelog

Version	Description
5.3.0	This function now throws an E_DEPRECATED notice.
4.3.0	This function became deprecated, do not use this function. Instead, use mysql_real_escape_string.

Examples

Example 5.16 mysql_escape_string example

```
<?php
$item = "Zak's Laptop";
$escaped_item = mysql_escape_string($item);
printf("Escaped string: %s\n", $escaped_item);
?>
```

The above example will output:

```
Escaped string: Zak\'s Laptop
```

Notes

> **Note**
>
> mysql_escape_string does not escape % and _.

See Also

mysql_real_escape_string
addslashes
The magic_quotes_gpc directive.

5.5.13 mysql_fetch_array

- mysql_fetch_array

  Fetch a result row as an associative array, a numeric array, or both

  > **Warning**
  >
  > This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
  >
  > mysqli_fetch_array
  > PDOStatement::fetch

Description

```
array mysql_fetch_array(
  resource result,
  int result_type
      =  =MYSQL_BOTH);
```

Returns an array that corresponds to the fetched row and moves the internal data pointer ahead.

Parameters

result	The result resource that is being evaluated. This result comes from a call to mysql_query.
result_type	The type of array that is to be fetched. It's a constant and can take the following values: MYSQL_ASSOC, MYSQL_NUM, and MYSQL_BOTH.

Return Values

Returns an array of strings that corresponds to the fetched row, or FALSE if there are no more rows. The type of returned array depends on how *result_type* is defined. By using MYSQL_BOTH (default), you'll get an array with both associative and number indices. Using MYSQL_ASSOC, you only get associative indices (as mysql_fetch_assoc works), using MYSQL_NUM, you only get number indices (as mysql_fetch_row works).

If two or more columns of the result have the same field names, the last column will take precedence. To access the other column(s) of the same name, you must use the numeric index of the column or make an alias for the column. For aliased columns, you cannot access the contents with the original column name.

Examples

Example 5.17 Query with aliased duplicate field names

```
SELECT table1.field AS foo, table2.field AS bar FROM table1, table2
```

Example 5.18 `mysql_fetch_array` with `MYSQL_NUM`

```
<?php
mysql_connect("localhost", "mysql_user", "mysql_password") or
    die("Could not connect: " . mysql_error());
mysql_select_db("mydb");

$result = mysql_query("SELECT id, name FROM mytable");

while ($row = mysql_fetch_array($result, MYSQL_NUM)) {
    printf("ID: %s  Name: %s", $row[0], $row[1]);
}

mysql_free_result($result);
?>
```

Example 5.19 `mysql_fetch_array` with `MYSQL_ASSOC`

```
<?php
mysql_connect("localhost", "mysql_user", "mysql_password") or
    die("Could not connect: " . mysql_error());
mysql_select_db("mydb");

$result = mysql_query("SELECT id, name FROM mytable");

while ($row = mysql_fetch_array($result, MYSQL_ASSOC)) {
    printf("ID: %s  Name: %s", $row["id"], $row["name"]);
}

mysql_free_result($result);
?>
```

Example 5.20 `mysql_fetch_array` with `MYSQL_BOTH`

```
<?php
mysql_connect("localhost", "mysql_user", "mysql_password") or
    die("Could not connect: " . mysql_error());
mysql_select_db("mydb");

$result = mysql_query("SELECT id, name FROM mytable");

while ($row = mysql_fetch_array($result, MYSQL_BOTH)) {
    printf ("ID: %s  Name: %s", $row[0], $row["name"]);
}

mysql_free_result($result);
?>
```

Notes

> **Performance**
>
> An important thing to note is that using `mysql_fetch_array` is *not significantly* slower than using `mysql_fetch_row`, while it provides a significant added value.

> **Note**
>
> Field names returned by this function are *case-sensitive*.

> **Note**
>
> This function sets NULL fields to the PHP `NULL` value.

See Also

`mysql_fetch_row`
`mysql_fetch_assoc`
`mysql_data_seek`
`mysql_query`

5.5.14 `mysql_fetch_assoc`

- `mysql_fetch_assoc`

  Fetch a result row as an associative array

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_fetch_assoc`
> `PDOStatement::fetch(PDO::FETCH_ASSOC)`

Description

```
array mysql_fetch_assoc(
  resource result);
```

Returns an associative array that corresponds to the fetched row and moves the internal data pointer ahead. `mysql_fetch_assoc` is equivalent to calling `mysql_fetch_array` with MYSQL_ASSOC for the optional second parameter. It only returns an associative array.

Parameters

`result`	The result resource that is being evaluated. This result comes from a call to `mysql_query`.

Return Values

Returns an associative array of strings that corresponds to the fetched row, or `FALSE` if there are no more rows.

If two or more columns of the result have the same field names, the last column will take precedence. To access the other column(s) of the same name, you either need to access the result with numeric indices by

using mysql_fetch_row or add alias names. See the example at the mysql_fetch_array description about aliases.

Examples

Example 5.21 An expanded mysql_fetch_assoc example

```
<?php

$conn = mysql_connect("localhost", "mysql_user", "mysql_password");

if (!$conn) {
    echo "Unable to connect to DB: " . mysql_error();
    exit;
}

if (!mysql_select_db("mydbname")) {
    echo "Unable to select mydbname: " . mysql_error();
    exit;
}

$sql = "SELECT id as userid, fullname, userstatus
        FROM   sometable
        WHERE  userstatus = 1";

$result = mysql_query($sql);

if (!$result) {
    echo "Could not successfully run query ($sql) from DB: " . mysql_error();
    exit;
}

if (mysql_num_rows($result) == 0) {
    echo "No rows found, nothing to print so am exiting";
    exit;
}

// While a row of data exists, put that row in $row as an associative array
// Note: If you're expecting just one row, no need to use a loop
// Note: If you put extract($row); inside the following loop, you'll
//       then create $userid, $fullname, and $userstatus
while ($row = mysql_fetch_assoc($result)) {
    echo $row["userid"];
    echo $row["fullname"];
    echo $row["userstatus"];
}

mysql_free_result($result);

?>
```

Notes

> **Performance**
>
> An important thing to note is that using mysql_fetch_assoc is *not significantly* slower than using mysql_fetch_row, while it provides a significant added value.

> **Note**
>
> Field names returned by this function are *case-sensitive*.

> **Note**
>
> This function sets NULL fields to the PHP `NULL` value.

See Also

`mysql_fetch_row`
`mysql_fetch_array`
`mysql_data_seek`
`mysql_query`
`mysql_error`

5.5.15 `mysql_fetch_field`

- `mysql_fetch_field`

  Get column information from a result and return as an object

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_fetch_field`
> `PDOStatement::getColumnMeta`

Description

```
object mysql_fetch_field(
  resource result,
  int field_offset
      =  =0);
```

Returns an object containing field information. This function can be used to obtain information about fields in the provided query result.

Parameters

`result`	The result resource that is being evaluated. This result comes from a call to `mysql_query`.
`field_offset`	The numerical field offset. If the field offset is not specified, the next field that was not yet retrieved by this function is retrieved. The *`field_offset`* starts at `0`.

Return Values

Returns an object containing field information. The properties of the object are:

- name - column name
- table - name of the table the column belongs to, which is the alias name if one is defined
- max_length - maximum length of the column

- not_null - 1 if the column cannot be NULL
- primary_key - 1 if the column is a primary key
- unique_key - 1 if the column is a unique key
- multiple_key - 1 if the column is a non-unique key
- numeric - 1 if the column is numeric
- blob - 1 if the column is a BLOB
- type - the type of the column
- unsigned - 1 if the column is unsigned
- zerofill - 1 if the column is zero-filled

Examples

Example 5.22 `mysql_fetch_field` example

```
<?php
$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$conn) {
    die('Could not connect: ' . mysql_error());
}
mysql_select_db('database');
$result = mysql_query('select * from table');
if (!$result) {
    die('Query failed: ' . mysql_error());
}
/* get column metadata */
$i = 0;
while ($i < mysql_num_fields($result)) {
    echo "Information for column $i:<br />\n";
    $meta = mysql_fetch_field($result, $i);
    if (!$meta) {
        echo "No information available<br />\n";
    }
    echo "<pre>
blob:         $meta->blob
max_length:   $meta->max_length
multiple_key: $meta->multiple_key
name:         $meta->name
not_null:     $meta->not_null
numeric:      $meta->numeric
primary_key:  $meta->primary_key
table:        $meta->table
type:         $meta->type
unique_key:   $meta->unique_key
unsigned:     $meta->unsigned
zerofill:     $meta->zerofill
</pre>";
    $i++;
}
mysql_free_result($result);
?>
```

Notes

> **Note**
>
> Field names returned by this function are *case-sensitive*.

> **Note**
>
> If field or tablenames are aliased in the SQL query the aliased name will be returned. The original name can be retrieved for instance by using `mysqli_result::fetch_field`.

See Also

`mysql_field_seek`

5.5.16 `mysql_fetch_lengths`

- `mysql_fetch_lengths`

  Get the length of each output in a result

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_fetch_lengths`
> `PDOStatement::getColumnMeta`

Description

```
array mysql_fetch_lengths(
  resource result);
```

Returns an array that corresponds to the lengths of each field in the last row fetched by MySQL.

`mysql_fetch_lengths` stores the lengths of each result column in the last row returned by `mysql_fetch_row`, `mysql_fetch_assoc`, `mysql_fetch_array`, and `mysql_fetch_object` in an array, starting at offset 0.

Parameters

`result`	The result resource that is being evaluated. This result comes from a call to `mysql_query`.

Return Values

An array of lengths on success or `FALSE` on failure.

Examples

Example 5.23 A `mysql_fetch_lengths` example

```
```

```
<?php
$result = mysql_query("SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' . mysql_error();
    exit;
}
$row     = mysql_fetch_assoc($result);
$lengths = mysql_fetch_lengths($result);

print_r($row);
print_r($lengths);
?>
```

The above example will output something similar to:

```
Array
(
    [id] => 42
    [email] => user@example.com
)
Array
(
    [0] => 2
    [1] => 16
)
```

See Also

mysql_field_len
mysql_fetch_row
strlen

5.5.17 mysql_fetch_object

- mysql_fetch_object

  Fetch a result row as an object

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> mysqli_fetch_object
> PDOStatement::fetch(PDO::FETCH_OBJ)

Description

```
object mysql_fetch_object(
  resource result,
  string class_name,
  array params);
```

Returns an object with properties that correspond to the fetched row and moves the internal data pointer ahead.

Parameters

result	The result resource that is being evaluated. This result comes from a call to mysql_query.
class_name	The name of the class to instantiate, set the properties of and return. If not specified, a stdClass object is returned.
params	An optional array of parameters to pass to the constructor for *class_name* objects.

Return Values

Returns an object with string properties that correspond to the fetched row, or FALSE if there are no more rows.

Examples

Example 5.24 mysql_fetch_object example

```
<?php
mysql_connect("hostname", "user", "password");
mysql_select_db("mydb");
$result = mysql_query("select * from mytable");
while ($row = mysql_fetch_object($result)) {
    echo $row->user_id;
    echo $row->fullname;
}
mysql_free_result($result);
?>
```

Example 5.25 mysql_fetch_object example

```
<?php
class foo {
    public $name;
}

mysql_connect("hostname", "user", "password");
mysql_select_db("mydb");

$result = mysql_query("select name from mytable limit 1");
$obj = mysql_fetch_object($result, 'foo');
var_dump($obj);
?>
```

Notes

> **Performance**
>
> Speed-wise, the function is identical to mysql_fetch_array, and almost as quick as mysql_fetch_row (the difference is insignificant).

> **Note**
>
> `mysql_fetch_object` is similar to `mysql_fetch_array`, with one difference - an object is returned, instead of an array. Indirectly, that means that you can only access the data by the field names, and not by their offsets (numbers are illegal property names).

> **Note**
>
> Field names returned by this function are *case-sensitive*.

> **Note**
>
> This function sets NULL fields to the PHP `NULL` value.

See Also

`mysql_fetch_array`
`mysql_fetch_assoc`
`mysql_fetch_row`
`mysql_data_seek`
`mysql_query`

5.5.18 `mysql_fetch_row`

- `mysql_fetch_row`

  Get a result row as an enumerated array

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_fetch_row`
> `PDOStatement::fetch(PDO::FETCH_NUM)`

Description

```
array mysql_fetch_row(
  resource result);
```

Returns a numerical array that corresponds to the fetched row and moves the internal data pointer ahead.

Parameters

`result`	The result resource that is being evaluated. This result comes from a call to `mysql_query`.

Return Values

Returns an numerical array of strings that corresponds to the fetched row, or `FALSE` if there are no more rows.

mysql_fetch_row fetches one row of data from the result associated with the specified result identifier. The row is returned as an array. Each result column is stored in an array offset, starting at offset 0.

Examples

Example 5.26 Fetching one row with mysql_fetch_row

```
<?php
$result = mysql_query("SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' . mysql_error();
    exit;
}
$row = mysql_fetch_row($result);

echo $row[0]; // 42
echo $row[1]; // the email value
?>
```

Notes

> **Note**
>
> This function sets NULL fields to the PHP NULL value.

See Also

mysql_fetch_array
mysql_fetch_assoc
mysql_fetch_object
mysql_data_seek
mysql_fetch_lengths
mysql_result

5.5.19 mysql_field_flags

- mysql_field_flags

  Get the flags associated with the specified field in a result

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> mysqli_fetch_field_direct [flags]
> PDOStatement::getColumnMeta [flags]

Description

```
string mysql_field_flags(
  resource result,
  int field_offset);
```

`mysql_field_flags` returns the field flags of the specified field. The flags are reported as a single word per flag separated by a single space, so that you can split the returned value using `explode`.

Parameters

`result`	The result resource that is being evaluated. This result comes from a call to `mysql_query`.
`field_offset`	The numerical field offset. The *`field_offset`* starts at `0`. If *`field_offset`* does not exist, an error of level `E_WARNING` is also issued.

Return Values

Returns a string of flags associated with the result or `FALSE` on failure.

The following flags are reported, if your version of MySQL is current enough to support them: `"not_null"`, `"primary_key"`, `"unique_key"`, `"multiple_key"`, `"blob"`, `"unsigned"`, `"zerofill"`, `"binary"`, `"enum"`, `"auto_increment"` and `"timestamp"`.

Examples

Example 5.27 A `mysql_field_flags` example

```
<?php
$result = mysql_query("SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' . mysql_error();
    exit;
}
$flags = mysql_field_flags($result, 0);

echo $flags;
print_r(explode(' ', $flags));
?>
```

The above example will output something similar to:

```
not_null primary_key auto_increment
Array
(
    [0] => not_null
    [1] => primary_key
    [2] => auto_increment
)
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used: `mysql_fieldflags`

See Also

mysql_field_type
mysql_field_len

5.5.20 mysql_field_len

- mysql_field_len

  Returns the length of the specified field

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> mysqli_fetch_field_direct [length]
> PDOStatement::getColumnMeta [len]

Description

```
int mysql_field_len(
  resource result,
  int field_offset);
```

mysql_field_len returns the length of the specified field.

Parameters

result	The result resource that is being evaluated. This result comes from a call to mysql_query.
field_offset	The numerical field offset. The *field_offset* starts at 0. If *field_offset* does not exist, an error of level E_WARNING is also issued.

Return Values

The length of the specified field index on success or FALSE on failure.

Examples

Example 5.28 mysql_field_len example

```
<?php
$result = mysql_query("SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' . mysql_error();
    exit;
}

// Will get the length of the id field as specified in the database
// schema.
$length = mysql_field_len($result, 0);
echo $length;
?>
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used: `mysql_fieldlen`

See Also

`mysql_fetch_lengths`
`strlen`

5.5.21 `mysql_field_name`

- `mysql_field_name`

  Get the name of the specified field in a result

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_fetch_field_direct` [name] or [orgname]
> `PDOStatement::getColumnMeta` [name]

Description

```
string mysql_field_name(
  resource result,
  int field_offset);
```

`mysql_field_name` returns the name of the specified field index.

Parameters

`result`	The result resource that is being evaluated. This result comes from a call to `mysql_query`.
`field_offset`	The numerical field offset. The *`field_offset`* starts at `0`. If *`field_offset`* does not exist, an error of level `E_WARNING` is also issued.

Return Values

The name of the specified field index on success or `FALSE` on failure.

Examples

Example 5.29 `mysql_field_name` example

```
<?php
/* The users table consists of three fields:
 *   user_id
 *   username
 *   password.
 */
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect to MySQL server: ' . mysql_error());
}
$dbname = 'mydb';
$db_selected = mysql_select_db($dbname, $link);
if (!$db_selected) {
    die("Could not set $dbname: " . mysql_error());
}
$res = mysql_query('select * from users', $link);

echo mysql_field_name($res, 0) . "\n";
echo mysql_field_name($res, 2);
?>
```

The above example will output:

```
user_id
password
```

Notes

> **Note**
>
> Field names returned by this function are *case-sensitive*.

> **Note**
>
> For backward compatibility, the following deprecated alias may be used: `mysql_fieldname`

See Also

`mysql_field_type`
`mysql_field_len`

5.5.22 `mysql_field_seek`

- `mysql_field_seek`

  Set result pointer to a specified field offset

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:

> `mysqli_field_seek`
> `PDOStatement::fetch` using the *`cursor_orientation`* and *`offset`* parameters

Description

```
bool mysql_field_seek(
  resource result,
  int field_offset);
```

Seeks to the specified field offset. If the next call to `mysql_fetch_field` doesn't include a field offset, the field offset specified in `mysql_field_seek` will be returned.

Parameters

`result`	The result resource that is being evaluated. This result comes from a call to `mysql_query`.
`field_offset`	The numerical field offset. The *`field_offset`* starts at `0`. If *`field_offset`* does not exist, an error of level `E_WARNING` is also issued.

Return Values

Returns `TRUE` on success or `FALSE` on failure.

See Also

`mysql_fetch_field`

5.5.23 `mysql_field_table`

- `mysql_field_table`

  Get name of the table the specified field is in

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_fetch_field_direct` [table] or [orgtable]
> `PDOStatement::getColumnMeta` [table]

Description

```
string mysql_field_table(
  resource result,
  int field_offset);
```

Returns the name of the table that the specified field is in.

Parameters

result	The result resource that is being evaluated. This result comes from a call to mysql_query.
field_offset	The numerical field offset. The *field_offset* starts at 0. If *field_offset* does not exist, an error of level E_WARNING is also issued.

Return Values

The name of the table on success.

Examples

Example 5.30 A mysql_field_table example

```
<?php

$query = "SELECT account.*, country.* FROM account, country WHERE country.name = 'Portugal' AND account.cou

// get the result from the DB
$result = mysql_query($query);

// Lists the table name and then the field name
for ($i = 0; $i < mysql_num_fields($result); ++$i) {
    $table = mysql_field_table($result, $i);
    $field = mysql_field_name($result, $i);

    echo  "$table: $field\n";
}

?>
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used: mysql_fieldtable

See Also

mysql_list_tables

5.5.24 mysql_field_type

- mysql_field_type

  Get the type of the specified field in a result

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:

> `mysqli_fetch_field_direct` [type]
> `PDOStatement::getColumnMeta` [driver:decl_type] or [pdo_type]

Description

```
  string mysql_field_type(
    resource result,
    int field_offset);
```

`mysql_field_type` is similar to the `mysql_field_name` function. The arguments are identical, but the field type is returned instead.

Parameters

`result`	The result resource that is being evaluated. This result comes from a call to `mysql_query`.
`field_offset`	The numerical field offset. The *`field_offset`* starts at `0`. If *`field_offset`* does not exist, an error of level `E_WARNING` is also issued.

Return Values

The returned field type will be one of `"int"`, `"real"`, `"string"`, `"blob"`, and others as detailed in the MySQL documentation.

Examples

Example 5.31 `mysql_field_type` example

```
<?php
mysql_connect("localhost", "mysql_username", "mysql_password");
mysql_select_db("mysql");
$result = mysql_query("SELECT * FROM func");
$fields = mysql_num_fields($result);
$rows   = mysql_num_rows($result);
$table  = mysql_field_table($result, 0);
echo "Your '" . $table . "' table has " . $fields . " fields and " . $rows . " record(s)\n";
echo "The table has the following fields:\n";
for ($i=0; $i < $fields; $i++) {
    $type  = mysql_field_type($result, $i);
    $name  = mysql_field_name($result, $i);
    $len   = mysql_field_len($result, $i);
    $flags = mysql_field_flags($result, $i);
    echo $type . " " . $name . " " . $len . " " . $flags . "\n";
}
mysql_free_result($result);
mysql_close();
?>
```

The above example will output something similar to:

```
Your 'func' table has 4 fields and 1 record(s)
The table has the following fields:
string name 64 not_null primary_key binary
int ret 1 not_null
```

```
string dl 128 not_null
string type 9 not_null enum
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used:
> `mysql_fieldtype`

See Also

`mysql_field_name`
`mysql_field_len`

5.5.25 `mysql_free_result`

- `mysql_free_result`

  Free result memory

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_free_result`
> Assign the value of `NULL` to the PDO object, or `PDOStatement::closeCursor`

Description

```
bool mysql_free_result(
  resource result);
```

`mysql_free_result` will free all memory associated with the result identifier *`result`*.

`mysql_free_result` only needs to be called if you are concerned about how much memory is being used for queries that return large result sets. All associated result memory is automatically freed at the end of the script's execution.

Parameters

`result`	The result resource that is being evaluated. This result comes from a call to `mysql_query`.

Return Values

Returns `TRUE` on success or `FALSE` on failure.

If a non-resource is used for the *`result`*, an error of level E_WARNING will be emitted. It's worth noting that `mysql_query` only returns a resource for SELECT, SHOW, EXPLAIN, and DESCRIBE queries.

Examples

Example 5.32 A `mysql_free_result` example

```
<?php
$result = mysql_query("SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' . mysql_error();
    exit;
}
/* Use the result, assuming we're done with it afterwards */
$row = mysql_fetch_assoc($result);

/* Now we free up the result and continue on with our script */
mysql_free_result($result);

echo $row['id'];
echo $row['email'];
?>
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used:
> `mysql_freeresult`

See Also

`mysql_query`
`is_resource`

5.5.26 `mysql_get_client_info`

- `mysql_get_client_info`

  Get MySQL client info

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_get_client_info`
> `PDO::getAttribute(PDO::ATTR_CLIENT_VERSION)`

Description

```
  string mysql_get_client_info();
```

`mysql_get_client_info` returns a string that represents the client library version.

Return Values

The MySQL client version.

Examples

Example 5.33 `mysql_get_client_info` example

```
<?php
printf("MySQL client info: %s\n", mysql_get_client_info());
?>
```

The above example will output something similar to:

```
MySQL client info: 3.23.39
```

See Also

`mysql_get_host_info`
`mysql_get_proto_info`
`mysql_get_server_info`

5.5.27 `mysql_get_host_info`

- `mysql_get_host_info`

  Get MySQL host info

  > **Warning**
  >
  > This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
  >
  > `mysqli_get_host_info`
  > `PDO::getAttribute(PDO::ATTR_CONNECTION_STATUS)`

Description

```
  string mysql_get_host_info(
    resource link_identifier
        =  =NULL);
```

Describes the type of connection in use for the connection, including the server host name.

Parameters

`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

Returns a string describing the type of MySQL connection in use for the connection or FALSE on failure.

Examples

Example 5.34 `mysql_get_host_info` example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
printf("MySQL host info: %s\n", mysql_get_host_info());
?>
```

The above example will output something similar to:

```
MySQL host info: Localhost via UNIX socket
```

See Also

mysql_get_client_info
mysql_get_proto_info
mysql_get_server_info

5.5.28 mysql_get_proto_info

- mysql_get_proto_info

  Get MySQL protocol info

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> mysqli_get_proto_info

Description

```
int mysql_get_proto_info(
  resource link_identifier
      =  =NULL);
```

Retrieves the MySQL protocol.

Parameters

link_identifier	The MySQL connection. If the link identifier is not specified, the last link opened by mysql_connect is assumed. If no such link is found, it will try to create one as if mysql_connect had been called with no arguments. If no connection is found or established, an E_WARNING level error is generated.

Return Values

Returns the MySQL protocol on success or FALSE on failure.

Examples

Example 5.35 mysql_get_proto_info example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
printf("MySQL protocol version: %s\n", mysql_get_proto_info());
?>
```

The above example will output something similar to:

```
MySQL protocol version: 10
```

See Also

mysql_get_client_info
mysql_get_host_info
mysql_get_server_info

5.5.29 mysql_get_server_info

- mysql_get_server_info

  Get MySQL server info

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> mysqli_get_server_info
> PDO::getAttribute(PDO::ATTR_SERVER_VERSION)

Description

```
  string mysql_get_server_info(
    resource link_identifier
        =  =NULL);
```

Retrieves the MySQL server version.

Parameters

link_identifier	The MySQL connection. If the link identifier is not specified, the last link opened by mysql_connect is assumed. If no such link is found, it will try to create one as if mysql_connect had been called with no arguments. If no connection is found or established, an E_WARNING level error is generated.

Return Values

Returns the MySQL server version on success or FALSE on failure.

Examples

Example 5.36 mysql_get_server_info example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
printf("MySQL server version: %s\n", mysql_get_server_info());
?>
```

The above example will output something similar to:

```
MySQL server version: 4.0.1-alpha
```

See Also

mysql_get_client_info
mysql_get_host_info
mysql_get_proto_info
phpversion

5.5.30 mysql_info

- mysql_info

  Get information about the most recent query

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL:

> choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> mysqli_info

Description

```
string mysql_info(
  resource link_identifier
      =  =NULL);
```

Returns detailed information about the last query.

Parameters

link_identifier	The MySQL connection. If the link identifier is not specified, the last link opened by mysql_connect is assumed. If no such link is found, it will try to create one as if mysql_connect had been called with no arguments. If no connection is found or established, an E_WARNING level error is generated.

Return Values

Returns information about the statement on success, or FALSE on failure. See the example below for which statements provide information, and what the returned value may look like. Statements that are not listed will return FALSE.

Examples

Example 5.37 Relevant MySQL Statements

Statements that return string values. The numbers are only for illustrating purpose; their values will correspond to the query.

```
INSERT INTO ... SELECT ...
String format: Records: 23 Duplicates: 0 Warnings: 0
INSERT INTO ... VALUES (...),(...),(...)...
String format: Records: 37 Duplicates: 0 Warnings: 0
LOAD DATA INFILE ...
String format: Records: 42 Deleted: 0 Skipped: 0 Warnings: 0
ALTER TABLE
String format: Records: 60 Duplicates: 0 Warnings: 0
UPDATE
String format: Rows matched: 65 Changed: 65 Warnings: 0
```

Notes

> **Note**
>
> mysql_info returns a non-FALSE value for the INSERT ... VALUES statement only if multiple value lists are specified in the statement.

See Also

mysql_affected_rows
mysql_insert_id

mysql_stat

5.5.31 mysql_insert_id

- mysql_insert_id

  Get the ID generated in the last query

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> mysqli_insert_id
> PDO::lastInsertId

Description

```
int mysql_insert_id(
  resource link_identifier
      =  =NULL);
```

Retrieves the ID generated for an AUTO_INCREMENT column by the previous query (usually INSERT).

Parameters

link_identifier	The MySQL connection. If the link identifier is not specified, the last link opened by mysql_connect is assumed. If no such link is found, it will try to create one as if mysql_connect had been called with no arguments. If no connection is found or established, an E_WARNING level error is generated.

Return Values

The ID generated for an AUTO_INCREMENT column by the previous query on success, 0 if the previous query does not generate an AUTO_INCREMENT value, or FALSE if no MySQL connection was established.

Examples

Example 5.38 mysql_insert_id example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
mysql_select_db('mydb');

mysql_query("INSERT INTO mytable (product) values ('kossu')");
printf("Last inserted record has id %d\n", mysql_insert_id());
?>
```

Notes

> **Caution**
>
> `mysql_insert_id` will convert the return type of the native MySQL C API function `mysql_insert_id()` to a type of `long` (named int in PHP). If your AUTO_INCREMENT column has a column type of BIGINT (64 bits) the conversion may result in an incorrect value. Instead, use the internal MySQL SQL function LAST_INSERT_ID() in an SQL query. For more information about PHP's maximum integer values, please see the integer documentation.

> **Note**
>
> Because `mysql_insert_id` acts on the last performed query, be sure to call `mysql_insert_id` immediately after the query that generates the value.

> **Note**
>
> The value of the MySQL SQL function `LAST_INSERT_ID()` always contains the most recently generated AUTO_INCREMENT value, and is not reset between queries.

See Also

`mysql_query`
`mysql_info`

5.5.32 `mysql_list_dbs`

- `mysql_list_dbs`

  List databases available on a MySQL server

> **Warning**
>
> This function was deprecated in PHP 5.4.0, and it and the entire original MySQL extension was removed in PHP 7.0.0. Instead, use either the actively developed MySQLi or PDO_MySQL extensions. See also the MySQL: choosing an API guide and its related FAQ entry for additional information. Alternatives to this function include:
>
> SQL Query: `SHOW DATABASES`

Description

```
resource mysql_list_dbs(
  resource link_identifier
      =  =NULL);
```

Returns a result pointer containing the databases available from the current mysql daemon.

Parameters

`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found,

it will try to create one as if mysql_connect had been called with no arguments. If no connection is found or established, an E_WARNING level error is generated.

Return Values

Returns a result pointer resource on success, or FALSE on failure. Use the mysql_tablename function to traverse this result pointer, or any function for result tables, such as mysql_fetch_array.

Examples

Example 5.39 mysql_list_dbs example

```
<?php
// Usage without mysql_list_dbs()
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
$res = mysql_query("SHOW DATABASES");

while ($row = mysql_fetch_assoc($res)) {
    echo $row['Database'] . "\n";
}

// Deprecated as of PHP 5.4.0
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
$db_list = mysql_list_dbs($link);

while ($row = mysql_fetch_object($db_list)) {
     echo $row->Database . "\n";
}
?>
```

The above example will output something similar to:

```
database1
database2
database3
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used: mysql_listdbs

See Also

mysql_db_name
mysql_select_db

5.5.33 mysql_list_fields

- mysql_list_fields

List MySQL table fields

> **Warning**
>
> This function was deprecated in PHP 5.4.0, and it and the entire original MySQL extension was removed in PHP 7.0.0. Instead, use either the actively developed MySQLi or PDO_MySQL extensions. See also the MySQL: choosing an API guide and its related FAQ entry for additional information. Alternatives to this function include:
>
> SQL Query: `SHOW COLUMNS FROM sometable`

Description

```
resource mysql_list_fields(
  string database_name,
  string table_name,
  resource link_identifier
      =  =NULL);
```

Retrieves information about the given table name.

This function is deprecated. It is preferable to use `mysql_query` to issue an SQL `SHOW COLUMNS FROM table [LIKE 'name']` statement instead.

Parameters

`database_name`	The name of the database that's being queried.
`table_name`	The name of the table that's being queried.
`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

A result pointer resource on success, or `FALSE` on failure.

The returned result can be used with `mysql_field_flags`, `mysql_field_len`, `mysql_field_name` and `mysql_field_type`.

Examples

Example 5.40 Alternate to deprecated `mysql_list_fields`

```
<?php
$result = mysql_query("SHOW COLUMNS FROM sometable");
if (!$result) {
    echo 'Could not run query: ' . mysql_error();
    exit;
}
if (mysql_num_rows($result) > 0) {
    while ($row = mysql_fetch_assoc($result)) {
        print_r($row);
    }
```

```
}
?>
```

The above example will output something similar to:

```
Array
(
    [Field] => id
    [Type] => int(7)
    [Null] =>
    [Key] => PRI
    [Default] =>
    [Extra] => auto_increment
)
Array
(
    [Field] => email
    [Type] => varchar(100)
    [Null] =>
    [Key] =>
    [Default] =>
    [Extra] =>
)
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used:
> `mysql_listfields`

See Also

`mysql_field_flags`
`mysql_info`

5.5.34 `mysql_list_processes`

- `mysql_list_processes`

  List MySQL processes

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_thread_id`

Description

```
  resource mysql_list_processes(
```

```
    resource link_identifier
        =  =NULL);
```

Retrieves the current MySQL server threads.

Parameters

link_identifier	The MySQL connection. If the link identifier is not specified, the last link opened by mysql_connect is assumed. If no such link is found, it will try to create one as if mysql_connect had been called with no arguments. If no connection is found or established, an E_WARNING level error is generated.

Return Values

A result pointer resource on success or FALSE on failure.

Examples

Example 5.41 mysql_list_processes example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');

$result = mysql_list_processes($link);
while ($row = mysql_fetch_assoc($result)){
    printf("%s %s %s %s %s\n", $row["Id"], $row["Host"], $row["db"],
        $row["Command"], $row["Time"]);
}
mysql_free_result($result);
?>
```

The above example will output something similar to:

```
1 localhost test Processlist 0
4 localhost mysql sleep 5
```

See Also

mysql_thread_id
mysql_stat

5.5.35 mysql_list_tables

- mysql_list_tables

  List tables in a MySQL database

> **Warning**
>
> This function was deprecated in PHP 4.3.0, and it and the entire original MySQL extension was removed in PHP 7.0.0. Instead, use either the actively developed

> MySQLi or PDO_MySQL extensions. See also the MySQL: choosing an API guide and its related FAQ entry for additional information. Alternatives to this function include:
>
> SQL Query: `SHOW TABLES FROM dbname`

Description

```
 resource mysql_list_tables(
   string database,
   resource link_identifier
       =  =NULL);
```

Retrieves a list of table names from a MySQL database.

This function is deprecated. It is preferable to use `mysql_query` to issue an SQL `SHOW TABLES [FROM db_name] [LIKE 'pattern']` statement instead.

Parameters

`database`	The name of the database
`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

A result pointer resource on success or `FALSE` on failure.

Use the `mysql_tablename` function to traverse this result pointer, or any function for result tables, such as `mysql_fetch_array`.

Changelog

Version	Description
4.3.7	This function became deprecated.

Examples

Example 5.42 `mysql_list_tables` alternative example

```
<?php
$dbname = 'mysql_dbname';

if (!mysql_connect('mysql_host', 'mysql_user', 'mysql_password')) {
    echo 'Could not connect to mysql';
    exit;
}

$sql = "SHOW TABLES FROM $dbname";
$result = mysql_query($sql);

if (!$result) {
    echo "DB Error, could not list tables\n";
    echo 'MySQL Error: ' . mysql_error();
```

```
    exit;
}

while ($row = mysql_fetch_row($result)) {
    echo "Table: {$row[0]}\n";
}

mysql_free_result($result);
?>
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used: `mysql_listtables`

See Also

`mysql_list_dbs`
`mysql_tablename`

5.5.36 `mysql_num_fields`

- `mysql_num_fields`

  Get number of fields in result

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_num_fields`
> `PDOStatement::columnCount`

Description

```
  int mysql_num_fields(
    resource result);
```

Retrieves the number of fields from a query.

Parameters

`result`	The result resource that is being evaluated. This result comes from a call to `mysql_query`.

Return Values

Returns the number of fields in the result set resource on success or FALSE on failure.

Examples

Example 5.43 A `mysql_num_fields` example

```
<?php
$result = mysql_query("SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' . mysql_error();
    exit;
}

/* returns 2 because id,email === two fields */
echo mysql_num_fields($result);
?>
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used:
> `mysql_numfields`

See Also

`mysql_select_db`
`mysql_query`
`mysql_fetch_field`
`mysql_num_rows`

5.5.37 `mysql_num_rows`

- `mysql_num_rows`

  Get number of rows in result

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_num_rows`
> `mysqli_stmt_num_rows`
> `PDOStatement::rowCount`

Description

```
  int mysql_num_rows(
    resource result);
```

Retrieves the number of rows from a result set. This command is only valid for statements like SELECT or SHOW that return an actual result set. To retrieve the number of rows affected by a INSERT, UPDATE, REPLACE or DELETE query, use `mysql_affected_rows`.

Parameters

result	The result resource that is being evaluated. This result comes from a call to mysql_query.

Return Values

The number of rows in a result set on success or FALSE on failure.

Examples

Example 5.44 mysql_num_rows example

```
<?php

$link = mysql_connect("localhost", "mysql_user", "mysql_password");
mysql_select_db("database", $link);

$result = mysql_query("SELECT * FROM table1", $link);
$num_rows = mysql_num_rows($result);

echo "$num_rows Rows\n";

?>
```

Notes

> **Note**
>
> If you use mysql_unbuffered_query, mysql_num_rows will not return the correct value until all the rows in the result set have been retrieved.

> **Note**
>
> For backward compatibility, the following deprecated alias may be used: mysql_numrows

See Also

mysql_affected_rows
mysql_connect
mysql_data_seek
mysql_select_db
mysql_query

5.5.38 mysql_pconnect

- mysql_pconnect

  Open a persistent connection to a MySQL server

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL:

> choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_connect` with `p:` host prefix
> `PDO::__construct` with `PDO::ATTR_PERSISTENT` as a driver option

Description

```
resource mysql_pconnect(
  string server
      =  =ini_get("mysql.default_host"),
  string username
      =  =ini_get("mysql.default_user"),
  string password
      =  =ini_get("mysql.default_password"),
  int client_flags
      =  =0);
```

Establishes a persistent connection to a MySQL server.

`mysql_pconnect` acts very much like `mysql_connect` with two major differences.

First, when connecting, the function would first try to find a (persistent) link that's already open with the same host, username and password. If one is found, an identifier for it will be returned instead of opening a new connection.

Second, the connection to the SQL server will not be closed when the execution of the script ends. Instead, the link will remain open for future use (`mysql_close` will not close links established by `mysql_pconnect`).

This type of link is therefore called 'persistent'.

Parameters

`server`	The MySQL server. It can also include a port number. e.g. "hostname:port" or a path to a local socket e.g. ":/path/to/socket" for the localhost. If the PHP directive mysql.default_host is undefined (default), then the default value is 'localhost:3306'
`username`	The username. Default value is the name of the user that owns the server process.
`password`	The password. Default value is an empty password.
`client_flags`	The *`client_flags`* parameter can be a combination of the following constants: 128 (enable `LOAD DATA LOCAL` handling), `MYSQL_CLIENT_SSL`, `MYSQL_CLIENT_COMPRESS`, `MYSQL_CLIENT_IGNORE_SPACE` or `MYSQL_CLIENT_INTERACTIVE`.

Return Values

Returns a MySQL persistent link identifier on success, or `FALSE` on failure.

Changelog

Version	Description
5.5.0	This function will generate an `E_DEPRECATED` error.

Notes

> **Note**
>
> Note, that these kind of links only work if you are using a module version of PHP. See the Persistent Database Connections section for more information.

> **Warning**
>
> Using persistent connections can require a bit of tuning of your Apache and MySQL configurations to ensure that you do not exceed the number of connections allowed by MySQL.

> **Note**
>
> You can suppress the error message on failure by prepending a @ to the function name.

See Also

mysql_connect
Persistent Database Connections

5.5.39 mysql_ping

- mysql_ping

  Ping a server connection or reconnect if there is no connection

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> mysqli_ping

Description

```
bool mysql_ping(
  resource link_identifier
      =  =NULL);
```

Checks whether or not the connection to the server is working. If it has gone down, an automatic reconnection is attempted. This function can be used by scripts that remain idle for a long while, to check whether or not the server has closed the connection and reconnect if necessary.

> **Note**
>
> Automatic reconnection is disabled by default in versions of MySQL >= 5.0.3.

Parameters

link_identifier	The MySQL connection. If the link identifier is not specified, the last link opened by mysql_connect is assumed. If no such link is found,

it will try to create one as if mysql_connect had been called with no arguments. If no connection is found or established, an E_WARNING level error is generated.

Return Values

Returns TRUE if the connection to the server MySQL server is working, otherwise FALSE.

Examples

Example 5.45 A mysql_ping example

```
<?php
set_time_limit(0);

$conn = mysql_connect('localhost', 'mysqluser', 'mypass');
$db   = mysql_select_db('mydb');

/* Assuming this query will take a long time */
$result = mysql_query($sql);
if (!$result) {
    echo 'Query #1 failed, exiting.';
    exit;
}

/* Make sure the connection is still alive, if not, try to reconnect */
if (!mysql_ping($conn)) {
    echo 'Lost connection, exiting after query #1';
    exit;
}
mysql_free_result($result);

/* So the connection is still alive, let's run another query */
$result2 = mysql_query($sql2);
?>
```

See Also

mysql_thread_id
mysql_list_processes

5.5.40 mysql_query

- mysql_query

  Send a MySQL query

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> mysqli_query

PDO::query

Description

```
mixed mysql_query(
  string query,
  resource link_identifier
      =  =NULL);
```

`mysql_query` sends a unique query (multiple queries are not supported) to the currently active database on the server that's associated with the specified *`link_identifier`*.

Parameters

`query`	An SQL query The query string should not end with a semicolon. Data inside the query should be properly escaped.
`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

For SELECT, SHOW, DESCRIBE, EXPLAIN and other statements returning resultset, `mysql_query` returns a resource on success, or `FALSE` on error.

For other type of SQL statements, INSERT, UPDATE, DELETE, DROP, etc, `mysql_query` returns `TRUE` on success or `FALSE` on error.

The returned result resource should be passed to `mysql_fetch_array`, and other functions for dealing with result tables, to access the returned data.

Use `mysql_num_rows` to find out how many rows were returned for a SELECT statement or `mysql_affected_rows` to find out how many rows were affected by a DELETE, INSERT, REPLACE, or UPDATE statement.

`mysql_query` will also fail and return `FALSE` if the user does not have permission to access the table(s) referenced by the query.

Examples

Example 5.46 Invalid Query

The following query is syntactically invalid, so `mysql_query` fails and returns `FALSE`.

```
<?php
$result = mysql_query('SELECT * WHERE 1=1');
if (!$result) {
    die('Invalid query: ' . mysql_error());
}

?>
```

Example 5.47 Valid Query

The following query is valid, so `mysql_query` returns a resource.

```
<?php
// This could be supplied by a user, for example
$firstname = 'fred';
$lastname  = 'fox';

// Formulate Query
// This is the best way to perform an SQL query
// For more examples, see mysql_real_escape_string()
$query = sprintf("SELECT firstname, lastname, address, age FROM friends
    WHERE firstname='%s' AND lastname='%s'",
    mysql_real_escape_string($firstname),
    mysql_real_escape_string($lastname));

// Perform Query
$result = mysql_query($query);

// Check result
// This shows the actual query sent to MySQL, and the error. Useful for debugging.
if (!$result) {
    $message  = 'Invalid query: ' . mysql_error() . "\n";
    $message .= 'Whole query: ' . $query;
    die($message);
}

// Use result
// Attempting to print $result won't allow access to information in the resource
// One of the mysql result functions must be used
// See also mysql_result(), mysql_fetch_array(), mysql_fetch_row(), etc.
while ($row = mysql_fetch_assoc($result)) {
    echo $row['firstname'];
    echo $row['lastname'];
    echo $row['address'];
    echo $row['age'];
}

// Free the resources associated with the result set
// This is done automatically at the end of the script
mysql_free_result($result);
?>
```

See Also

`mysql_connect`
`mysql_error`
`mysql_real_escape_string`
`mysql_result`
`mysql_fetch_assoc`
`mysql_unbuffered_query`

5.5.41 `mysql_real_escape_string`

- `mysql_real_escape_string`

Escapes special characters in a string for use in an SQL statement

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_real_escape_string`
> `PDO::quote`

Description

```
string mysql_real_escape_string(
  string unescaped_string,
  resource link_identifier
      =  =NULL);
```

Escapes special characters in the *`unescaped_string`*, taking into account the current character set of the connection so that it is safe to place it in a `mysql_query`. If binary data is to be inserted, this function must be used.

`mysql_real_escape_string` calls MySQL's library function mysql_real_escape_string, which prepends backslashes to the following characters: `\x00`, `\n`, `\r`, `\`, `'`, `"` and `\x1a`.

This function must always (with few exceptions) be used to make data safe before sending a query to MySQL.

> **Security: the default character set**
>
> The character set must be set either at the server level, or with the API function `mysql_set_charset` for it to affect `mysql_real_escape_string`. See the concepts section on character sets for more information.

Parameters

`unescaped_string`	The string that is to be escaped.
`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

Returns the escaped string, or `FALSE` on error.

Errors/Exceptions

Executing this function without a MySQL connection present will also emit `E_WARNING` level PHP errors. Only execute this function with a valid MySQL connection present.

Examples

Example 5.48 Simple `mysql_real_escape_string` example

```
<?php
// Connect
$link = mysql_connect('mysql_host', 'mysql_user', 'mysql_password')
    OR die(mysql_error());

// Query
$query = sprintf("SELECT * FROM users WHERE user='%s' AND password='%s'",
            mysql_real_escape_string($user),
            mysql_real_escape_string($password));
?>
```

Example 5.49 `mysql_real_escape_string` requires a connection example

This example demonstrates what happens if a MySQL connection is not present when calling this function.

```
<?php
// We have not connected to MySQL

$lastname  = "O'Reilly";
$_lastname = mysql_real_escape_string($lastname);

$query = "SELECT * FROM actors WHERE last_name = '$_lastname'";

var_dump($_lastname);
var_dump($query);
?>
```

The above example will output something similar to:

```
Warning: mysql_real_escape_string(): No such file or directory in /this/test/script.php on line 5
Warning: mysql_real_escape_string(): A link to the server could not be established in /this/test/script.php on

bool(false)
string(41) "SELECT * FROM actors WHERE last_name = ''"
```

Example 5.50 An example SQL Injection Attack

```
<?php
// We didn't check $_POST['password'], it could be anything the user wanted! For example:
$_POST['username'] = 'aidan';
$_POST['password'] = "' OR ''='";

// Query database to check if there are any matching users
$query = "SELECT * FROM users WHERE user='{$_POST['username']}' AND password='{$_POST['password']}'";
mysql_query($query);

// This means the query sent to MySQL would be:
echo $query;
?>
```

The query sent to MySQL:

```
SELECT * FROM users WHERE user='aidan' AND password='' OR ''=''
```

This would allow anyone to log in without a valid password.

Notes

> **Note**
>
> A MySQL connection is required before using `mysql_real_escape_string` otherwise an error of level `E_WARNING` is generated, and `FALSE` is returned. If *`link_identifier`* isn't defined, the last MySQL connection is used.

> **Note**
>
> If magic_quotes_gpc is enabled, first apply `stripslashes` to the data. Using this function on data which has already been escaped will escape the data twice.

> **Note**
>
> If this function is not used to escape data, the query is vulnerable to SQL Injection Attacks.

> **Note**
>
> `mysql_real_escape_string` does not escape `%` and `_`. These are wildcards in MySQL if combined with `LIKE`, `GRANT`, or `REVOKE`.

See Also

`mysql_set_charset`
`mysql_client_encoding`
`addslashes`
`stripslashes`
The magic_quotes_gpc directive
The magic_quotes_runtime directive

5.5.42 `mysql_result`

- `mysql_result`

  Get result data

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_data_seek` in conjunction with `mysqli_field_seek` and `mysqli_fetch_field`

PDOStatement::fetchColumn

Description

```
string mysql_result(
  resource result,
  int row,
  mixed field
      =  =0);
```

Retrieves the contents of one cell from a MySQL result set.

When working on large result sets, you should consider using one of the functions that fetch an entire row (specified below). As these functions return the contents of multiple cells in one function call, they're MUCH quicker than `mysql_result`. Also, note that specifying a numeric offset for the field argument is much quicker than specifying a fieldname or tablename.fieldname argument.

Parameters

result	The result resource that is being evaluated. This result comes from a call to `mysql_query`.
row	The row number from the result that's being retrieved. Row numbers start at `0`.
field	The name or offset of the field being retrieved. It can be the field's offset, the field's name, or the field's table dot field name (tablename.fieldname). If the column name has been aliased ('select foo as bar from...'), use the alias instead of the column name. If undefined, the first field is retrieved.

Return Values

The contents of one cell from a MySQL result set on success, or `FALSE` on failure.

Examples

Example 5.51 `mysql_result` example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
if (!mysql_select_db('database_name')) {
    die('Could not select database: ' . mysql_error());
}
$result = mysql_query('SELECT name FROM work.employee');
if (!$result) {
    die('Could not query:' . mysql_error());
}
echo mysql_result($result, 2); // outputs third employee's name

mysql_close($link);
?>
```

Notes

> **Note**
>
> Calls to `mysql_result` should not be mixed with calls to other functions that deal with the result set.

See Also

`mysql_fetch_row`
`mysql_fetch_array`
`mysql_fetch_assoc`
`mysql_fetch_object`

5.5.43 `mysql_select_db`

- `mysql_select_db`

  Select a MySQL database

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_select_db`
> `PDO::__construct` (part of dsn)

Description

```
bool mysql_select_db(
  string database_name,
  resource link_identifier
      =  =NULL);
```

Sets the current active database on the server that's associated with the specified link identifier. Every subsequent call to `mysql_query` will be made on the active database.

Parameters

`database_name`	The name of the database that is to be selected.
`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

Returns `TRUE` on success or `FALSE` on failure.

Examples

Example 5.52 `mysql_select_db` example

```
```

```
<?php

$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
if (!$link) {
    die('Not connected : ' . mysql_error());
}

// make foo the current db
$db_selected = mysql_select_db('foo', $link);
if (!$db_selected) {
    die ('Can\'t use foo : ' . mysql_error());
}
?>
```

Notes

> **Note**
>
> For backward compatibility, the following deprecated alias may be used: `mysql_selectdb`

See Also

`mysql_connect`
`mysql_pconnect`
`mysql_query`

5.5.44 `mysql_set_charset`

- `mysql_set_charset`

  Sets the client character set

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_set_charset`
> PDO: Add `charset` to the connection string, such as `charset=utf8`

Description

```
bool mysql_set_charset(
  string charset,
  resource link_identifier
      =  =NULL);
```

Sets the default character set for the current connection.

Parameters

`charset`	A valid character set name.

link_identifier — The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

Returns `TRUE` on success or `FALSE` on failure.

Notes

> **Note**
>
> This function requires MySQL 5.0.7 or later.

> **Note**
>
> This is the preferred way to change the charset. Using `mysql_query` to set it (such as `SET NAMES utf8`) is not recommended. See the MySQL character set concepts section for more information.

See Also

Setting character sets in MySQL
List of character sets that MySQL supports
`mysql_client_encoding`

5.5.45 `mysql_stat`

- `mysql_stat`

  Get current system status

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_stat`
> `PDO::getAttribute(PDO::ATTR_SERVER_INFO)`

Description

```
string mysql_stat(
  resource link_identifier
      =  =NULL);
```

`mysql_stat` returns the current server status.

Parameters

link_identifier — The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found,

it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

Returns a string with the status for uptime, threads, queries, open tables, flush tables and queries per second. For a complete list of other status variables, you have to use the `SHOW STATUS` SQL command. If *`link_identifier`* is invalid, `NULL` is returned.

Examples

Example 5.53 `mysql_stat` example

```
<?php
$link   = mysql_connect('localhost', 'mysql_user', 'mysql_password');
$status = explode('  ', mysql_stat($link));
print_r($status);
?>
```

The above example will output something similar to:

```
Array
(
    [0] => Uptime: 5380
    [1] => Threads: 2
    [2] => Questions: 1321299
    [3] => Slow queries: 0
    [4] => Opens: 26
    [5] => Flush tables: 1
    [6] => Open tables: 17
    [7] => Queries per second avg: 245.595
)
```

Example 5.54 Alternative `mysql_stat` example

```
<?php
$link   = mysql_connect('localhost', 'mysql_user', 'mysql_password');
$result = mysql_query('SHOW STATUS', $link);
while ($row = mysql_fetch_assoc($result)) {
    echo $row['Variable_name'] . ' = ' . $row['Value'] . "\n";
}
?>
```

The above example will output something similar to:

```
back_log = 50
basedir = /usr/local/
bdb_cache_size = 8388600
bdb_log_buffer_size = 32768
bdb_home = /var/db/mysql/
```

```
bdb_max_lock = 10000
bdb_logdir =
bdb_shared_data = OFF
bdb_tmpdir = /var/tmp/
...
```

See Also

mysql_get_server_info
mysql_list_processes

5.5.46 mysql_tablename

- mysql_tablename

  Get table name of field

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> SQL Query: SHOW TABLES

Description

```
string mysql_tablename(
  resource result,
  int i);
```

Retrieves the table name from a *result*.

This function is deprecated. It is preferable to use mysql_query to issue an SQL SHOW TABLES [FROM db_name] [LIKE 'pattern'] statement instead.

Parameters

result	A result pointer resource that's returned from mysql_list_tables.
i	The integer index (row/table number)

Return Values

The name of the table on success or FALSE on failure.

Use the mysql_tablename function to traverse this result pointer, or any function for result tables, such as mysql_fetch_array.

Changelog

Version	Description
5.5.0	The mysql_tablename function is deprecated, and emits an E_DEPRECATED level error.

Examples

Example 5.55 `mysql_tablename` example

```
<?php
mysql_connect("localhost", "mysql_user", "mysql_password");
$result = mysql_list_tables("mydb");
$num_rows = mysql_num_rows($result);
for ($i = 0; $i < $num_rows; $i++) {
    echo "Table: ", mysql_tablename($result, $i), "\n";
}

mysql_free_result($result);
?>
```

Notes

> **Note**
>
> The `mysql_num_rows` function may be used to determine the number of tables in the result pointer.

See Also

`mysql_list_tables`
`mysql_field_table`
`mysql_db_name`

5.5.47 `mysql_thread_id`

- `mysql_thread_id`

  Return the current thread ID

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> `mysqli_thread_id`

Description

```
int mysql_thread_id(
  resource link_identifier
      =  =NULL);
```

Retrieves the current thread ID. If the connection is lost, and a reconnect with `mysql_ping` is executed, the thread ID will change. This means only retrieve the thread ID when needed.

Parameters

link_identifier	The MySQL connection. If the link identifier is not specified, the last link opened by mysql_connect is assumed. If no such link is found, it will try to create one as if mysql_connect had been called with no arguments. If no connection is found or established, an E_WARNING level error is generated.

Return Values

The thread ID on success or FALSE on failure.

Examples

Example 5.56 mysql_thread_id example

```
<?php
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
$thread_id = mysql_thread_id($link);
if ($thread_id){
    printf("current thread id is %d\n", $thread_id);
}
?>
```

The above example will output something similar to:

```
current thread id is 73
```

See Also

mysql_ping
mysql_list_processes

5.5.48 mysql_unbuffered_query

- mysql_unbuffered_query

  Send an SQL query to MySQL without fetching and buffering the result rows.

> **Warning**
>
> This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the MySQLi or PDO_MySQL extension should be used. See also MySQL: choosing an API guide and related FAQ for more information. Alternatives to this function include:
>
> See: Buffered and Unbuffered queries

Description

```
  resource mysql_unbuffered_query(
    string query,
```

```
    resource link_identifier
        =  =NULL);
```

`mysql_unbuffered_query` sends the SQL query *`query`* to MySQL without automatically fetching and buffering the result rows as `mysql_query` does. This saves a considerable amount of memory with SQL queries that produce large result sets, and you can start working on the result set immediately after the first row has been retrieved as you don't have to wait until the complete SQL query has been performed. To use `mysql_unbuffered_query` while multiple database connections are open, you must specify the optional parameter *`link_identifier`* to identify which connection you want to use.

Parameters

`query`	The SQL query to execute. Data inside the query should be properly escaped.
`link_identifier`	The MySQL connection. If the link identifier is not specified, the last link opened by `mysql_connect` is assumed. If no such link is found, it will try to create one as if `mysql_connect` had been called with no arguments. If no connection is found or established, an `E_WARNING` level error is generated.

Return Values

For SELECT, SHOW, DESCRIBE or EXPLAIN statements, `mysql_unbuffered_query` returns a resource on success, or `FALSE` on error.

For other type of SQL statements, UPDATE, DELETE, DROP, etc, `mysql_unbuffered_query` returns `TRUE` on success or `FALSE` on error.

Notes

> **Note**
>
> The benefits of `mysql_unbuffered_query` come at a cost: you cannot use `mysql_num_rows` and `mysql_data_seek` on a result set returned from `mysql_unbuffered_query`, until all rows are fetched. You also have to fetch all result rows from an unbuffered SQL query before you can send a new SQL query to MySQL, using the same *`link_identifier`*.

See Also

`mysql_query`

Chapter 6 MySQL Native Driver

Table of Contents

MySQL Native Driver is a replacement for the MySQL Client Library (libmysqlclient). MySQL Native Driver is part of the official PHP sources as of PHP 5.3.0.

The MySQL database extensions MySQL extension, `mysqli` and PDO MYSQL all communicate with the MySQL server. In the past, this was done by the extension using the services provided by the MySQL Client Library. The extensions were compiled against the MySQL Client Library in order to use its client-server protocol.

With MySQL Native Driver there is now an alternative, as the MySQL database extensions can be compiled to use MySQL Native Driver instead of the MySQL Client Library.

MySQL Native Driver is written in C as a PHP extension.

6.1 Overview

What it is not

Although MySQL Native Driver is written as a PHP extension, it is important to note that it does not provide a new API to the PHP programmer. The programmer APIs for MySQL database connectivity are provided by the MySQL extension, `mysqli` and PDO MYSQL. These extensions can now use the services of MySQL Native Driver to communicate with the MySQL Server. Therefore, you should not think of MySQL Native Driver as an API.

Why use it?

Using the MySQL Native Driver offers a number of advantages over using the MySQL Client Library.

The older MySQL Client Library was written by MySQL AB (now Oracle Corporation) and so was released under the MySQL license. This ultimately led to MySQL support being disabled by default in PHP.

However, the MySQL Native Driver has been developed as part of the PHP project, and is therefore released under the PHP license. This removes licensing issues that have been problematic in the past.

Also, in the past, you needed to build the MySQL database extensions against a copy of the MySQL Client Library. This typically meant you needed to have MySQL installed on a machine where you were building the PHP source code. Also, when your PHP application was running, the MySQL database extensions would call down to the MySQL Client library file at run time, so the file needed to be installed on your system. With MySQL Native Driver that is no longer the case as it is included as part of the standard distribution. So you do not need MySQL installed in order to build PHP or run PHP database applications.

Because MySQL Native Driver is written as a PHP extension, it is tightly coupled to the workings of PHP. This leads to gains in efficiency, especially when it comes to memory usage, as the driver uses the PHP memory management system. It also supports the PHP memory limit. Using MySQL Native Driver leads to comparable or better performance than using MySQL Client Library, it always ensures the most efficient use of memory. One example of the memory efficiency is the fact that when using the MySQL Client Library, each row is stored in memory twice, whereas with the MySQL Native Driver each row is only stored once in memory.

> **Reporting memory usage**
>
> Because MySQL Native Driver uses the PHP memory management system, its memory usage can be tracked with `memory_get_usage`. This is not possible with libmysqlclient because it uses the C function malloc() instead.

Special features

MySQL Native Driver also provides some special features not available when the MySQL database extensions use MySQL Client Library. These special features are listed below:

- Improved persistent connections
- The special function `mysqli_fetch_all`
- Performance statistics calls: `mysqli_get_cache_stats`, `mysqli_get_client_stats`, `mysqli_get_connection_stats`

The performance statistics facility can prove to be very useful in identifying performance bottlenecks.

MySQL Native Driver also allows for persistent connections when used with the `mysqli` extension.

SSL Support

MySQL Native Driver has supported SSL since PHP version 5.3.3

Compressed Protocol Support

As of PHP 5.3.2 MySQL Native Driver supports the compressed client server protocol. MySQL Native Driver did not support this in 5.3.0 and 5.3.1. Extensions such as `ext/mysql`, `ext/mysqli`, that are configured to use MySQL Native Driver, can also take advantage of this feature. Note that `PDO_MYSQL` does *NOT* support compression when used together with mysqlnd.

Named Pipes Support

Named pipes support for Windows was added in PHP version 5.4.0.

6.2 Installation

Changelog

Table 6.1 Changelog

Version	Description
5.3.0	The MySQL Native Driver was added, with support for all MySQL extensions (i.e., mysql, mysqli and PDO_MYSQL). Passing in `mysqlnd` to the appropriate configure switch enables this support.
5.4.0	The MySQL Native Driver is now the default for all MySQL extensions (i.e., mysql, mysqli and PDO_MYSQL). Passing in `mysqlnd` to configure is now optional.
5.5.0	SHA-256 Authentication Plugin support was added

Installation on Unix

The MySQL database extensions must be configured to use the MySQL Client Library. In order to use the MySQL Native Driver, PHP needs to be built specifying that the MySQL database extensions are compiled with MySQL Native Driver support. This is done through configuration options prior to building the PHP source code.

For example, to build the MySQL extension, `mysqli` and PDO MYSQL using the MySQL Native Driver, the following command would be given:

```
 ./configure --with-mysql=mysqlnd \
--with-mysqli=mysqlnd \
--with-pdo-mysql=mysqlnd \
[other options]
```

Installation on Windows

In the official PHP Windows distributions from 5.3 onwards, MySQL Native Driver is enabled by default, so no additional configuration is required to use it. All MySQL database extensions will use MySQL Native Driver in this case.

SHA-256 Authentication Plugin support

The MySQL Native Driver requires the OpenSSL functionality of PHP to be loaded and enabled to connect to MySQL through accounts that use the MySQL SHA-256 Authentication Plugin. For example, PHP could be configured using:

```
./configure --with-mysql=mysqlnd \
--with-mysqli=mysqlnd \
--with-pdo-mysql=mysqlnd \
--with-openssl
[other options]
```

6.3 Runtime Configuration

The behaviour of these functions is affected by settings in `php.ini`.

Table 6.2 MySQL Native Driver Configuration Options

Name	Default	Changeable	Changelog
mysqlnd.collect_statistics	"1"	PHP_INI_SYSTEM	Available since PHP 5.3.0.
mysqlnd.collect_memory_statistics	"0"	PHP_INI_SYSTEM	Available since PHP 5.3.0.
mysqlnd.debug	""	PHP_INI_SYSTEM	Available since PHP 5.3.0.
mysqlnd.log_mask	0	PHP_INI_ALL	Available since PHP 5.3.0
mysqlnd.mempool_default_size	16000	PHP_INI_ALL	Available since PHP 5.3.3
mysqlnd.net_read_timeout	"31536000"	PHP_INI_SYSTEM	Available since PHP 5.3.0.
mysqlnd.net_cmd_buffer_size	5.3.0 - "2048", 5.3.1 - "4096"	PHP_INI_SYSTEM	Available since PHP 5.3.0.
mysqlnd.net_read_buffer_size	"32768"	PHP_INI_SYSTEM	Available since PHP 5.3.0.
mysqlnd.sha256_server_public_key	""	PHP_INI_PERDIR	Available since PHP 5.5.0.
mysqlnd.fetch_data_copy	0	PHP_INI_ALL	Available since PHP 5.6.0.

For further details and definitions of the PHP_INI_* modes, see the http://www.php.net/manual/en/configuration.changes.modes.

Here's a short explanation of the configuration directives.

`mysqlnd.collect_statistics` boolean
: Enables the collection of various client statistics which can be accessed through `mysqli_get_client_stats`, `mysqli_get_connection_stats`, `mysqli_get_cache_stats` and are shown in `mysqlnd` section of the output of the `phpinfo` function as well.

  This configuration setting enables all MySQL Native Driver statistics except those relating to memory management.

`mysqlnd.collect_memory_statistics` boolean
: Enable the collection of various memory statistics which can be accessed through `mysqli_get_client_stats`, `mysqli_get_connection_stats`, `mysqli_get_cache_stats` and are shown in `mysqlnd` section of the output of the `phpinfo` function as well.

  This configuration setting enables the memory management statistics within the overall set of MySQL Native Driver statistics.

`mysqlnd.debug` string
: Records communication from all extensions using `mysqlnd` to the specified log file.

The format of the directive is `mysqlnd.debug = "option1[,parameter_option1] [:option2[,parameter_option2]]"`.

The options for the format string are as follows:

- A[,file] - Appends trace output to specified file. Also ensures that data is written after each write. This is done by closing and reopening the trace file (this is slow). It helps ensure a complete log file should the application crash.
- a[,file] - Appends trace output to the specified file.
- d - Enables output from DBUG_<N> macros for the current state. May be followed by a list of keywords which selects output only for the DBUG macros with that keyword. An empty list of keywords implies output for all macros.
- f[,functions] - Limits debugger actions to the specified list of functions. An empty list of functions implies that all functions are selected.
- F - Marks each debugger output line with the name of the source file containing the macro causing the output.
- i - Marks each debugger output line with the PID of the current process.
- L - Marks each debugger output line with the name of the source file line number of the macro causing the output.
- n - Marks each debugger output line with the current function nesting depth
- o[,file] - Similar to a[,file] but overwrites old file, and does not append.
- O[,file] - Similar to A[,file] but overwrites old file, and does not append.
- t[,N] - Enables function control flow tracing. The maximum nesting depth is specified by N, and defaults to 200.
- x - This option activates profiling.
- m - Trace memory allocation and deallocation related calls.

Example:

```
d:t:x:O,/tmp/mysqlnd.trace
```

> **Note**
>
> This feature is only available with a debug build of PHP. Works on Microsoft Windows if using

a debug build of PHP and PHP was built using Microsoft Visual C version 9 and above.

mysqlnd.log_mask integer

Defines which queries will be logged. The default 0, which disables logging. Define using an integer, and not with PHP constants. For example, a value of 48 (16 + 32) will log slow queries which either use 'no good index' (SERVER_QUERY_NO_GOOD_INDEX_USED = 16) or no index at all (SERVER_QUERY_NO_INDEX_USED = 32). A value of 2043 (1 + 2 + 8 + ... + 1024) will log all slow query types.

The types are as follows: SERVER_STATUS_IN_TRANS=1, SERVER_STATUS_AUTOCOMMIT=2, SERVER_MORE_RESULTS_EXISTS=8, SERVER_QUERY_NO_GOOD_INDEX_USED=16, SERVER_QUERY_NO_INDEX_USED=32, SERVER_STATUS_CURSOR_EXISTS=64, SERVER_STATUS_LAST_ROW_SENT=128, SERVER_STATUS_DB_DROPPED=256, SERVER_STATUS_NO_BACKSLASH_ESCAPES=512, and SERVER_QUERY_WAS_SLOW=1024.

mysqlnd.mempool_default_size integer

Default size of the mysqlnd memory pool, which is used by result sets.

mysqlnd.net_read_timeout integer

`mysqlnd` and the MySQL Client Library, `libmysqlclient` use different networking APIs. `mysqlnd` uses PHP streams, whereas `libmysqlclient` uses its own wrapper around the operating level network calls. PHP, by default, sets a read timeout of 60s for streams. This is set via `php.ini`, `default_socket_timeout`. This default applies to all streams that set no other timeout value. `mysqlnd` does not set any other value and therefore connections of long running queries can be disconnected after `default_socket_timeout` seconds resulting in an error message “2006 - MySQL Server has gone away”. The MySQL Client Library sets a default timeout of 365 * 24 * 3600 seconds (1 year) and waits for other timeouts to occur, such as TCP/IP timeouts. `mysqlnd` now uses the same very long timeout. The value is configurable through a new `php.ini` setting: `mysqlnd.net_read_timeout`. `mysqlnd.net_read_timeout` gets used by any extension (`ext/mysql`, `ext/mysqli`, `PDO_MySQL`) that uses `mysqlnd`. `mysqlnd` tells PHP Streams to use `mysqlnd.net_read_timeout`. Please note that there may be subtle differences between `MYSQL_OPT_READ_TIMEOUT` from the MySQL Client Library and PHP Streams, for example `MYSQL_OPT_READ_TIMEOUT` is documented to work only for TCP/IP connections and, prior to MySQL 5.1.2, only for Windows. PHP streams may not have this limitation. Please check the streams documentation, if in doubt.

mysqlnd.net_cmd_buffer_size long

`mysqlnd` allocates an internal command/network buffer of `mysqlnd.net_cmd_buffer_size` (in `php.ini`) bytes for every connection. If a MySQL Client Server protocol command, for example, `COM_QUERY` (“normal” query), does not fit into the buffer, `mysqlnd` will grow the buffer to the size required for sending the

command. Whenever the buffer gets extended for one connection, `command_buffer_too_small` will be incremented by one.

If `mysqlnd` has to grow the buffer beyond its initial size of `mysqlnd.net_cmd_buffer_size` bytes for almost every connection, you should consider increasing the default size to avoid re-allocations.

The default buffer size is 2048 bytes in PHP 5.3.0. In later versions the default is 4096 bytes.

It is recommended that the buffer size be set to no less than 4096 bytes because `mysqlnd` also uses it when reading certain communication packet from MySQL. In PHP 5.3.0, `mysqlnd` will not grow the buffer if MySQL sends a packet that is larger than the current size of the buffer. As a consequence, `mysqlnd` is unable to decode the packet and the client application will get an error. There are only two situations when the packet can be larger than the 2048 bytes default of `mysqlnd.net_cmd_buffer_size` in PHP 5.3.0: the packet transports a very long error message, or the packet holds column meta data from `COM_LIST_FIELD` (`mysql_list_fields()` and the meta data come from a string column with a very long default value (>1900 bytes).

As of PHP 5.3.2 mysqlnd does not allow setting buffers smaller than 4096 bytes.

The value can also be set using `mysqli_options(link, MYSQLI_OPT_NET_CMD_BUFFER_SIZE, size)`.

`mysqlnd.net_read_buffer_size`
long

Maximum read chunk size in bytes when reading the body of a MySQL command packet. The MySQL client server protocol encapsulates all its commands in packets. The packets consist of a small header and a body with the actual payload. The size of the body is encoded in the header. `mysqlnd` reads the body in chunks of `MIN(header.size, mysqlnd.net_read_buffer_size)` bytes. If a packet body is larger than `mysqlnd.net_read_buffer_size` bytes, `mysqlnd` has to call `read()` multiple times.

The value can also be set using `mysqli_options(link, MYSQLI_OPT_NET_READ_BUFFER_SIZE, size)`.

`mysqlnd.sha256_server_public_key`
string

SHA-256 Authentication Plugin related. File with the MySQL server public RSA key.

Clients can either omit setting a public RSA key, specify the key through this PHP configuration setting or set the key at runtime using `mysqli_options`. If not public RSA key file is given by the client, then the key will be exchanged as part of the standard SHA-256 Authentication Plugin authentication procedure.

`mysqlnd.fetch_data_copy`
long

Enforce copying result sets from the internal result set buffers into PHP variables instead of using the default reference and copy-on-write logic. Please, see the memory management implementation notes for further details.

Copying result sets instead of having PHP variables reference them allows releasing the memory occupied for the PHP variables earlier.

Depending on the user API code, the actual database quries and the size of their result sets this may reduce the memory footprint of mysqlnd.

Do not set if using PDO_MySQL. PDO_MySQL has not yet been updated to support the new fetch mode.

6.4 Incompatibilities

MySQL Native Driver is in most cases compatible with MySQL Client Library (`libmysql`). This section documents incompatibilities between these libraries.

- Values of `bit` data type are returned as binary strings (e.g. "\0" or "\x1F") with `libmysql` and as decimal strings (e.g. "0" or "31") with `mysqlnd`. If you want the code to be compatible with both libraries then always return bit fields as numbers from MySQL with a query like this: `SELECT bit + 0 FROM table`.

6.5 Persistent Connections

Using Persistent Connections

If `mysqli` is used with `mysqlnd`, when a persistent connection is created it generates a `COM_CHANGE_USER` (`mysql_change_user()`) call on the server. This ensures that re-authentication of the connection takes place.

As there is some overhead associated with the `COM_CHANGE_USER` call, it is possible to switch this off at compile time. Reusing a persistent connection will then generate a `COM_PING` (`mysql_ping`) call to simply test the connection is reusable.

Generation of `COM_CHANGE_USER` can be switched off with the compile flag `MYSQLI_NO_CHANGE_USER_ON_PCONNECT`. For example:

```
shell# CFLAGS="-DMYSQLI_NO_CHANGE_USER_ON_PCONNECT" ./configure --with-mysql=/usr/local/mysql/ --with-mysqli=/
```

Or alternatively:

```
shell# export CFLAGS="-DMYSQLI_NO_CHANGE_USER_ON_PCONNECT"
shell# configure --whatever-option
shell# make clean
shell# make
```

Note that only `mysqli` on `mysqlnd` uses `COM_CHANGE_USER`. Other extension-driver combinations use `COM_PING` on initial use of a persistent connection.

6.6 Statistics

Using Statistical Data

MySQL Native Driver contains support for gathering statistics on the communication between the client and the server. The statistics gathered are of two main types:

- Client statistics
- Connection statistics

If you are using the `mysqli` extension, these statistics can be obtained through two API calls:

- `mysqli_get_client_stats`
- `mysqli_get_connection_stats`

> **Note**
>
> Statistics are aggregated among all extensions that use MySQL Native Driver. For example, when compiling both `ext/mysql` and `ext/mysqli` against MySQL Native Driver, both function calls of `ext/mysql` and `ext/mysqli` will change the statistics. There is no way to find out how much a certain API call of any extension that has been compiled against MySQL Native Driver has impacted a certain statistic. You can configure the PDO MySQL Driver, `ext/mysql` and `ext/mysqli` to optionally use the MySQL Native Driver. When doing so, all three extensions will change the statistics.

Accessing Client Statistics

To access client statistics, you need to call `mysqli_get_client_stats`. The function call does not require any parameters.

The function returns an associative array that contains the name of the statistic as the key and the statistical data as the value.

Client statistics can also be accessed by calling the `phpinfo` function.

Accessing Connection Statistics

To access connection statistics call `mysqli_get_connection_stats`. This takes the database connection handle as the parameter.

The function returns an associative array that contains the name of the statistic as the key and the statistical data as the value.

Buffered and Unbuffered Result Sets

Result sets can be buffered or unbuffered. Using default settings, `ext/mysql` and `ext/mysqli` work with buffered result sets for normal (non prepared statement) queries. Buffered result sets are cached on the client. After the query execution all results are fetched from the MySQL Server and stored in a cache on the client. The big advantage of buffered result sets is that they allow the server to free all resources allocated to a result set, once the results have been fetched by the client.

Unbuffered result sets on the other hand are kept much longer on the server. If you want to reduce memory consumption on the client, but increase load on the server, use unbuffered results. If you experience a high server load and the figures for unbuffered result sets are high, you should consider moving the load to the clients. Clients typically scale better than servers. “Load” does not only refer to memory buffers - the server also needs to keep other resources open, for example file handles and threads, before a result set can be freed.

Prepared Statements use unbuffered result sets by default. However, you can use `mysqli_stmt_store_result` to enable buffered result sets.

Statistics returned by MySQL Native Driver

The following tables show a list of statistics returned by the `mysqli_get_client_stats` and `mysqli_get_connection_stats` functions.

Table 6.3 Returned mysqlnd statistics: Network

Statistic	Scope	Description	Notes
bytes_sent	Connection	Number of bytes sent from PHP to the MySQL server	Can be used to check the efficiency of the compression protocol
bytes_received	Connection	Number of bytes received from MySQL server	Can be used to check the efficiency of the compression protocol
packets_sent	Connection	Number of MySQL Client Server protocol packets sent	Used for debugging Client Server protocol implementation
packets_received	Connection	Number of MySQL Client Server protocol packets received	Used for debugging Client Server protocol implementation
protocol_overhead_in	Connection	MySQL Client Server protocol overhead in bytes for incoming traffic. Currently only the Packet Header (4 bytes) is considered as overhead. protocol_overhead_in = packets_received * 4	Used for debugging Client Server protocol implementation
protocol_overhead_out	Connection	MySQL Client Server protocol overhead in bytes for outgoing traffic. Currently only the Packet Header (4 bytes) is considered as overhead. protocol_overhead_out = packets_sent * 4	Used for debugging Client Server protocol implementation
bytes_received_ok_packet	Connection	Total size of bytes of MySQL Client Server protocol OK packets received. OK packets can contain a status message. The length of the status message can vary and thus the size of an OK packet is not fixed.	Used for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
packets_received_ok	Connection	Number of MySQL Client Server protocol OK packets received.	Used for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
bytes_received_eof_packet	Connection	Total size in bytes of MySQL Client Server protocol EOF packets received. EOF can vary in size depending on the server version. Also, EOF can transport an error message.	Used for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
packets_received_eof	Connection	Number of MySQL Client Server protocol EOF packets. Like with other packet statistics the number of packets will be increased even if PHP does not receive the expected packet but, for example, an error message.	Used for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
bytes_received_rset_header_packet	Connection	Total size in bytes of MySQL Client Server protocol result set header packets.	Used for debugging CS protocol implementation. Note that the total size

Statistic	Scope	Description	Notes
		The size of the packets varies depending on the payload (`LOAD LOCAL INFILE`, `INSERT`, `UPDATE`, `SELECT`, error message).	in bytes includes the size of the header packet (4 bytes, see protocol overhead).
`packets_received_rset_header`	Connection	Number of MySQL Client Server protocol result set header packets.	Used for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
`bytes_received_rset_field_meta_packet`	Connection	Total size in bytes of MySQL Client Server protocol result set meta data (field information) packets. Of course the size varies with the fields in the result set. The packet may also transport an error or an EOF packet in case of COM_LIST_FIELDS.	Only useful for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
`packets_received_rset_field_meta`	Connection	Number of MySQL Client Server protocol result set meta data (field information) packets.	Only useful for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
`bytes_received_rset_row_packet`	Connection	Total size in bytes of MySQL Client Server protocol result set row data packets. The packet may also transport an error or an EOF packet. You can reverse engineer the number of error and EOF packets by subtracting `rows_fetched_from_server_normal` and `rows_fetched_from_server_ps` from `bytes_received_rset_row_packet`.	Only useful for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
`packets_received_rset_row`	Connection	Number of MySQL Client Server protocol result set row data packets and their total size in bytes.	Only useful for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
`bytes_received_prepare_response_packet`	Connection	Total size in bytes of MySQL Client Server protocol OK for Prepared Statement Initialization packets (prepared statement init packets). The packet may also transport an error. The packet size depends on the MySQL version: 9 bytes with MySQL 4.1 and 12 bytes from MySQL 5.0 on. There is no safe way to know how many errors happened. You may be able to guess that an error has occurred if, for example, you always connect to MySQL 5.0 or newer and, `bytes_received_prepare_response_packet` != `packets_received_prepare_response` * 12. See also	Only useful for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).

Statistic	Scope	Description	Notes
		`ps_prepared_never_executed`, `ps_prepared_once_executed`.	
`packets_sent_prepare_close`	Connection	Number of MySQL Client Server protocol OK for Prepared Statement Initialization packets (prepared statement init packets).	Only useful for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
`bytes_sent_change_user`	Connection	Total size in bytes of MySQL Client Server protocol COM_CHANGE_USER packets. The packet may also transport an error or EOF.	Only useful for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
`packets_sent_change_user`	Connection	Number of MySQL Client Server protocol COM_CHANGE_USER packets	Only useful for debugging CS protocol implementation. Note that the total size in bytes includes the size of the header packet (4 bytes, see protocol overhead).
`packets_sent_command`	Connection	Number of MySQL Client Server protocol commands sent from PHP to MySQL. There is no way to know which specific commands and how many of them have been sent. At its best you can use it to check if PHP has sent any commands to MySQL to know if you can consider to disable MySQL support in your PHP binary. There is also no way to reverse engineer the number of errors that may have occurred while sending data to MySQL. The only error that is recorded is command_buffer_too_small (see below).	Only useful for debugging CS protocol implementation.

`bytes_received_real_data_normal`	Connection	Number of bytes of payload fetched by the PHP client from `mysqlnd` using the text protocol.	This is the size of the actual data contained in result sets that do not originate from prepared statements and which have been fetched by the PHP client. Note that although a full result set may have been pulled from MySQL by `mysqlnd`, this statistic only counts actual data pulled from `mysqlnd` by the PHP client. An example of a code sequence that will increase the value is as follows:
			
			`$mysqli = new mysqli();`
			`$res = $mysqli->query("SELECT 'abc'");`
			`$res->fetch_assoc();`
			`$res->close();`
			
			Every fetch operation will increase the value.
			
			The statistic will not be increased if the result set is only buffered on the client,

Statistic	Scope	Description	Notes

			but not fetched, such as in the following example:
			
			`$mysqli = new mysqli();`
			`$res = $mysqli->query("SELECT 'abc'");`
			`$res->close();`
			
			This statistic is available as of PHP version 5.3.4.

`bytes_rec`	Connection	Number of bytes of the payload fetched by the PHP client from `mysqlnd` using the prepared statement protocol.	This is the size of the actual data contained in result sets that originate from prepared statements and which has been fetched by the PHP client. The value will not be increased if the result set is not subsequently read by the PHP client. Note that although a full result set may have been pulled from MySQL by `mysqlnd`, this statistic only counts actual data pulled from `mysqlnd` by the PHP client. See also `bytes_received_real_data_normal`. This statistic is available as of PHP version 5.3.4.

Result Set

Table 6.4 Returned mysqlnd statistics: Result Set

Statistic	Scope	Description	Notes
`result_set_queries`	Connection	Number of queries that have generated a result set. Examples of queries that generate a result set: `SELECT`, `SHOW`. The statistic will not be incremented if there is an error reading the result set header packet from the line.	You may use it as an indirect measure for the number of queries PHP has sent to MySQL, for example, to identify a client that causes a high database load.
`non_result_set_queries`	Connection	Number of queries that did not generate a result set. Examples of queries that do not generate a result set: `INSERT`, `UPDATE`, `LOAD DATA`. The statistic will not be incremented if there is an error reading the result set header packet from the line.	You may use it as an indirect measure for the number of queries PHP has sent to MySQL, for example, to identify a client that causes a high database load.
`no_index_used`	Connection	Number of queries that have generated a result set but did not use an index (see also mysqld start option –log-queries-not-using-indexes). If you want these queries to be reported you can use mysqli_report(MYSQLI_REPORT_INDEX) to make ext/mysqli throw an exception. If you prefer a warning	

Statistic	Scope	Description	Notes
		instead of an exception use mysqli_report(MYSQLI_REPORT_INDEX ^ MYSQLI_REPORT_STRICT).	
bad_index_used	Connection	Number of queries that have generated a result set and did not use a good index (see also mysqld start option –log-slow-queries).	If you want these queries to be reported you can use mysqli_report(MYSQLI_REPORT_INDEX) to make ext/mysqli throw an exception. If you prefer a warning instead of an exception use mysqli_report(MYSQLI_REPORT_INDEX ^ MYSQLI_REPORT_STRICT)
slow_queries	Connection	SQL statements that took more than long_query_time seconds to execute and required at least min_examined_row_limit rows to be examined.	Not reported through mysqli_report
buffered_sets	Connection	Number of buffered result sets returned by “normal” queries. “Normal” means “not prepared statement” in the following notes.	Examples of API calls that will buffer result sets on the client: mysql_query, mysqli_query, mysqli_store_result, mysqli_stmt_get_result. Buffering result sets on the client ensures that server resources are freed as soon as possible and it makes result set scrolling easier. The downside is the additional memory consumption on the client for buffering data. Note that mysqlnd (unlike the MySQL Client Library) respects the PHP memory limit because it uses PHP internal memory management functions to allocate memory. This is also the reason why memory_get_usage reports a higher memory consumption when using mysqlnd instead of the MySQL Client Library. memory_get_usage does not measure the memory consumption of the MySQL Client Library at all because the MySQL Client Library does not use PHP internal memory management functions monitored by the function!
unbuffered_sets	Connection	Number of unbuffered result sets returned by normal (non prepared statement) queries.	Examples of API calls that will not buffer result sets on the client: mysqli_use_result
ps_buffered_sets	Connection	Number of buffered result sets returned by prepared statements. By default prepared statements are unbuffered.	Examples of API calls that will buffer result sets on the client: mysqli_stmt_store_result
ps_unbuffered_sets	Connection	Number of unbuffered result sets returned by prepared statements.	By default prepared statements are unbuffered.

Statistic	Scope	Description	Notes
`flushed_normal_sets`	Connection	Number of result sets from normal (non prepared statement) queries with unread data which have been flushed silently for you. Flushing happens only with unbuffered result sets.	Unbuffered result sets must be fetched completely before a new query can be run on the connection otherwise MySQL will throw an error. If the application does not fetch all rows from an unbuffered result set, mysqlnd does implicitly fetch the result set to clear the line. See also `rows_skipped_normal`, `rows_skipped_ps`. Some possible causes for an implicit flush: • Faulty client application • Client stopped reading after it found what it was looking for but has made MySQL calculate more records than needed • Client application has stopped unexpectedly
`flushed_ps_sets`	Connection	Number of result sets from prepared statements with unread data which have been flushed silently for you. Flushing happens only with unbuffered result sets.	Unbuffered result sets must be fetched completely before a new query can be run on the connection otherwise MySQL will throw an error. If the application does not fetch all rows from an unbuffered result set, mysqlnd does implicitly fetch the result set to clear the line. See also `rows_skipped_normal`, `rows_skipped_ps`. Some possible causes for an implicit flush: • Faulty client application • Client stopped reading after it found what it was looking for but has made MySQL calculate more records than needed • Client application has stopped unexpectedly
`ps_prepared_never_executed`	Connection	Number of statements prepared but never executed.	Prepared statements occupy server resources. You should not prepare a statement if you do not plan to execute it.
`ps_prepared_once_executed`	Connection	Number of prepared statements executed only one.	One of the ideas behind prepared statements is that the same query gets executed over and over again (with different parameters) and some parsing and other preparation work can be saved, if statement execution is split up in separate prepare and execute stages. The idea is to prepare once and “cache” results, for example, the

Statistic	Scope	Description	Notes
			parse tree to be reused during multiple statement executions. If you execute a prepared statement only once the two stage processing can be inefficient compared to “normal” queries because all the caching means extra work and it takes (limited) server resources to hold the cached information. Consequently, prepared statements that are executed only once may cause performance hurts.
`rows_fetched_from_server_normal`, `rows_fetched_from_server_ps`	Connection	Total number of result set rows successfully fetched from MySQL regardless if the client application has consumed them or not. Some of the rows may not have been fetched by the client application but have been flushed implicitly.	See also `packets_received_rset_row`
`rows_buffered_from_client_normal`, `rows_buffered_from_client_ps`	Connection	Total number of successfully buffered rows originating from a "normal" query or a prepared statement. This is the number of rows that have been fetched from MySQL and buffered on client. Note that there are two distinct statistics on rows that have been buffered (MySQL to mysqlnd internal buffer) and buffered rows that have been fetched by the client application (mysqlnd internal buffer to client application). If the number of buffered rows is higher than the number of fetched buffered rows it can mean that the client application runs queries that cause larger result sets than needed resulting in rows not read by the client.	Examples of queries that will buffer results: `mysqli_query`, `mysqli_store_result`
`rows_fetched_from_client_normal_buffered`, `rows_fetched_from_client_ps_buffered`	Connection	Total number of rows fetched by the client from a buffered result set created by a normal query or a prepared statement.	
`rows_fetched_from_client_normal_unbuffered`, `rows_fetched_from_client_ps_unbuffered`	Connection	Total number of rows fetched by the client from a unbuffered result set created by a "normal" query or a prepared statement.	
`rows_fetched_from_client_ps_cursor`	Connection	Total number of rows fetch by the client from a cursor created by a prepared statement.	
`rows_skipped_normal`, `rows_skipped_ps`	Connection	Reserved for future use (currently not supported)	
`copy_on_write_saved`, `copy_on_write_performed`	Process	With mysqlnd, variables returned by the extensions point into mysqlnd internal network result buffers. If you do not	

Statistic	Scope	Description	Notes
		change the variables, fetched data will be kept only once in memory. If you change the variables, mysqlnd has to perform a copy-on-write to protect the internal network result buffers from being changed. With the MySQL Client Library you always hold fetched data twice in memory. Once in the internal MySQL Client Library buffers and once in the variables returned by the extensions. In theory mysqlnd can save up to 40% memory. However, note that the memory saving cannot be measured using `memory_get_usage`.	
`explicit_free_result`, `implicit_free_result`	Connection, Process (only during prepared statement cleanup)	Total number of freed result sets.	The free is always considered explicit but for result sets created by an init command, for example, `mysqli_options(MYSQLI_INIT_COMMAND ,`
`proto_text_fetched_null`, `proto_text_fetched_bit`, `proto_text_fetched_tinyint`, `proto_text_fetched_short`, `proto_text_fetched_int24`, `proto_text_fetched_int`, `proto_text_fetched_bigint`, `proto_text_fetched_decimal`, `proto_text_fetched_float`, `proto_text_fetched_double`, `proto_text_fetched_date`, `proto_text_fetched_year`, `proto_text_fetched_time`, `proto_text_fetched_datetime`, `proto_text_fetched_timestamp`, `proto_text_fetched_string`, `proto_text_fetched_blob`, `proto_text_fetched_enum`, `proto_text_fetched_set`, `proto_text_fetched_geometry`, `proto_text_fetched_other`	Connection	Total number of columns of a certain type fetched from a normal query (MySQL text protocol)	Mapping from C API / MySQL meta data type to statistics name: • `MYSQL_TYPE_NULL` - proto_text_fetched_null • `MYSQL_TYPE_BIT` - proto_text_fetched_bit • `MYSQL_TYPE_TINY` - proto_text_fetched_tinyint • `MYSQL_TYPE_SHORT` - proto_text_fetched_short • `MYSQL_TYPE_INT24` - proto_text_fetched_int24 • `MYSQL_TYPE_LONG` - proto_text_fetched_int • `MYSQL_TYPE_LONGLONG` - proto_text_fetched_bigint • `MYSQL_TYPE_DECIMAL`, `MYSQL_TYPE_NEWDECIMAL` - proto_text_fetched_decimal • `MYSQL_TYPE_FLOAT` - proto_text_fetched_float

Statistic	Scope	Description	Notes
			• MYSQL_TYPE_DOUBLE - proto_text_fetched_double • MYSQL_TYPE_DATE, MYSQL_TYPE_NEWDATE - proto_text_fetched_date • MYSQL_TYPE_YEAR - proto_text_fetched_year • MYSQL_TYPE_TIME - proto_text_fetched_time • MYSQL_TYPE_DATETIME - proto_text_fetched_datetime • MYSQL_TYPE_TIMESTAMP - proto_text_fetched_timestamp • MYSQL_TYPE_STRING, MYSQL_TYPE_VARSTRING, MYSQL_TYPE_VARCHAR - proto_text_fetched_string • MYSQL_TYPE_TINY_BLOB, MYSQL_TYPE_MEDIUM_BLOB, MYSQL_TYPE_LONG_BLOB, MYSQL_TYPE_BLOB - proto_text_fetched_blob • MYSQL_TYPE_ENUM - proto_text_fetched_enum • MYSQL_TYPE_SET - proto_text_fetched_set • MYSQL_TYPE_GEOMETRY - proto_text_fetched_geometry • Any MYSQL_TYPE_* not listed before (there should be none) - proto_text_fetched_other Note that the MYSQL_*-type constants may not be associated with the very same SQL column types in every version of MySQL.
proto_binary_fetched_null, proto_binary_fetched_bit, proto_binary_fetched_tiny, proto_binary_fetched_short, proto_binary_fetched_int24, proto_binary_fetched_int, proto_binary_fetched_bigint,	Connection	Total number of columns of a certain type fetched from a prepared statement (MySQL binary protocol).	For type mapping see proto_text_* described in the preceding text.

Statistic	Scope	Description	Notes
proto_binary_fetched_decimal, proto_binary_fetched_float, proto_binary_fetched_double, proto_binary_fetched_date, proto_binary_fetched_year, proto_binary_fetched_time, proto_binary_fetched_datetime, proto_binary_fetched_timestamp, proto_binary_fetched_string, proto_binary_fetched_blob, proto_binary_fetched_enum, proto_binary_fetched_set, proto_binary_fetched_geometry, proto_binary_fetched_other			

Table 6.5 Returned mysqlnd statistics: Connection

Statistic	Scope	Description	Notes
connect_success, connect_failure	Connection	Total number of successful / failed connection attempt.	Reused connections and all other kinds of connections are included.
reconnect	Process	Total number of (real_)connect attempts made on an already opened connection handle.	The code sequence `$link = new mysqli(...); $link->real_connect(...)` will cause a reconnect. But `$link = new mysqli(...); $link->connect(...)` will not because `$link->connect(...)` will explicitly close the existing connection before a new connection is established.
pconnect_success	Connection	Total number of successful persistent connection attempts.	Note that `connect_success` holds the sum of successful persistent and non-persistent connection attempts. The number of successful non-persistent connection attempts is `connect_success` - `pconnect_success`.
active_connections	Connection	Total number of active persistent and non-persistent connections.	
active_persistent_connections	Connection	Total number of active persistent connections.	The total number of active non-persistent connections is `active_connections` - `active_persistent_connections`.

explicit_close	Connection	Total number of explicitly closed connections (ext/mysqli only).	Examples of code snippets that cause an explicit close :
			`$link = new mysqli(...); $link->close(...)`
			`$link = new mysqli(...); $link->connect(...)`

implicit_close	Connection	Total number of implicitly closed connections (ext/mysqli only).	Examples of code snippets that cause an implicit close :

Statistic	Scope	Description	Notes
			• `$link = new mysqli(...); $link->real_connect(...)` • `unset($link)` • Persistent connection: pooled connection has been created with real_connect and there may be unknown options set - close implicitly to avoid returning a connection with unknown options • Persistent connection: ping/change_user fails and ext/mysqli closes the connection • end of script execution: close connections that have not been closed by the user
`disconnect_close`	Connection	Connection failures indicated by the C API call `mysql_real_connect` during an attempt to establish a connection.	It is called `disconnect_close` because the connection handle passed to the C API call will be closed.
`in_middle_of_command_close`	Process	A connection has been closed in the middle of a command execution (outstanding result sets not fetched, after sending a query and before retrieving an answer, while fetching data, while transferring data with LOAD DATA).	Unless you use asynchronous queries this should only happen if your script stops unexpectedly and PHP shuts down the connections for you.
`init_command_executed_count`	Connection	Total number of init command executions, for example, `mysqli_options(MYSQLI_INIT_COMMAND, ...)`.	The number of successful executions is `init_command_executed_count` - `init_command_failed_count`.
`init_command_failed_count`	Connection	Total number of failed init commands.	

Table 6.6 Returned mysqlnd statistics: COM_* Command

Statistic	Scope	Description	Notes
`com_quit`, `com_init_db`, `com_query`, `com_field_list`, `com_create_db`, `com_drop_db`, `com_refresh`, `com_shutdown`, `com_statistics`, `com_process_info`, `com_connect`, `com_process_kill`, `com_debug`, `com_ping`, `com_time`, `com_delayed_insert`,	Connection	Total number of attempts to send a certain COM_* command from PHP to MySQL.	The statistics are incremented after checking the line and immediately before sending the corresponding MySQL client server protocol packet. If mysqlnd fails to send the packet over the wire the statistics will not be decremented. In case of a failure mysqlnd emits a PHP warning “Error while sending %s packet. PID=%d.” Usage examples: • Check if PHP sends certain commands to MySQL, for example, check if a client sends `COM_PROCESS_KILL`

Statistic	Scope	Description	Notes
com_change_user, com_binlog_dump, com_table_dump, com_connect_out, com_register_slave, com_stmt_prepare, com_stmt_execute, com_stmt_send_long_data, com_stmt_close, com_stmt_reset, com_stmt_set_option, com_stmt_fetch, com_daemon			• Calculate the average number of prepared statement executions by comparing COM_EXECUTE with COM_PREPARE • Check if PHP has run any non-prepared SQL statements by checking if COM_QUERY is zero • Identify PHP scripts that run an excessive number of SQL statements by checking COM_QUERY and COM_EXECUTE

Miscellaneous

Table 6.7 Returned mysqlnd statistics: Miscellaneous

Statistic	Scope	Description	Notes
explicit_stmt_close, implicit_stmt_close	Process	Total number of close prepared statements.	A close is always considered explicit but for a failed prepare.
mem_emalloc_count, mem_emalloc_ammount, mem_ecalloc_count, mem_ecalloc_ammount, mem_erealloc_count, mem_erealloc_ammount, mem_efree_count, mem_malloc_count, mem_malloc_ammount, mem_calloc_count, mem_calloc_ammount, mem_realloc_count, mem_realloc_ammount, mem_free_count	Process	Memory management calls.	Development only.
command_buffer_too_small	Connection	Number of network command buffer extensions while sending commands from PHP to MySQL.	mysqlnd allocates an internal command/network buffer of mysqlnd.net_cmd_buffer_size (php.ini) bytes for every connection. If a MySQL Client Server protocol command, for example, COM_QUERY (normal query), does not fit into the buffer, mysqlnd will grow the buffer to what is needed for sending the command. Whenever the buffer gets extended for one connection command_buffer_too_small will be incremented by one. If mysqlnd has to grow the buffer beyond its initial size of mysqlnd.net_cmd_buffer_size

Statistic	Scope	Description	Notes
			(`php.ini`) bytes for almost every connection, you should consider to increase the default size to avoid re-allocations. The default buffer size is 2048 bytes in PHP 5.3.0. In future versions the default will be 4kB or larger. The default can changed either through the `php.ini` setting `mysqlnd.net_cmd_buffer_size` or using `mysqli_options(MYSQLI_OPT_NET_CMD_BUFFER_int size)`. It is recommended to set the buffer size to no less than 4096 bytes because mysqlnd also uses it when reading certain communication packet from MySQL. In PHP 5.3.0, mysqlnd will not grow the buffer if MySQL sends a packet that is larger than the current size of the buffer. As a consequence mysqlnd is unable to decode the packet and the client application will get an error. There are only two situations when the packet can be larger than the 2048 bytes default of `mysqlnd.net_cmd_buffer_size` in PHP 5.3.0: the packet transports a very long error message or the packet holds column meta data from `COM_LIST_FIELD` (`mysql_list_fields`) and the meta data comes from a string column with a very long default value (>1900 bytes). No bug report on this exists - it should happen rarely. As of PHP 5.3.2 mysqlnd does not allow setting buffers smaller than 4096 bytes.
`connection_reused`			

6.7 Notes

This section provides a collection of miscellaneous notes on MySQL Native Driver usage.

- Using `mysqlnd` means using PHP streams for underlying connectivity. For `mysqlnd`, the PHP streams documentation (http://www.php.net/manual/en/book.stream) should be consulted on such details as timeout settings, not the documentation for the MySQL Client Library.

6.8 Memory management

Introduction

The MySQL Native Driver manages memory different than the MySQL Client Library. The libraries differ in the way memory is allocated and released, how memory is allocated in chunks while reading results from MySQL, which debug and development options exist, and how results read from MySQL are linked to PHP user variables.

The following notes are intended as an introduction and summary to users interested at understanding the MySQL Native Driver at the C code level.

Memory management functions used

All memory allocation and deallocation is done using the PHP memory management functions. Therefore, the memory consumption of mysqlnd can be tracked using PHP API calls, such as `memory_get_usage`. Because memory is allocated and released using the PHP memory management, the changes may not immediately become visible at the operating system level. The PHP memory management acts as a proxy which may delay releasing memory towards the system. Due to this, comparing the memory usage of the MySQL Native Driver and the MySQL Client Library is difficult. The MySQL Client Library is using the operating system memory management calls directly, hence the effects can be observed immediately at the operating system level.

Any memory limit enforced by PHP also affects the MySQL Native Driver. This may cause out of memory errors when fetching large result sets that exceed the size of the remaining memory made available by PHP. Because the MySQL Client Library is not using PHP memory management functions, it does not comply to any PHP memory limit set. If using the MySQL Client Library, depending on the deployment model, the memory footprint of the PHP process may grow beyond the PHP memory limit. But also PHP scripts may be able to process larger result sets as parts of the memory allocated to hold the result sets are beyond the control of the PHP engine.

PHP memory management functions are invoked by the MySQL Native Driver through a lightweight wrapper. Among others, the wrapper makes debugging easier.

Handling of result sets

The various MySQL Server and the various client APIs differentiate between buffered and unbuffered result sets. Unbuffered result sets are transferred row-by-row from MySQL to the client as the client iterates over the results. Buffered results are fetched in their entirety by the client library before passing them on to the client.

The MySQL Native Driver is using PHP Streams for the network communication with the MySQL Server. Results sent by MySQL are fetched from the PHP Streams network buffers into the result buffer of mysqlnd. The result buffer is made of zvals. In a second step the results are made available to the PHP script. This final transfer from the result buffer into PHP variables impacts the memory consumption and is mostly noticible when using buffered result sets.

By default the MySQL Native Driver tries to avoid holding buffered results twice in memory. Results are kept only once in the internal result buffers and their zvals. When results are fetched into PHP variables by the PHP script, the variables will reference the internal result buffers. Database query results are not copied and kept in memory only once. Should the user modify the contents of a variable holding the database results a copy-on-write must be performed to avoid changing the referenced internal result buffer. The contents of the buffer must not be modified because the user may decide to read the result set a

second time. The copy-on-write mechanism is implemented using an additional reference management list and the use of standard zval reference counters. Copy-on-write must also be done if the user reads a result set into PHP variables and frees a result set before the variables are unset.

Generally speaking, this pattern works well for scripts that read a result set once and do not modify variables holding results. Its major drawback is the memory overhead caused by the additional reference management which comes primarily from the fact that user variables holding results cannot be entirely released until the mysqlnd reference management stops referencing them. The MySQL Native driver removes the reference to the user variables when the result set is freed or a copy-on-write is performed. An observer will see the total memory consumption grow until the result set is released. Use the statistics to check whether a script does release result sets explicitly or the driver is does implicit releases and thus memory is used for a time longer than necessary. Statistics also help to see how many copy-on-write operations happened.

A PHP script reading many small rows of a buffered result set using a code snippet equal or equivalent to `while ($row = $res->fetch_assoc()) { ... }` may optimize memory consumption by requesting copies instead of references. Albeit requesting copies means keeping results twice in memory, it allows PHP to free the copy contained in `$row` as the result set is being iterated and prior to releasing the result set itself. On a loaded server optimizing peak memory usage may help improving the overall system performace although for an individual script the copy approach may be slower due to additional allocations and memory copy operations.

The copy mode can be enforced by setting mysqlnd.fetch_data_copy=1.

Monitoring and debugging

There are multiple ways of tracking the memory usage of the MySQL Native Driver. If the goal is to get a quick high level overview or to verify the memory efficiency of PHP scripts, then check the statistics collected by the library. The statistics allow you, for example, to catch SQL statements which generate more results than are processed by a PHP script.

The debug trace log can be configured to record memory management calls. This helps to see when memory is allocated or free'd. However, the size of the requested memory chunks may not be listed.

Some, recent versions of the MySQL Native Driver feature the emulation of random out of memory situations. This feature is meant to be used by the C developers of the library or mysqlnd plugin authors only. Please, search the source code for corresponding PHP configuration settings and further details. The feature is considered private and may be modified at any time without prior notice.

6.9 MySQL Native Driver Plugin API

The MySQL Native Driver Plugin API is a feature of MySQL Native Driver, or `mysqlnd`. `Mysqlnd` plugins operate in the layer between PHP applications and the MySQL server. This is comparable to MySQL Proxy. MySQL Proxy operates on a layer between any MySQL client application, for example, a PHP application and, the MySQL server. `Mysqlnd` plugins can undertake typical MySQL Proxy tasks such as load balancing, monitoring and performance optimizations. Due to the different architecture and location, `mysqlnd` plugins do not have some of MySQL Proxy's disadvantages. For example, with plugins, there is no single point of failure, no dedicated proxy server to deploy, and no new programming language to learn (Lua).

A `mysqlnd` plugin can be thought of as an extension to `mysqlnd`. Plugins can intercept the majority of `mysqlnd` functions. The `mysqlnd` functions are called by the PHP MySQL extensions such as `ext/`

mysql, ext/mysqli, and PDO_MYSQL. As a result, it is possible for a mysqlnd plugin to intercept all calls made to these extensions from the client application.

Internal mysqlnd function calls can also be intercepted, or replaced. There are no restrictions on manipulating mysqlnd internal function tables. It is possible to set things up so that when certain mysqlnd functions are called by the extensions that use mysqlnd, the call is directed to the appropriate function in the mysqlnd plugin. The ability to manipulate mysqlnd internal function tables in this way allows maximum flexibility for plugins.

Mysqlnd plugins are in fact PHP Extensions, written in C, that use the mysqlnd plugin API (which is built into MySQL Native Driver, mysqlnd). Plugins can be made 100% transparent to PHP applications. No application changes are needed because plugins operate on a different layer. The mysqlnd plugin can be thought of as operating in a layer below mysqlnd.

The following list represents some possible applications of mysqlnd plugins.

- Load Balancing
  - Read/Write Splitting. An example of this is the PECL/mysqlnd_ms (Master Slave) extension. This extension splits read/write queries for a replication setup.
  - Failover
  - Round-Robin, least loaded
- Monitoring
  - Query Logging
  - Query Analysis
  - Query Auditing. An example of this is the PECL/mysqlnd_sip (SQL Injection Protection) extension. This extension inspects queries and executes only those that are allowed according to a ruleset.
- Performance
  - Caching. An example of this is the PECL/mysqlnd_qc (Query Cache) extension.
  - Throttling
  - Sharding. An example of this is the PECL/mysqlnd_mc (Multi Connect) extension. This extension will attempt to split a SELECT statement into n-parts, using SELECT ... LIMIT part_1, SELECT LIMIT part_n. It sends the queries to distinct MySQL servers and merges the result at the client.

MySQL Native Driver Plugins Available

There are a number of mysqlnd plugins already available. These include:

- *PECL/mysqlnd_mc* - Multi Connect plugin.
- *PECL/mysqlnd_ms* - Master Slave plugin.
- *PECL/mysqlnd_qc* - Query Cache plugin.
- *PECL/mysqlnd_pscache* - Prepared Statement Handle Cache plugin.
- *PECL/mysqlnd_sip* - SQL Injection Protection plugin.
- *PECL/mysqlnd_uh* - User Handler plugin.

6.9.1 A comparison of mysqlnd plugins with MySQL Proxy

`Mysqlnd` plugins and MySQL Proxy are different technologies using different approaches. Both are valid tools for solving a variety of common tasks such as load balancing, monitoring, and performance enhancements. An important difference is that MySQL Proxy works with all MySQL clients, whereas `mysqlnd` plugins are specific to PHP applications.

As a PHP Extension, a `mysqlnd` plugin gets installed on the PHP application server, along with the rest of PHP. MySQL Proxy can either be run on the PHP application server or can be installed on a dedicated machine to handle multiple PHP application servers.

Deploying MySQL Proxy on the application server has two advantages:

1. No single point of failure
2. Easy to scale out (horizontal scale out, scale by client)

MySQL Proxy (and `mysqlnd` plugins) can solve problems easily which otherwise would have required changes to existing applications.

However, MySQL Proxy does have some disadvantages:

- MySQL Proxy is a new component and technology to master and deploy.
- MySQL Proxy requires knowledge of the Lua scripting language.

MySQL Proxy can be customized with C and Lua programming. Lua is the preferred scripting language of MySQL Proxy. For most PHP experts Lua is a new language to learn. A `mysqlnd` plugin can be written in C. It is also possible to write plugins in PHP using PECL/mysqlnd_uh.

MySQL Proxy runs as a daemon - a background process. MySQL Proxy can recall earlier decisions, as all state can be retained. However, a `mysqlnd` plugin is bound to the request-based lifecycle of PHP. MySQL Proxy can also share one-time computed results among multiple application servers. A `mysqlnd` plugin would need to store data in a persistent medium to be able to do this. Another daemon would need to be used for this purpose, such as Memcache. This gives MySQL Proxy an advantage in this case.

MySQL Proxy works on top of the wire protocol. With MySQL Proxy you have to parse and reverse engineer the MySQL Client Server Protocol. Actions are limited to those that can be achieved by manipulating the communication protocol. If the wire protocol changes (which happens very rarely) MySQL Proxy scripts would need to be changed as well.

`Mysqlnd` plugins work on top of the C API, which mirrors the `libmysqlclient` client and Connector/C APIs. This C API is basically a wrapper around the MySQL Client Server protocol, or wire protocol, as it is sometimes called. You can intercept all C API calls. PHP makes use of the C API, therefore you can hook all PHP calls, without the need to program at the level of the wire protocol.

`Mysqlnd` implements the wire protocol. Plugins can therefore parse, reverse engineer, manipulate and even replace the communication protocol. However, this is usually not required.

As plugins allow you to create implementations that use two levels (C API and wire protocol), they have greater flexibility than MySQL Proxy. If a `mysqlnd` plugin is implemented using the C API, any subsequent changes to the wire protocol do not require changes to the plugin itself.

6.9.2 Obtaining the mysqlnd plugin API

The mysqlnd plugin API is simply part of the MySQL Native Driver PHP extension, ext/mysqlnd. Development started on the mysqlnd plugin API in December 2009. It is developed as part of the PHP source repository, and as such is available to the public either via Git, or through source snapshot downloads.

The following table shows PHP versions and the corresponding mysqlnd version contained within.

Table 6.8 The bundled mysqlnd version per PHP release

PHP Version	MySQL Native Driver version
5.3.0	5.0.5
5.3.1	5.0.5
5.3.2	5.0.7
5.3.3	5.0.7
5.3.4	5.0.7

Plugin developers can determine the mysqlnd version through accessing MYSQLND_VERSION, which is a string of the format “mysqlnd 5.0.7-dev - 091210 - $Revision: 300535”, or through MYSQLND_VERSION_ID, which is an integer such as 50007. Developers can calculate the version number as follows:

Table 6.9 MYSQLND_VERSION_ID calculation table

Version (part)	Example
Major*10000	5*10000 = 50000
Minor*100	0*100 = 0
Patch	7 = 7
MYSQLND_VERSION_ID	50007

During development, developers should refer to the mysqlnd version number for compatibility and version tests, as several iterations of mysqlnd could occur during the lifetime of a PHP development branch with a single PHP version number.

6.9.3 MySQL Native Driver Plugin Architecture

This section provides an overview of the mysqlnd plugin architecture.

MySQL Native Driver Overview

Before developing mysqlnd plugins, it is useful to know a little of how mysqlnd itself is organized. Mysqlnd consists of the following modules:

Table 6.10 The mysqlnd organization chart, per module

Modules Statistics	mysqlnd_statistics.c
Connection	mysqlnd.c
Resultset	mysqlnd_result.c
Resultset Metadata	mysqlnd_result_meta.c
Statement	mysqlnd_ps.c

Network	mysqlnd_net.c
Wire protocol	mysqlnd_wireprotocol.c

C Object Oriented Paradigm

At the code level, `mysqlnd` uses a C pattern for implementing object orientation.

In C you use a `struct` to represent an object. Members of the struct represent object properties. Struct members pointing to functions represent methods.

Unlike with other languages such as C++ or Java, there are no fixed rules on inheritance in the C object oriented paradigm. However, there are some conventions that need to be followed that will be discussed later.

The PHP Life Cycle

When considering the PHP life cycle there are two basic cycles:

- PHP engine startup and shutdown cycle
- Request cycle

When the PHP engine starts up it will call the module initialization (MINIT) function of each registered extension. This allows each module to setup variables and allocate resources that will exist for the lifetime of the PHP engine process. When the PHP engine shuts down it will call the module shutdown (MSHUTDOWN) function of each extension.

During the lifetime of the PHP engine it will receive a number of requests. Each request constitutes another life cycle. On each request the PHP engine will call the request initialization function of each extension. The extension can perform any variable setup and resource allocation required for request processing. As the request cycle ends the engine calls the request shutdown (RSHUTDOWN) function of each extension so the extension can perform any cleanup required.

How a plugin works

A `mysqlnd` plugin works by intercepting calls made to `mysqlnd` by extensions that use `mysqlnd`. This is achieved by obtaining the `mysqlnd` function table, backing it up, and replacing it by a custom function table, which calls the functions of the plugin as required.

The following code shows how the `mysqlnd` function table is replaced:

```
/* a place to store original function table */
struct st_mysqlnd_conn_methods org_methods;

void minit_register_hooks(TSRMLS_D) {
  /* active function table */
  struct st_mysqlnd_conn_methods * current_methods
    = mysqlnd_conn_get_methods();

  /* backup original function table */
  memcpy(&org_methods, current_methods,
    sizeof(struct st_mysqlnd_conn_methods);

  /* install new methods */
  current_methods->query = MYSQLND_METHOD(my_conn_class, query);
}
```

Connection function table manipulations must be done during Module Initialization (MINIT). The function table is a global shared resource. In an multi-threaded environment, with a TSRM build, the manipulation of a global shared resource during the request processing will almost certainly result in conflicts.

> **Note**
>
> Do not use any fixed-size logic when manipulating the `mysqlnd` function table: new methods may be added at the end of the function table. The function table may change at any time in the future.

Calling parent methods

If the original function table entries are backed up, it is still possible to call the original function table entries - the parent methods.

In some cases, such as for `Connection::stmt_init()`, it is vital to call the parent method prior to any other activity in the derived method.

```
MYSQLND_METHOD(my_conn_class, query)(MYSQLND *conn,
  const char *query, unsigned int query_len TSRMLS_DC) {

  php_printf("my_conn_class::query(query = %s)\n", query);

  query = "SELECT 'query rewritten' FROM DUAL";
  query_len = strlen(query);

  return org_methods.query(conn, query, query_len); /* return with call to parent */
}
```

Extending properties

A `mysqlnd` object is represented by a C struct. It is not possible to add a member to a C struct at run time. Users of `mysqlnd` objects cannot simply add properties to the objects.

Arbitrary data (properties) can be added to a `mysqlnd` objects using an appropriate function of the `mysqlnd_plugin_get_plugin_<object>_data()` family. When allocating an object `mysqlnd` reserves space at the end of the object to hold a `void *` pointer to arbitrary data. `mysqlnd` reserves space for one `void *` pointer per plugin.

The following table shows how to calculate the position of the pointer for a specific plugin:

Table 6.11 Pointer calculations for mysqlnd

Memory address	Contents
0	Beginning of the mysqlnd object C struct
n	End of the mysqlnd object C struct
n + (m x sizeof(void*))	void* to object data of the m-th plugin

If you plan to subclass any of the `mysqlnd` object constructors, which is allowed, you must keep this in mind!

The following code shows extending properties:

```
/* any data we want to associate */
```

```
typedef struct my_conn_properties {
  unsigned long query_counter;
} MY_CONN_PROPERTIES;

/* plugin id */
unsigned int my_plugin_id;

void minit_register_hooks(TSRMLS_D) {
  /* obtain unique plugin ID */
  my_plugin_id = mysqlnd_plugin_register();
  /* snip - see Extending Connection: methods */
}

static MY_CONN_PROPERTIES** get_conn_properties(const MYSQLND *conn TSRMLS_DC) {
  MY_CONN_PROPERTIES** props;
  props = (MY_CONN_PROPERTIES**)mysqlnd_plugin_get_plugin_connection_data(
    conn, my_plugin_id);
  if (!props || !(*props)) {
    *props = mnd_pecalloc(1, sizeof(MY_CONN_PROPERTIES), conn->persistent);
    (*props)->query_counter = 0;
  }
  return props;
}
```

The plugin developer is responsible for the management of plugin data memory.

Use of the `mysqlnd` memory allocator is recommended for plugin data. These functions are named using the convention: `mnd_*loc()`. The `mysqlnd` allocator has some useful features, such as the ability to use a debug allocator in a non-debug build.

Table 6.12 When and how to subclass

	When to subclass?	Each instance has its own private function table?	How to subclass?
Connection (MYSQLND)	MINIT	No	mysqlnd_conn_get_methods()
Resultset (MYSQLND_RES)	MINIT or later	Yes	mysqlnd_result_get_methods() or object method function table manipulation
Resultset Meta (MYSQLND_RES_METADATA)	MINIT	No	mysqlnd_result_metadata_get_method
Statement (MYSQLND_STMT)	MINIT	No	mysqlnd_stmt_get_methods()
Network (MYSQLND_NET)	MINIT or later	Yes	mysqlnd_net_get_methods() or object method function table manipulation
Wire protocol (MYSQLND_PROTOCOL)	MINIT or later	Yes	mysqlnd_protocol_get_methods() or object method function table manipulation

You must not manipulate function tables at any time later than MINIT if it is not allowed according to the above table.

Some classes contain a pointer to the method function table. All instances of such a class will share the same function table. To avoid chaos, in particular in threaded environments, such function tables must only be manipulated during MINIT.

Other classes use copies of a globally shared function table. The class function table copy is created together with the object. Each object uses its own function table. This gives you two options: you can manipulate the default function table of an object at MINIT, and you can additionally refine methods of an object without impacting other instances of the same class.

The advantage of the shared function table approach is performance. There is no need to copy a function table for each and every object.

Table 6.13 Constructor status

	Allocation, construction, reset	Can be modified?	Caller
Connection (MYSQLND)	mysqlnd_init()	No	mysqlnd_connect()
Resultset(MYSQLND_RES)	Allocation: • Connection::result_init() Reset and re-initialized during: • Result::use_result() • Result::store_result	Yes, but call parent!	• Connection::list_fields() • Statement::get_result() • Statement::prepare() (Metadata only) • Statement::resultMetaData()
Resultset Meta (MYSQLND_RES_METADATA)	Connection::result_meta_init()	Yes, but call parent!	Result::read_result_metadata()
Statement (MYSQLND_STMT)	Connection::stmt_init()	Yes, but call parent!	Connection::stmt_init()
Network (MYSQLND_NET)	mysqlnd_net_init()	No	Connection::init()
Wire protocol (MYSQLND_PROTOCOL)	mysqlnd_protocol_init()	No	Connection::init()

It is strongly recommended that you do not entirely replace a constructor. The constructors perform memory allocations. The memory allocations are vital for the `mysqlnd` plugin API and the object logic of `mysqlnd`. If you do not care about warnings and insist on hooking the constructors, you should at least call the parent constructor before doing anything in your constructor.

Regardless of all warnings, it can be useful to subclass constructors. Constructors are the perfect place for modifying the function tables of objects with non-shared object tables, such as Resultset, Network, Wire Protocol.

Table 6.14 Destruction status

	Derived method must call parent?	Destructor
Connection	yes, after method execution	free_contents(), end_psession()
Resultset	yes, after method execution	free_result()
Resultset Meta	yes, after method execution	free()
Statement	yes, after method execution	dtor(), free_stmt_content()
Network	yes, after method execution	free()
Wire protocol	yes, after method execution	free()

The destructors are the appropriate place to free properties, `mysqlnd_plugin_get_plugin_<object>_data()`.

The listed destructors may not be equivalent to the actual `mysqlnd` method freeing the object itself. However, they are the best possible place for you to hook in and free your plugin data. As with constructors you may replace the methods entirely but this is not recommended. If multiple methods are listed in the above table you will need to hook all of the listed methods and free your plugin data in whichever method is called first by `mysqlnd`.

The recommended method for plugins is to simply hook the methods, free your memory and call the parent implementation immediately following this.

> **Caution**
>
> Due to a bug in PHP versions 5.3.0 to 5.3.3, plugins do not associate plugin data with a persistent connection. This is because `ext/mysql` and `ext/mysqli` do not trigger all the necessary `mysqlnd end_psession()` method calls and the plugin may therefore leak memory. This has been fixed in PHP 5.3.4.

6.9.4 The mysqlnd plugin API

The following is a list of functions provided in the `mysqlnd` plugin API:

- mysqlnd_plugin_register()
- mysqlnd_plugin_count()
- mysqlnd_plugin_get_plugin_connection_data()
- mysqlnd_plugin_get_plugin_result_data()
- mysqlnd_plugin_get_plugin_stmt_data()
- mysqlnd_plugin_get_plugin_net_data()
- mysqlnd_plugin_get_plugin_protocol_data()
- mysqlnd_conn_get_methods()
- mysqlnd_result_get_methods()
- mysqlnd_result_meta_get_methods()
- mysqlnd_stmt_get_methods()
- mysqlnd_net_get_methods()
- mysqlnd_protocol_get_methods()

There is no formal definition of what a plugin is and how a plugin mechanism works.

Components often found in plugins mechanisms are:

- A plugin manager
- A plugin API

- Application services (or modules)
- Application service APIs (or module APIs)

The `mysqlnd` plugin concept employs these features, and additionally enjoys an open architecture.

No Restrictions

A plugin has full access to the inner workings of `mysqlnd`. There are no security limits or restrictions. Everything can be overwritten to implement friendly or hostile algorithms. It is recommended you only deploy plugins from a trusted source.

As discussed previously, plugins can use pointers freely. These pointers are not restricted in any way, and can point into another plugin's data. Simple offset arithmetic can be used to read another plugin's data.

It is recommended that you write cooperative plugins, and that you always call the parent method. The plugins should always cooperate with `mysqlnd` itself.

Table 6.15 Issues: an example of chaining and cooperation

Extension	mysqlnd.query() pointer	call stack if calling parent
ext/mysqlnd	mysqlnd.query()	mysqlnd.query
ext/mysqlnd_cache	mysqlnd_cache.query()	1. mysqlnd_cache.query() 2. mysqlnd.query
ext/mysqlnd_monitor	mysqlnd_monitor.query()	1. mysqlnd_monitor.query() 2. mysqlnd_cache.query() 3. mysqlnd.query

In this scenario, a cache (`ext/mysqlnd_cache`) and a monitor (`ext/mysqlnd_monitor`) plugin are loaded. Both subclass `Connection::query()`. Plugin registration happens at `MINIT` using the logic shown previously. PHP calls extensions in alphabetical order by default. Plugins are not aware of each other and do not set extension dependencies.

By default the plugins call the parent implementation of the query method in their derived version of the method.

PHP Extension Recap

This is a recap of what happens when using an example plugin, `ext/mysqlnd_plugin`, which exposes the `mysqlnd` C plugin API to PHP:

- Any PHP MySQL application tries to establish a connection to 192.168.2.29
- The PHP application will either use `ext/mysql`, `ext/mysqli` or `PDO_MYSQL`. All three PHP MySQL extensions use `mysqlnd` to establish the connection to 192.168.2.29.
- `Mysqlnd` calls its connect method, which has been subclassed by `ext/mysqlnd_plugin`.
- `ext/mysqlnd_plugin` calls the userspace hook `proxy::connect()` registered by the user.
- The userspace hook changes the connection host IP from 192.168.2.29 to 127.0.0.1 and returns the connection established by `parent::connect()`.
- `ext/mysqlnd_plugin` performs the equivalent of `parent::connect(127.0.0.1)` by calling the original `mysqlnd` method for establishing a connection.

- `ext/mysqlnd` establishes a connection and returns to `ext/mysqlnd_plugin`. `ext/mysqlnd_plugin` returns as well.
- Whatever PHP MySQL extension had been used by the application, it receives a connection to 127.0.0.1. The PHP MySQL extension itself returns to the PHP application. The circle is closed.

6.9.5 Getting started building a mysqlnd plugin

It is important to remember that a `mysqlnd` plugin is itself a PHP extension.

The following code shows the basic structure of the MINIT function that will be used in the typical `mysqlnd` plugin:

```
/* my_php_mysqlnd_plugin.c */

 static PHP_MINIT_FUNCTION(mysqlnd_plugin) {
  /* globals, ini entries, resources, classes */

  /* register mysqlnd plugin */
  mysqlnd_plugin_id = mysqlnd_plugin_register();

  conn_m = mysqlnd_get_conn_methods();
  memcpy(org_conn_m, conn_m,
    sizeof(struct st_mysqlnd_conn_methods));

  conn_m->query = MYSQLND_METHOD(mysqlnd_plugin_conn, query);
  conn_m->connect = MYSQLND_METHOD(mysqlnd_plugin_conn, connect);
}
```

```
/* my_mysqlnd_plugin.c */

 enum_func_status MYSQLND_METHOD(mysqlnd_plugin_conn, query)(/* ... */) {
  /* ... */
}
enum_func_status MYSQLND_METHOD(mysqlnd_plugin_conn, connect)(/* ... */) {
  /* ... */
}
```

Task analysis: from C to userspace

```
 class proxy extends mysqlnd_plugin_connection {
  public function connect($host, ...) { .. }
}
mysqlnd_plugin_set_conn_proxy(new proxy());
```

Process:

1. PHP: user registers plugin callback
2. PHP: user calls any PHP MySQL API to connect to MySQL
3. C: ext/*mysql* calls mysqlnd method

4. C: mysqlnd ends up in ext/mysqlnd_plugin
5. C: ext/mysqlnd_plugin
    a. Calls userspace callback
    b. Or original `mysqlnd` method, if userspace callback not set

You need to carry out the following:

1. Write a class "mysqlnd_plugin_connection" in C
2. Accept and register proxy object through "mysqlnd_plugin_set_conn_proxy()"
3. Call userspace proxy methods from C (optimization - zend_interfaces.h)

Userspace object methods can either be called using `call_user_function()` or you can operate at a level closer to the Zend Engine and use `zend_call_method()`.

Optimization: calling methods from C using zend_call_method

The following code snippet shows the prototype for the `zend_call_method` function, taken from `zend_interfaces.h`.

```
 ZEND_API zval* zend_call_method(
  zval **object_pp, zend_class_entry *obj_ce,
  zend_function **fn_proxy, char *function_name,
  int function_name_len, zval **retval_ptr_ptr,
  int param_count, zval* arg1, zval* arg2 TSRMLS_DC
);
```

Zend API supports only two arguments. You may need more, for example:

```
 enum_func_status (*func_mysqlnd_conn__connect)(
  MYSQLND *conn, const char *host,
  const char * user, const char * passwd,
  unsigned int passwd_len, const char * db,
  unsigned int db_len, unsigned int port,
  const char * socket, unsigned int mysql_flags TSRMLS_DC
);
```

To get around this problem you will need to make a copy of `zend_call_method()` and add a facility for additional parameters. You can do this by creating a set of `MY_ZEND_CALL_METHOD_WRAPPER` macros.

Calling PHP userspace

This code snippet shows the optimized method for calling a userspace function from C:

```
/* my_mysqlnd_plugin.c */

MYSQLND_METHOD(my_conn_class,connect)(
  MYSQLND *conn, const char *host /* ... */ TSRMLS_DC) {
  enum_func_status ret = FAIL;
  zval * global_user_conn_proxy = fetch_userspace_proxy();
```

```
  if (global_user_conn_proxy) {
    /* call userspace proxy */
    ret = MY_ZEND_CALL_METHOD_WRAPPER(global_user_conn_proxy, host, /*...*/);
  } else {
    /* or original mysqlnd method = do nothing, be transparent */
    ret = org_methods.connect(conn, host, user, passwd,
          passwd_len, db, db_len, port,
          socket, mysql_flags TSRMLS_CC);
  }
  return ret;
}
```

Calling userspace: simple arguments

```

/* my_mysqlnd_plugin.c */

 MYSQLND_METHOD(my_conn_class,connect)(
  /* ... */, const char *host, /* ...*/) {
  /* ... */
  if (global_user_conn_proxy) {
    /* ... */
    zval* zv_host;
    MAKE_STD_ZVAL(zv_host);
    ZVAL_STRING(zv_host, host, 1);
    MY_ZEND_CALL_METHOD_WRAPPER(global_user_conn_proxy, zv_retval, zv_host /*, ...*/);
    zval_ptr_dtor(&zv_host);
    /* ... */
  }
  /* ... */
}
```

Calling userspace: structs as arguments

```

/* my_mysqlnd_plugin.c */

MYSQLND_METHOD(my_conn_class, connect)(
  MYSQLND *conn, /* ...*/) {
  /* ... */
  if (global_user_conn_proxy) {
    /* ... */
    zval* zv_conn;
    ZEND_REGISTER_RESOURCE(zv_conn, (void *)conn, le_mysqlnd_plugin_conn);
    MY_ZEND_CALL_METHOD_WRAPPER(global_user_conn_proxy, zv_retval, zv_conn, zv_host /*, ...*/);
    zval_ptr_dtor(&zv_conn);
    /* ... */
  }
  /* ... */
}
```

The first argument of many `mysqlnd` methods is a C "object". For example, the first argument of the connect() method is a pointer to `MYSQLND`. The struct MYSQLND represents a `mysqlnd` connection object.

The `mysqlnd` connection object pointer can be compared to a standard I/O file handle. Like a standard I/O file handle a `mysqlnd` connection object shall be linked to the userspace using the PHP resource variable type.

From C to userspace and back

```
 class proxy extends mysqlnd_plugin_connection {
  public function connect($conn, $host, ...) {
    /* "pre" hook */
    printf("Connecting to host = '%s'\n", $host);
    debug_print_backtrace();
    return parent::connect($conn);
  }

  public function query($conn, $query) {
    /* "post" hook */
    $ret = parent::query($conn, $query);
    printf("Query = '%s'\n", $query);
    return $ret;
  }
}
mysqlnd_plugin_set_conn_proxy(new proxy());
```

PHP users must be able to call the parent implementation of an overwritten method.

As a result of subclassing it is possible to refine only selected methods and you can choose to have "pre" or "post" hooks.

Buildin class: mysqlnd_plugin_connection::connect()

```
/*  my_mysqlnd_plugin_classes.c */

 PHP_METHOD("mysqlnd_plugin_connection", connect) {
  /* ... simplified! ... */
  zval* mysqlnd_rsrc;
  MYSQLND* conn;
  char* host; int host_len;
  if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "rs",
    &mysqlnd_rsrc, &host, &host_len) == FAILURE) {
    RETURN_NULL();
  }
  ZEND_FETCH_RESOURCE(conn, MYSQLND* conn, &mysqlnd_rsrc, -1,
    "Mysqlnd Connection", le_mysqlnd_plugin_conn);
  if (PASS == org_methods.connect(conn, host, /* simplified! */ TSRMLS_CC))
    RETVAL_TRUE;
  else
    RETVAL_FALSE;
}
```

Chapter 7 Mysqlnd replication and load balancing plugin

Table of Contents

The mysqlnd replication and load balancing plugin (mysqlnd_ms) adds easy to use MySQL replication support to all PHP MySQL extensions that use mysqlnd.

As of version PHP 5.3.3 the MySQL native driver for PHP (mysqlnd) features an internal plugin C API. C plugins, such as the replication and load balancing plugin, can extend the functionality of mysqlnd.

The MySQL native driver for PHP is a C library that ships together with PHP as of PHP 5.3.0. It serves as a drop-in replacement for the MySQL Client Library (libmysqlclient). Using mysqlnd has several advantages: no extra downloads are required because it's bundled with PHP, it's under the PHP license, there is lower memory consumption in certain cases, and it contains new functionality such as asynchronous queries.

Mysqlnd plugins like mysqlnd_ms operate, for the most part, transparently from a user perspective. The replication and load balancing plugin supports all PHP applications, and all MySQL PHP extensions. It does not change existing APIs. Therefore, it can easily be used with existing PHP applications.

7.1 Key Features

The key features of PECL/mysqlnd_ms are as follows.

- Transparent and therefore easy to use.
  - Supports all of the PHP MySQL extensions.
  - SSL support.
  - A consistent API.
  - Little to no application changes required, dependent on the required usage scenario.
  - Lazy connections: connections to master and slave servers are not opened before a SQL statement is executed.
  - Optional: automatic use of master after the first write in a web request, to lower the possible impact of replication lag.
- Can be used with any MySQL clustering solution.
  - MySQL Replication: Read-write splitting is done by the plugin. Primary focus of the plugin.
  - MySQL Cluster: Read-write splitting can be disabled. Configuration of multiple masters possible

    - Third-party solutions: the plugin is optimized for MySQL Replication but can be used with any other kind of MySQL clustering solution.
- Featured read-write split strategies
    - Automatic detection of SELECT.
    - Supports SQL hints to overrule automatism.
    - User-defined.
    - Can be disabled for, for example, when using synchronous clusters such as MySQL Cluster.
- Featured load balancing strategies
    - Round Robin: choose a different slave in round-robin fashion for every slave request.
    - Random: choose a random slave for every slave request.
    - Random once (sticky): choose a random slave once to run all slave requests for the duration of a web request.
    - User-defined. The application can register callbacks with mysqlnd_ms.
    - PHP 5.4.0 or newer: transaction aware when using API calls only to control transactions.
    - Weighted load balancing: servers can be assigned different priorities, for example, to direct more requests to a powerful machine than to another less powerful machine. Or, to prefer nearby machines to reduce latency.
- Global transaction ID
    - Client-side emulation. Makes manual master server failover and slave promotion easier with asynchronous clusters, such as MySQL Replication.
    - Support for built-in global transaction identifier feature of MySQL 5.6.5 or newer.
    - Supports using transaction ids to identify up-to-date asynchronous slaves for reading when session consistency is required. Please, note the restrictions mentioned in the manual.
    - Throttling: optionally, the plugin can wait for a slave to become "synchronous" before continuing.
- Service and consistency levels
    - Applications can request eventual, session and strong consistency service levels for connections. Appropriate cluster nodes will be searched automatically.
    - Eventual consistent MySQL Replication slave accesses can be replaced with fast local cache accesses transparently to reduce server load.
- Partitioning and sharding
    - Servers of a replication cluster can be organized into groups. SQL hints can be used to manually direct queries to a specific group. Grouping can be used to partition (shard) the data, or to cure the issue of hotspots with updates.
    - MySQL Replication filters are supported through the table filter.

- MySQL Fabric
  - Experimental support for MySQL Fabric is included.

7.2 Limitations

The built-in read-write-split mechanism is very basic. Every query which starts with `SELECT` is considered a read request to be sent to a MySQL slave server. All other queries (such as `SHOW` statements) are considered as write requests that are sent to the MySQL master server. The build-in behavior can be overruled using SQL hints, or a user-defined callback function.

The read-write splitter is not aware of multi-statements. Multi-statements are considered as one statement. The decision of where to run the statement will be based on the beginning of the statement string. For example, if using `mysqli_multi_query` to execute the multi-statement `SELECT id FROM test ; INSERT INTO test(id) VALUES (1)`, the statement will be redirected to a slave server because it begins with `SELECT`. The `INSERT` statement, which is also part of the multi-statement, will not be redirected to a master server.

> **Note**
>
> Applications must be aware of the consequences of connection switches that are performed for load balancing purposes. Please check the documentation on connection pooling and switching, transaction handling, failover load balancing and read-write splitting.

7.3 On the name

The shortcut `mysqlnd_ms` stands for `mysqlnd master slave plugin`. The name was chosen for a quick-and-dirty proof-of-concept. In the beginning the developers did not expect to continue using the code base.

7.4 Quickstart and Examples

The mysqlnd replication load balancing plugin is easy to use. This quickstart will demo typical use-cases, and provide practical advice on getting started.

It is strongly recommended to read the reference sections in addition to the quickstart. The quickstart tries to avoid discussing theoretical concepts and limitations. Instead, it will link to the reference sections. It is safe to begin with the quickstart. However, before using the plugin in mission critical environments we urge you to read additionally the background information from the reference sections.

The focus is on using PECL mysqlnd_ms for work with an asynchronous MySQL cluster, namely MySQL replication. Generally speaking an asynchronous cluster is more difficult to use than a synchronous one. Thus, users of, for example, MySQL Cluster will find more information than needed.

7.4.1 Setup

The plugin is implemented as a PHP extension. See also the installation instructions to install the PECL/mysqlnd_ms extension.

Compile or configure the PHP MySQL extension (API) (mysqli, PDO_MYSQL, mysql) that you plan to use with support for the mysqlnd library. PECL/mysqlnd_ms is a plugin for the mysqlnd library. To use the plugin with any of the PHP MySQL extensions, the extension has to use the mysqlnd library.

Then, load the extension into PHP and activate the plugin in the PHP configuration file using the PHP configuration directive named mysqlnd_ms.enable.

Example 7.1 Enabling the plugin (php.ini)

```
mysqlnd_ms.enable=1
mysqlnd_ms.config_file=/path/to/mysqlnd_ms_plugin.ini
```

The plugin uses its own configuration file. Use the PHP configuration directive mysqlnd_ms.config_file to set the full file path to the plugin-specific configuration file. This file must be readable by PHP (e.g., the web server user). Please note, the configuration directive mysqlnd_ms.config_file superseeds mysqlnd_ms.ini_file since 1.4.0. It is a common pitfall to use the old, no longer available configuration directive.

Create a plugin-specific configuration file. Save the file to the path set by the PHP configuration directive mysqlnd_ms.config_file.

The plugins configuration file is JSON based. It is divided into one or more sections. Each section has a name, for example, `myapp`. Every section makes its own set of configuration settings.

A section must, at a minimum, list the MySQL replication master server, and set a list of slaves. The plugin supports using only one master server per section. Multi-master MySQL replication setups are not yet fully supported. Use the configuration setting master to set the hostname, and the port or socket of the MySQL master server. MySQL slave servers are configured using the slave keyword.

Example 7.2 Minimal plugin-specific configuration file (mysqlnd_ms_plugin.ini)

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": [

        ]
    }
}
```

Configuring a MySQL slave server list is required, although it may contain an empty list. It is recommended to always configure at least one slave server.

Server lists can use anonymous or non-anonymous syntax. Non-anonymous lists include alias names for the servers, such as `master_0` for the master in the above example. The quickstart uses the more verbose non-anonymous syntax.

Example 7.3 Recommended minimal plugin-specific config (mysqlnd_ms_plugin.ini)

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.2.27",
                "port": "3306"
            }
        }
    }
}
```

If there are at least two servers in total, the plugin can start to load balance and switch connections. Switching connections is not always transparent and can cause issues in certain cases. The reference sections about connection pooling and switching, transaction handling, fail over load balancing and read-write splitting all provide more details. And potential pitfalls are described later in this guide.

It is the responsibility of the application to handle potential issues caused by connection switches, by configuring a master with at least one slave server, which allows switching to work therefore related problems can be found.

The MySQL master and MySQL slave servers, which you configure, do not need to be part of MySQL replication setup. For testing purpose you can use single MySQL server and make it known to the plugin as a master and slave server as shown below. This could help you to detect many potential issues with connection switches. However, such a setup will not be prone to the issues caused by replication lag.

Example 7.4 Using one server as a master and as a slave (testing only!)

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "127.0.0.1",
                "port": "3306"
            }
        }
    }
}
```

The plugin attempts to notify you of invalid configurations. Since 1.5.0 it will throw a warning during PHP startup if the configuration file cannot be read, is empty or parsing the JSON failed. Depending on your PHP settings those errors may appear in some log files only. Further validation is done when

a connection is to be established and the configuration file is searched for valid sections. Setting mysqlnd_ms.force_config_usage may help debugging a faulty setup. Please, see also configuration file debugging notes.

7.4.2 Running statements

The plugin can be used with any PHP MySQL extension (mysqli, mysql, and PDO_MYSQL) that is compiled to use the mysqlnd library. `PECL/mysqlnd_ms` plugs into the mysqlnd library. It does not change the API or behavior of those extensions.

Whenever a connection to MySQL is being opened, the plugin compares the host parameter value of the connect call, with the section names from the plugin specific configuration file. If, for example, the plugin specific configuration file has a section `myapp` then the section should be referenced by opening a MySQL connection to the host `myapp`

Example 7.5 Plugin specific configuration file (mysqlnd_ms_plugin.ini)

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.2.27",
                "port": "3306"
            }
        }
    }
}
```

Example 7.6 Opening a load balanced connection

```
<?php
/* Load balanced following "myapp" section rules from the plugins config file */
$mysqli = new mysqli("myapp", "username", "password", "database");
$pdo = new PDO('mysql:host=myapp;dbname=database', 'username', 'password');
$mysql = mysql_connect("myapp", "username", "password");
?>
```

The connection examples above will be load balanced. The plugin will send read-only statements to the MySQL slave server with the IP `192.168.2.27` and will listen on port `3306` for the MySQL client connection. All other statements will be directed to the MySQL master server running on the host `localhost`. If on Unix like operating systems, the master on `localhost` will be accepting MySQL client connections on the Unix domain socket `/tmp/mysql.sock`, while TCP/IP is the default port on Windows. The plugin will use the user name `username` and the password `password` to connect to any of the MySQL servers listed in the section `myapp` of the plugins configuration file. Upon connect, the plugin will select `database` as the current schemata.

The username, password and schema name are taken from the connect API calls and used for all servers. In other words: you must use the same username and password for every MySQL server listed in a plugin configuration file section. The is not a general limitation. As of PECL/mysqlnd_ms 1.1.0, it is possible to set the username and password for any server in the plugins configuration file, to be used instead of the credentials passed to the API call.

The plugin does not change the API for running statements. Read-write splitting works out of the box. The following example assumes that there is no significant replication lag between the master and the slave.

Example 7.7 Executing statements

```
<?php
/* Load balanced following "myapp" section rules from the plugins config file */
$mysqli = new mysqli("myapp", "username", "password", "database");
if (mysqli_connect_errno()) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* Statements will be run on the master */
if (!$mysqli->query("DROP TABLE IF EXISTS test")) {
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}
if (!$mysqli->query("CREATE TABLE test(id INT)")) {
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}
if (!$mysqli->query("INSERT INTO test(id) VALUES (1)")) {
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}

/* read-only: statement will be run on a slave */
if (!($res = $mysqli->query("SELECT id FROM test"))) {
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
} else {
    $row = $res->fetch_assoc();
    $res->close();
    printf("Slave returns id = '%s'\n", $row['id']);
}
$mysqli->close();
?>
```

The above example will output something similar to:

```
Slave returns id = '1'
```

7.4.3 Connection state

The plugin changes the semantics of a PHP MySQL connection handle. A new connection handle represents a connection pool, instead of a single MySQL client-server network connection. The connection pool consists of a master connection, and optionally any number of slave connections.

Every connection from the connection pool has its own state. For example, SQL user variables, temporary tables and transactions are part of the state. For a complete list of items that belong to the state of a

connection, see the connection pooling and switching concepts documentation. If the plugin decides to switch connections for load balancing, the application could be given a connection which has a different state. Applications must be made aware of this.

Example 7.8 Plugin config with one slave and one master

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.2.27",
                "port": "3306"
            }
        }
    }
}
```

Example 7.9 Pitfall: connection state and SQL user variables

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* Connection 1, connection bound SQL user variable, no SELECT thus run on master */
if (!$mysqli->query("SET @myrole='master'")) {
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}

/* Connection 2, run on slave because SELECT */
if (!($res = $mysqli->query("SELECT @myrole AS _role"))) {
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
} else {
    $row = $res->fetch_assoc();
    $res->close();
    printf("@myrole = '%s'\n", $row['_role']);
}
$mysqli->close();
?>
```

The above example will output:

```
@myrole = ''
```

The example opens a load balanced connection and executes two statements. The first statement `SET @myrole='master'` does not begin with the string `SELECT`. Therefore the plugin does not recognize it

as a read-only query which shall be run on a slave. The plugin runs the statement on the connection to the master. The statement sets a SQL user variable which is bound to the master connection. The state of the master connection has been changed.

The next statement is `SELECT @myrole AS _role`. The plugin does recognize it as a read-only query and sends it to the slave. The statement is run on a connection to the slave. This second connection does not have any SQL user variables bound to it. It has a different state than the first connection to the master. The requested SQL user variable is not set. The example script prints `@myrole = ''`.

It is the responsibility of the application developer to take care of the connection state. The plugin does not monitor all connection state changing activities. Monitoring all possible cases would be a very CPU intensive task, if it could be done at all.

The pitfalls can easily be worked around using SQL hints.

7.4.4 SQL Hints

SQL hints can force a query to choose a specific server from the connection pool. It gives the plugin a hint to use a designated server, which can solve issues caused by connection switches and connection state.

SQL hints are standard compliant SQL comments. Because SQL comments are supposed to be ignored by SQL processing systems, they do not interfere with other programs such as the MySQL Server, the MySQL Proxy, or a firewall.

Three SQL hints are supported by the plugin: The `MYSQLND_MS_MASTER_SWITCH` hint makes the plugin run a statement on the master, `MYSQLND_MS_SLAVE_SWITCH` enforces the use of the slave, and `MYSQLND_MS_LAST_USED_SWITCH` will run a statement on the same server that was used for the previous statement.

The plugin scans the beginning of a statement for the existence of an SQL hint. SQL hints are only recognized if they appear at the beginning of the statement.

Example 7.10 Plugin config with one slave and one master

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.2.27",
                "port": "3306"
            }
        }
    }
}
```

Example 7.11 SQL hints to prevent connection switches

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (mysqli_connect_errno()) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* Connection 1, connection bound SQL user variable, no SELECT thus run on master */
if (!$mysqli->query("SET @myrole='master'")) {
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}

/* Connection 1, run on master because of SQL hint */
if (!($res = $mysqli->query(sprintf("/*%s*/SELECT @myrole AS _role", MYSQLND_MS_LAST_USED_SWITCH)))) {
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
} else {
    $row = $res->fetch_assoc();
    $res->close();
    printf("@myrole = '%s'\n", $row['_role']);
}
$mysqli->close();
?>
```

The above example will output:

```
@myrole = 'master'
```

In the above example, using `MYSQLND_MS_LAST_USED_SWITCH` prevents session switching from the master to a slave when running the `SELECT` statement.

SQL hints can also be used to run `SELECT` statements on the MySQL master server. This may be desired if the MySQL slave servers are typically behind the master, but you need current data from the cluster.

In version 1.2.0 the concept of a service level has been introduced to address cases when current data is required. Using a service level requires less attention and removes the need of using SQL hints for this use case. Please, find more information below in the service level and consistency section.

Example 7.12 Fighting replication lag

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* Force use of master, master has always fresh and current data */
if (!$mysqli->query(sprintf("/*%s*/SELECT critical_data FROM important_table", MYSQLND_MS_MASTER_SWITCH)))
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}
?>
```

A use case may include the creation of tables on a slave. If an SQL hint is not given, then the plugin will send `CREATE` and `INSERT` statements to the master. Use the SQL hint `MYSQLND_MS_SLAVE_SWITCH` if you want to run any such statement on a slave, for example, to build temporary reporting tables.

Example 7.13 Table creation on a slave

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* Force use of slave */
if (!$mysqli->query(sprintf("/*%s*/CREATE TABLE slave_reporting(id INT)", MYSQLND_MS_SLAVE_SWITCH))) {
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}
/* Continue using this particular slave connection */
if (!$mysqli->query(sprintf("/*%s*/INSERT INTO slave_reporting(id) VALUES (1), (2), (3)", MYSQLND_MS_LAST_USED
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}
/* Don't use MYSQLND_MS_SLAVE_SWITCH which would allow switching to another slave! */
if ($res = $mysqli->query(sprintf("/*%s*/SELECT COUNT(*) AS _num FROM slave_reporting", MYSQLND_MS_LAST_USED_S
    $row = $res->fetch_assoc();
    $res->close();
    printf("There are %d rows in the table 'slave_reporting'", $row['_num']);
} else {
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}
$mysqli->close();
?>
```

The SQL hint `MYSQLND_MS_LAST_USED` forbids switching a connection, and forces use of the previously used connection.

7.4.5 Local transactions

The current version of the plugin is not transaction safe by default, because it is not aware of running transactions in all cases. SQL transactions are units of work to be run on a single server. The plugin does not always know when the unit of work starts and when it ends. Therefore, the plugin may decide to switch connections in the middle of a transaction.

No kind of MySQL load balancer can detect transaction boundaries without any kind of hint from the application.

You can either use SQL hints to work around this limitation. Alternatively, you can activate transaction API call monitoring. In the latter case you must use API calls only to control transactions, see below.

Example 7.14 Plugin config with one slave and one master

```
[myapp]
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
```

```
        "slave": {
            "slave_0": {
                "host": "192.168.2.27",
                "port": "3306"
            }
        }
    }
}
```

Example 7.15 Using SQL hints for transactions

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* Not a SELECT, will use master */
if (!$mysqli->query("START TRANSACTION")) {
    /* Please use better error handling in your code */
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Prevent connection switch! */
if (!$mysqli->query(sprintf("/*%s*/INSERT INTO test(id) VALUES (1)", MYSQLND_MS_LAST_USED_SWITCH))) {
    /* Please do proper ROLLBACK in your code, don't just die */
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
if ($res = $mysqli->query(sprintf("/*%s*/SELECT COUNT(*) AS _num FROM test", MYSQLND_MS_LAST_USED_SWITCH))
    $row = $res->fetch_assoc();
    $res->close();
    if ($row['_num'] > 1000) {
        if (!$mysqli->query(sprintf("/*%s*/INSERT INTO events(task) VALUES ('cleanup')", MYSQLND_MS_LAST_US
            die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
        }
    }
} else {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
if (!$mysqli->query(sprintf("/*%s*/UPDATE log SET last_update = NOW()", MYSQLND_MS_LAST_USED_SWITCH))) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
if (!$mysqli->query(sprintf("/*%s*/COMMIT", MYSQLND_MS_LAST_USED_SWITCH))) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

$mysqli->close();
?>
```

Starting with PHP 5.4.0, the `mysqlnd` library allows the plugin to monitor the status of the `autocommit` mode, if the mode is set by API calls instead of using SQL statements such as `SET AUTOCOMMIT=0`. This makes it possible for the plugin to become transaction aware. In this case, you do not need to use SQL hints.

If using PHP 5.4.0 or newer, API calls that enable `autocommit` mode, and when setting the plugin configuration option trx_stickiness=master, the plugin can automatically disable load balancing and connection switches for SQL transactions. In this configuration, the plugin stops load balancing if `autocommit` is disabled and directs all statements to the master. This prevents connection switches in

the middle of a transaction. Once `autocommit` is re-enabled, the plugin starts to load balance statements again.

API based transaction boundary detection has been improved with PHP 5.5.0 and `PECL/mysqlnd_ms` 1.5.0 to cover not only calls to `mysqli_autocommit` but also `mysqli_begin`, `mysqli_commit` and `mysqli_rollback`.

Example 7.16 Transaction aware load balancing: trx_stickiness setting

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "127.0.0.1",
                "port": "3306"
            }
        },
        "trx_stickiness": "master"
    }
}
```

Example 7.17 Transaction aware

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* Disable autocommit, plugin will run all statements on the master */
$mysqli->autocommit(false);

if (!$mysqli->query("INSERT INTO test(id) VALUES (1)")) {
    /* Please do proper ROLLBACK in your code, don't just die */
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
if ($res = $mysqli->query("SELECT COUNT(*) AS _num FROM test")) {
    $row = $res->fetch_assoc();
    $res->close();
    if ($row['_num'] > 1000) {
        if (!$mysqli->query("INSERT INTO events(task) VALUES ('cleanup')")) {
            die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
        }
    }
} else {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
if (!$mysqli->query("UPDATE log SET last_update = NOW()")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
if (!$mysqli->commit()) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
```

```
/* Plugin assumes that the transaction has ended and starts load balancing again */
$mysqli->autocommit(true);
$mysqli->close();
?>
```

> **Version requirement**
>
> The plugin configuration option trx_stickiness=master requires PHP 5.4.0 or newer.

Please note the restrictions outlined in the transaction handling concepts section.

7.4.6 XA/Distributed Transactions

> **Version requirement**
>
> XA related functions have been introduced in PECL mysqlnd_ms version 1.6.0-alpha.

> **Early adaptors wanted**
>
> The feature is currently under development. There may be issues and/or feature limitations. Do not use in production environments, although early lab tests indicate reasonable quality.
>
> Please, contact the development team if you are interested in this feature. We are looking for real life feedback to complement the feature.

XA transactions are a standardized method for executing transactions across multiple resources. Those resources can be databases or other transactional systems. The MySQL server supports XA SQL statements which allows users to carry out a distributed SQL transaction that spawns multiple database servers or any kind as long as they support the SQL statements too. In such a scenario it is in the responsibility of the user to coordinate the participating servers.

PECL/mysqlnd_ms can act as a transaction coordinator for a global (distributed, XA) transaction carried out on MySQL servers only. As a transaction coordinator, the plugin tracks all servers involved in a global transaction and transparently issues appropriate SQL statements on the participants. The global transactions are controlled with mysqlnd_ms_xa_begin, mysqlnd_ms_xa_commit and mysqlnd_ms_xa_rollback. SQL details are mostly hidden from the application as is the need to track and coordinate participants.

Example 7.18 General pattern for XA transactions

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* start a global transaction */
$gtrid_id = "12345";
if (!mysqlnd_ms_xa_begin($mysqli, $gtrid_id)) {
```

```
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* run queries as usual: XA BEGIN will be injected upon running a query */
if (!$mysqli->query("INSERT INTO orders(order_id, item) VALUES (1, 'christmas tree, 1.8m')")) {
    /* Either INSERT failed or the injected XA BEGIN failed */
    if ('XA' == substr($mysqli->sqlstate, 0, 2)) {
        printf("Global transaction/XA related failure, [%d] %s\n", $mysqli->errno, $mysqli->error);
    } else {
        printf("INSERT failed, [%d] %s\n", $mysqli->errno, $mysqli->error);
    }
    /* rollback global transaction */
    mysqlnd_ms_xa_rollback($mysqli, $xid);
    die("Stopping.");
}

/* continue carrying out queries on other servers, e.g. other shards */

/* commit the global transaction */
if (!mysqlnd_ms_xa_commit($mysqli, $xa_id)) {
    printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}
?>
```

Unlike with local transactions, which are carried out on a single server, XA transactions have an identifier (xid) associated with them. The XA transaction identifier is composed of a global transaction identifier (gtrid), a branch qualifier (bqual) a format identifier (formatID). Only the global transaction identifier can and must be given when calling any of the plugins XA functions.

Once a global transaction has been started, the plugin begins tracking servers until the global transaction ends. When a server is picked for query execution, the plugin injects the SQL statement `XA BEGIN` prior to executing the actual SQL statement on the server. `XA BEGIN` makes the server participate in the global transaction. If the injected SQL statement fails, the plugin will report the issue in reply to the query execution function that was used. In the above example, `$mysqli->query("INSERT INTO orders(order_id, item) VALUES (1, 'christmas tree, 1.8m')")` would indicate such an error. You may want to check the errors SQL state code to determine whether the actual query (here: `INSERT`) has failed or the error is related to the global transaction. It is up to you to ignore the failure to start the global transaction on a server and continue execution without having the server participate in the global transaction.

Example 7.19 Local and global transactions are mutually exclusive

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* start a local transaction */
if (!$mysqli->begin_transaction()) {
    die(sprintf("[%d/%s] %s\n", $mysqli->errno, $mysqli->sqlstate, $mysqli->error));
}

/* cannot start global transaction now - must end local transaction first */
$gtrid_id = "12345";
if (!mysqlnd_ms_xa_begin($mysqli, $gtrid_id)) {
    die(sprintf("[%d/%s] %s\n", $mysqli->errno, $mysqli->sqlstate, $mysqli->error));
}
?>
```

```
```

The above example will output:

```
Warning: mysqlnd_ms_xa_begin(): (mysqlnd_ms) Some work is done outside global transaction. You must end th
[1400/XAE09] (mysqlnd_ms) Some work is done outside global transaction. You must end the active local trans
```

A global transaction cannot be started when a local transaction is active. The plugin tries to detect this situation as early as possible, that is when `mysqlnd_ms_xa_begin` is called. If using API calls only to control transactions, the plugin will know that a local transaction is open and return an error for `mysqlnd_ms_xa_begin`. However, note the plugins limitations on detecting transaction boundaries.. In the worst case, if using direct SQL for local transactions (`BEGIN`, `COMMIT`, ...), it may happen that an error is delayed until some SQL is executed on a server.

To end a global transaction invoke `mysqlnd_ms_xa_commit` or `mysqlnd_ms_xa_rollback`. When a global transaction is ended all participants must be informed of the end. Therefore, PECL/mysqlnd_ms transparently issues appropriate XA related SQL statements on some or all of them. Any failure during this phase will cause an implicit rollback. The XA related API is intentionally kept simple here. A more complex API that gave more control would bare few, if any, advantages over a user implementation that issues all lower level XA SQL statements itself.

XA transactions use the two-phase commit protocol. The two-phase commit protocol is a blocking protocol. There are cases when no progress can be made, not even when using timeouts. Transaction coordinators should survive their own failure, be able to detect blockades and break ties. `PECL/mysqlnd_ms` takes the role of a transaction coordinator and can be configured to survive its own crash to avoid issues with blocked MySQL servers. Therefore, the plugin can and should be configured to use a persistent and crash-safe state to allow garbage collection of unfinished, aborted global transactions. A global transaction can be aborted in an open state if either the plugin fails (crashes) or a connection from the plugin to a global transaction participant fails.

Example 7.20 Transaction coordinator state store

```
{
    "myapp": {
        "xa": {
            "state_store": {
                "participant_localhost_ip": "192.168.2.12",
                "mysql": {
                    "host": "192.168.2.13",
                    "user": "root",
                    "password": "",
                    "db": "test",
                    "port": "3312",
                    "socket": null
                }
            }
        },
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
```

```
            "slave_0": {
                "host": "192.168.2.14",
                "port": "3306"
            }
        }
    }
}
```

Currently, `PECL/mysqlnd_ms` supports only using MySQL database tables as a state store. The SQL definitions of the tables are given in the plugin configuration section. Please, make sure to use a transactional and crash-safe storage engine for the tables, such as InnoDB. InnoDB is the default table engine in recent versions of the MySQL server. Make also sure the database server itself is highly available.

If a state store has been configured, the plugin can perform a garbage collection. During garbage collection it may be necessary to connect to a participant of a failed global transaction. Thus, the state store holds a list of participants and, among others, their host names. If the garbage collection is run on another host but the one that has written a participant entry with the host name `localhost`, then `localhost` resolves to different machines. There are two solutions to the problem. Either you do not configure any servers with the host name `localhost` but configure an IP address (and port) or, you hint the garbage collection. In the above example, `localhost` is used for `master_0`, hence it may not resolve to the correct host during garbage collection. However, `participant_localhost_ip` is also set to hint the garbage collection that `localhost` stands for the IP `192.168.2.12`.

7.4.7 Service level and consistency

> **Version requirement**
>
> Service levels have been introduced in PECL mysqlnd_ms version 1.2.0-alpha. `mysqlnd_ms_set_qos` is available with PHP 5.4.0 or newer.

Different types of MySQL cluster solutions offer different service and data consistency levels to their users. An asynchronous MySQL replication cluster offers eventual consistency by default. A read executed on an asynchronous slave may return current, stale or no data at all, depending on whether the slave has replayed all changesets from the master or not.

Applications using an MySQL replication cluster need to be designed to work correctly with eventual consistent data. In some cases, however, stale data is not acceptable. In those cases only certain slaves or even only master accesses are allowed to achieve the required quality of service from the cluster.

As of PECL mysqlnd_ms 1.2.0 the plugin is capable of selecting MySQL replication nodes automatically that deliver session consistency or strong consistency. Session consistency means that one client can read its writes. Other clients may or may not see the clients' write. Strong consistency means that all clients will see all writes from the client.

Example 7.21 Session consistency: read your writes

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
```

```
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "127.0.0.1",
                "port": "3306"
            }
        }
    }
}
```

Example 7.22 Requesting session consistency

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* read-write splitting: master used */
if (!$mysqli->query("INSERT INTO orders(order_id, item) VALUES (1, 'christmas tree, 1.8m')")) {
    /* Please use better error handling in your code */
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Request session consistency: read your writes */
if (!mysqlnd_ms_set_qos($mysqli, MYSQLND_MS_QOS_CONSISTENCY_SESSION)) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Plugin picks a node which has the changes, here: master */
if (!$res = $mysqli->query("SELECT item FROM orders WHERE order_id = 1")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

var_dump($res->fetch_assoc());

/* Back to eventual consistency: stale data allowed */
if (!mysqlnd_ms_set_qos($mysqli, MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL)) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Plugin picks any slave, stale data is allowed */
if (!$res = $mysqli->query("SELECT item, price FROM specials")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
?>
```

Service levels can be set in the plugins configuration file and at runtime using `mysqlnd_ms_set_qos`. In the example the function is used to enforce session consistency (read your writes) for all future statements until further notice. The `SELECT` statement on the `orders` table is run on the master to ensure the previous write can be seen by the client. Read-write splitting logic has been adapted to fulfill the service level.

After the application has read its changes from the `orders` table it returns to the default service level, which is eventual consistency. Eventual consistency puts no restrictions on choosing a node for statement execution. Thus, the `SELECT` statement on the `specials` table is executed on a slave.

The new functionality supersedes the use of SQL hints and the `master_on_write` configuration option. In many cases `mysqlnd_ms_set_qos` is easier to use, more powerful improves portability.

Example 7.23 Maximum age/slave lag

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "127.0.0.1",
                "port": "3306"
            }
        },
        "failover" : "master"
    }
}
```

Example 7.24 Limiting slave lag

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* Read from slaves lagging no more than four seconds */
$ret = mysqlnd_ms_set_qos(
    $mysqli,
    MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL,
    MYSQLND_MS_QOS_OPTION_AGE,
    4
);

if (!$ret) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Plugin picks any slave, which may or may not have the changes */
if (!$res = $mysqli->query("SELECT item, price FROM daytrade")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Back to default: use of all slaves and masters permitted */
if (!mysqlnd_ms_set_qos($mysqli, MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL)) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
?>
```

The eventual consistency service level can be used with an optional parameter to set a maximum slave lag for choosing slaves. If set, the plugin checks `SHOW SLAVE STATUS` for all configured slaves. In case of the example, only slaves for which `Slave_IO_Running=Yes`, `Slave_SQL_Running=Yes` and

`Seconds_Behind_Master <= 4` is true are considered for executing the statement `SELECT item, price FROM daytrade`.

Checking `SHOW SLAVE STATUS` is done transparently from an applications perspective. Errors, if any, are reported as warnings. No error will be set on the connection handle. Even if all `SHOW SLAVE STATUS` SQL statements executed by the plugin fail, the execution of the users statement is not stopped, given that master fail over is enabled. Thus, no application changes are required.

> **Expensive and slow operation**
>
> Checking `SHOW SLAVE STATUS` for all slaves adds overhead to the application. It is an expensive and slow background operation. Try to minimize the use of it. Unfortunately, a MySQL replication cluster does not give clients the possibility to request a list of candidates from a central instance. Thus, a more efficient way of checking the slaves lag is not available.
>
> Please, note the limitations and properties of `SHOW SLAVE STATUS` as explained in the MySQL reference manual.

To prevent mysqlnd_ms from emitting a warning if no slaves can be found that lag no more than the defined number of seconds behind the master, it is necessary to enable master fail over in the plugins configuration file. If no slaves can be found and fail over is turned on, the plugin picks a master for executing the statement.

If no slave can be found and fail over is turned off, the plugin emits a warning, it does not execute the statement and it sets an error on the connection.

Example 7.25 Fail over not set

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "127.0.0.1",
                "port": "3306"
            }
        }
    }
}
```

Example 7.26 No slave within time limit

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* Read from slaves lagging no more than four seconds */
```

```
$ret = mysqlnd_ms_set_qos(
    $mysqli,
    MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL,
    MYSQLND_MS_QOS_OPTION_AGE,
    4
);

if (!$ret) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Plugin picks any slave, which may or may not have the changes */
if (!$res = $mysqli->query("SELECT item, price FROM daytrade")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}


/* Back to default: use of all slaves and masters permitted */
if (!mysqlnd_ms_set_qos($mysqli, MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL)) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
?>
```

The above example will output:

```


PHP Warning:  mysqli::query(): (mysqlnd_ms) Couldn't find the appropriate slave connection. 0 slaves to choose
PHP Warning:  mysqli::query(): (mysqlnd_ms) No connection selected by the last filter in %s on line %d
[2000] (mysqlnd_ms) No connection selected by the last filter
```

7.4.8 Global transaction IDs

> **Version requirement**
>
> A client-side global transaction ID injection has been introduced in mysqlnd_ms version 1.2.0-alpha. The feature is not required for synchronous clusters, such as MySQL Cluster. Use it with asynchronous clusters such as classical MySQL replication.
>
> As of MySQL 5.6.5-m8 release candidate the MySQL server features built-in global transaction identifiers. The MySQL built-in global transaction ID feature is supported by `PECL/mysqlnd_ms` 1.3.0-alpha or later. However, the final feature set found in MySQL 5.6 production releases to date is not sufficient to support the ideas discussed below in all cases. Please, see also the concepts section.

`PECL/mysqlnd_ms` can either use its own global transaction ID emulation or the global transaction ID feature built-in to MySQL 5.6.5-m8 or later. From a developer perspective the client-side and server-side approach offer the same features with regards to service levels provided by PECL/mysqlnd_ms. Their differences are discussed in the concepts section.

The quickstart first demonstrates the use of the client-side global transaction ID emulation built-in to `PECL/mysqlnd_ms` before its show how to use the server-side counterpart. The order ensures that the underlying idea is discussed first.

Idea and client-side emulation

In its most basic form a global transaction ID (GTID) is a counter in a table on the master. The counter is incremented whenever a transaction is committed on the master. Slaves replicate the table. The counter serves two purposes. In case of a master failure, it helps the database administrator to identify the most recent slave for promoting it to the new master. The most recent slave is the one with the highest counter value. Applications can use the global transaction ID to search for slaves which have replicated a certain write (identified by a global transaction ID) already.

`PECL/mysqlnd_ms` can inject SQL for every committed transaction to increment a GTID counter. The so created GTID is accessible by the application to identify an applications write operation. This enables the plugin to deliver session consistency (read your writes) service level by not only querying masters but also slaves which have replicated the change already. Read load is taken away from the master.

Client-side global transaction ID emulation has some limitations. Please, read the concepts section carefully to fully understand the principles and ideas behind it, before using in production environments. The background knowledge is not required to continue with the quickstart.

First, create a counter table on your master server and insert a record into it. The plugin does not assist creating the table. Database administrators must make sure it exists. Depending on the error reporting mode, the plugin will silently ignore the lack of the table or bail out.

Example 7.27 Create counter table on master

```
CREATE TABLE `trx` (
  `trx_id` int(11) DEFAULT NULL,
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=latin1
INSERT INTO `trx`(`trx_id`) VALUES (1);
```

In the plugins configuration file set the SQL to update the global transaction ID table using `on_commit` from the `global_transaction_id_injection` section. Make sure the table name used for the `UPDATE` statement is fully qualified. In the example, `test.trx` is used to refer to table `trx` in the schema (database) `test`. Use the table that was created in the previous step. It is important to set the fully qualified table name because the connection on which the injection is done may use a different default database. Make sure the user that opens the connection is allowed to execute the `UPDATE`.

Enable reporting of errors that may occur when mysqlnd_ms does global transaction ID injection.

Example 7.28 Plugin config: SQL for client-side GTID injection

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "127.0.0.1",
                "port": "3306"
            }
        },
        "global_transaction_id_injection":{
            "on_commit":"UPDATE test.trx SET trx_id = trx_id + 1",
            "report_error":true
```

```
        }
    }
}
```

Example 7.29 Transparent global transaction ID injection

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* auto commit mode, transaction on master, GTID must be incremented */
if (!$mysqli->query("DROP TABLE IF EXISTS test")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* auto commit mode, transaction on master, GTID must be incremented */
if (!$mysqli->query("CREATE TABLE test(id INT)")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* auto commit mode, transaction on master, GTID must be incremented */
if (!$mysqli->query("INSERT INTO test(id) VALUES (1)")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* auto commit mode, read on slave, no increment */
if (!($res = $mysqli->query("SELECT id FROM test"))) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

var_dump($res->fetch_assoc());
?>
```

The above example will output:

```
array(1) {
  ["id"]=>
  string(1) "1"
}
```

The example runs three statements in auto commit mode on the master, causing three transactions on the master. For every such statement, the plugin will inject the configured `UPDATE` transparently before executing the users SQL statement. When the script ends the global transaction ID counter on the master has been incremented by three.

The fourth SQL statement executed in the example, a `SELECT`, does not trigger an increment. Only transactions (writes) executed on a master shall increment the GTID counter.

> **SQL for global transaction ID: efficient solution wanted!**
>
> The SQL used for the client-side global transaction ID emulation is inefficient. It is optimized for clearity not for performance. Do not use it for production

environments. Please, help finding an efficient solution for inclusion in the manual. We appreciate your input.

Example 7.30 Plugin config: SQL for fetching GTID

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "127.0.0.1",
                "port": "3306"
            }
        },
        "global_transaction_id_injection":{
            "on_commit":"UPDATE test.trx SET trx_id = trx_id + 1",
            "fetch_last_gtid" : "SELECT MAX(trx_id) FROM test.trx",
            "report_error":true
        }
    }
}
```

Example 7.31 Obtaining GTID after injection

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* auto commit mode, transaction on master, GTID must be incremented */
if (!$mysqli->query("DROP TABLE IF EXISTS test")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

printf("GTID after transaction %s\n", mysqlnd_ms_get_last_gtid($mysqli));

/* auto commit mode, transaction on master, GTID must be incremented */
if (!$mysqli->query("CREATE TABLE test(id INT)")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

printf("GTID after transaction %s\n", mysqlnd_ms_get_last_gtid($mysqli));
?>
```

The above example will output:

```
GTID after transaction 7
GTID after transaction 8
```

Applications can ask PECL mysqlnd_ms for a global transaction ID which belongs to the last write operation performed by the application. The function `mysqlnd_ms_get_last_gtid` returns the GTID obtained when executing the SQL statement from the `fetch_last_gtid` entry of the `global_transaction_id_injection` section from the plugins configuration file. The function may be called after the GTID has been incremented.

Applications are adviced not to run the SQL statement themselves as this bares the risk of accidently causing an implicit GTID increment. Also, if the function is used, it is easy to migrate an application from one SQL statement for fetching a transaction ID to another, for example, if any MySQL server ever features built-in global transaction ID support.

The quickstart shows a SQL statement which will return a GTID equal or greater to that created for the previous statement. It is exactly the GTID created for the previous statement if no other clients have incremented the GTID in the time span between the statement execution and the `SELECT` to fetch the GTID. Otherwise, it is greater.

Example 7.32 Plugin config: Checking for a certain GTID

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "127.0.0.1",
                "port": "3306"
            }
        },
        "global_transaction_id_injection":{
            "on_commit":"UPDATE test.trx SET trx_id = trx_id + 1",
            "fetch_last_gtid" : "SELECT MAX(trx_id) FROM test.trx",
            "check_for_gtid" : "SELECT trx_id FROM test.trx WHERE trx_id >= #GTID",
            "report_error":true
        }
    }
}
```

Example 7.33 Session consistency service level and GTID combined

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* auto commit mode, transaction on master, GTID must be incremented */
if (   !$mysqli->query("DROP TABLE IF EXISTS test")
    || !$mysqli->query("CREATE TABLE test(id INT)")
    || !$mysqli->query("INSERT INTO test(id) VALUES (1)")
) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
```

```
}

/* GTID as an identifier for the last write */
$gtid = mysqlnd_ms_get_last_gtid($mysqli);

/* Session consistency (read your writes): try to read from slaves not only master */
if (false == mysqlnd_ms_set_qos($mysqli, MYSQLND_MS_QOS_CONSISTENCY_SESSION, MYSQLND_MS_QOS_OPTION_GTID, $g
    die(sprintf("[006] [%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Either run on master or a slave which has replicated the INSERT */
if (!($res = $mysqli->query("SELECT id FROM test"))) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

var_dump($res->fetch_assoc());
?>
```

A GTID returned from `mysqlnd_ms_get_last_gtid` can be used as an option for the session consistency service level. Session consistency delivers read your writes. Session consistency can be requested by calling `mysqlnd_ms_set_qos`. In the example, the plugin will execute the `SELECT` statement either on the master or on a slave which has replicated the previous `INSERT` already.

PECL mysqlnd_ms will transparently check every configured slave if it has replicated the `INSERT` by checking the slaves GTID table. The check is done running the SQL set with the `check_for_gtid` option from the `global_transaction_id_injection` section of the plugins configuration file. Please note, that this is a slow and expensive procedure. Applications should try to use it sparsely and only if read load on the master becomes to high otherwise.

Use of the server-side global transaction ID feature

> **Insufficient server support in MySQL 5.6**
>
> The plugin has been developed against a pre-production version of MySQL 5.6. It turns out that all released production versions of MySQL 5.6 do not provide clients with enough information to enforce session consistency based on GTIDs. Please, read the concepts section for details.

Starting with MySQL 5.6.5-m8 the MySQL Replication system features server-side global transaction IDs. Transaction identifiers are automatically generated and maintained by the server. Users do not need to take care of maintaining them. There is no need to setup any tables in advance, or for setting `on_commit`. A client-side emulation is no longer needed.

Clients can continue to use global transaction identifier to achieve session consistency when reading from MySQL Replication slaves in some cases but not all! The algorithm works as described above. Different SQL statements must be configured for `fetch_last_gtid` and `check_for_gtid`. The statements are given below. Please note, MySQL 5.6.5-m8 is a development version. Details of the server implementation may change in the future and require adoption of the SQL statements shown.

Using the following configuration any of the above described functionality can be used together with the server-side global transaction ID feature. `mysqlnd_ms_get_last_gtid` and `mysqlnd_ms_set_qos` continue to work as described above. The only difference is that the server does not use a simple sequence number but a string containing of a server identifier and a sequence number. Thus, users cannot easily derive an order from GTIDs returned by `mysqlnd_ms_get_last_gtid`.

Example 7.34 Plugin config: using MySQL 5.6.5-m8 built-in GTID feature

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "127.0.0.1",
                "port": "3306"
            }
        },
        "global_transaction_id_injection":{
            "fetch_last_gtid" : "SELECT @@GLOBAL.GTID_DONE AS trx_id FROM DUAL",
            "check_for_gtid" : "SELECT GTID_SUBSET('#GTID', @@GLOBAL.GTID_DONE) AS trx_id FROM DUAL",
            "report_error":true
        }
    }
}
```

7.4.9 Cache integration

> **Version requirement, dependencies and status**
>
> Please, find more about version requirements, extension load order dependencies and the current status in the concepts section!

Databases clusters can deliver different levels of consistency. As of `PECL/mysqlnd_ms` 1.2.0 it is possible to advice the plugin to consider only cluster nodes that can deliver the consistency level requested. For example, if using asynchronous MySQL Replication with its cluster-wide eventual consistency, it is possible to request session consistency (read your writes) at any time using `mysqlnd_ms_set_quos`. Please, see also the service level and consistency introduction.

Example 7.35 Recap: quality of service to request read your writes

```
/* Request session consistency: read your writes */
if (!mysqlnd_ms_set_qos($mysqli, MYSQLND_MS_QOS_CONSISTENCY_SESSION))
  die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
```

Assuming PECL/mysqlnd has been explicitly told to deliver no consistency level higher than eventual consistency, it is possible to replace a database node read access with a client-side cache using time-to-live (TTL) as its invalidation strategy. Both the database node and the cache may or may not serve current data as this is what eventual consistency defines.

Replacing a database node read access with a local cache access can improve overall performance and lower the database load. If the cache entry is every reused by other clients than the one creating the cache entry, a database access is saved and thus database load is lowered. Furthermore, system performance can become better if computation and delivery of a database query is slower than a local cache access.

Example 7.36 Plugin config: no special entries for caching

```
```

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "127.0.0.1",
                "port": "3306"
            }
        },
    }
}
```

Example 7.37 Caching a slave request

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

if (   !$mysqli->query("DROP TABLE IF EXISTS test")
    || !$mysqli->query("CREATE TABLE test(id INT)")
    || !$mysqli->query("INSERT INTO test(id) VALUES (1)")
) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Explicitly allow eventual consistency and caching (TTL <= 60 seconds) */
if (false == mysqlnd_ms_set_qos($mysqli, MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL, MYSQLND_MS_QOS_OPTION_CACHE,
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* To make this example work, we must wait for a slave to catch up. Brute force style. */
$attempts = 0;
do {
    /* check if slave has the table */
    if ($res = $mysqli->query("SELECT id FROM test")) {
        break;
    } else if ($mysqli->errno) {
        die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
    }
    /* wait for slave to catch up */
    usleep(200000);
} while ($attempts++ < 10);

/* Query has been run on a slave, result is in the cache */
assert($res);
var_dump($res->fetch_assoc());

/* Served from cache */
$res = $mysqli->query("SELECT id FROM test");
?>
```

The example shows how to use the cache feature. First, you have to set the quality of service to eventual consistency and explicitly allow for caching. This is done by calling `mysqlnd_ms_set_qos`. Then,

the result set of every read-only statement is cached for upto that many seconds as allowed with `mysqlnd_ms_set_qos`.

The actual TTL is lower or equal to the value set with `mysqlnd_ms_set_qos`. The value passed to the function sets the maximum age (seconds) of the data delivered. To calculate the actual TTL value the replication lag on a slave is checked and subtracted from the given value. If, for example, the maximum age is set to 60 seconds and the slave reports a lag of 10 seconds the resulting TTL is 50 seconds. The TTL is calculated individually for every cached query.

Example 7.38 Read your writes and caching combined

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

if (   !$mysqli->query("DROP TABLE IF EXISTS test")
    || !$mysqli->query("CREATE TABLE test(id INT)")
    || !$mysqli->query("INSERT INTO test(id) VALUES (1)")
) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Explicitly allow eventual consistency and caching (TTL <= 60 seconds) */
if (false == mysqlnd_ms_set_qos($mysqli, MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL, MYSQLND_MS_QOS_OPTION_CACHE, 60)
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* To make this example work, we must wait for a slave to catch up. Brute force style. */
$attempts = 0;
do {
    /* check if slave has the table */
    if ($res = $mysqli->query("SELECT id FROM test")) {
        break;
    } else if ($mysqli->errno) {
        die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
    }
    /* wait for slave to catch up */
    usleep(200000);
} while ($attempts++ < 10);

assert($res);

/* Query has been run on a slave, result is in the cache */
var_dump($res->fetch_assoc());

/* Served from cache */
if (!($res = $mysqli->query("SELECT id FROM test"))) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
var_dump($res->fetch_assoc());

/* Update on master */
if (!$mysqli->query("UPDATE test SET id = 2")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Read your writes */
if (false == mysqlnd_ms_set_qos($mysqli, MYSQLND_MS_QOS_CONSISTENCY_SESSION)) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
```

```
/* Fetch latest data */
if (!($res = $mysqli->query("SELECT id FROM test"))) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
var_dump($res->fetch_assoc());
?>
```

The quality of service can be changed at any time to avoid further cache usage. If needed, you can switch to read your writes (session consistency). In that case, the cache will not be used and fresh data is read.

7.4.10 Failover

By default, the plugin does not attempt to fail over if connecting to a host fails. This prevents pitfalls related to connection state. It is recommended to manually handle connection errors in a way similar to a failed transaction. You should catch the error, rebuild the connection state and rerun your query as shown below.

If connection state is no issue to you, you can alternatively enable automatic and silent failover. Depending on the configuration, the automatic and silent failover will either attempt to fail over to the master before issuing and error or, try to connect to other slaves, given the query allowes for it, before attempting to connect to a master. Because automatic failover is not fool-proof, it is not discussed in the quickstart. Instead, details are given in the concepts section below.

Example 7.39 Manual failover, automatic optional

```
 {
   "myapp": {
       "master": {
           "master_0": {
               "host": "localhost",
               "socket": "\/tmp\/mysql.sock"
           }
       },
       "slave": {
           "slave_0": {
               "host": "simulate_slave_failure",
               "port": "0"
           },
           "slave_1": {
               "host": "127.0.0.1",
               "port": 3311
           }
       },
      "filters": { "roundrobin": [] }
   }
}
```

Example 7.40 Manual failover

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
```

```
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

$sql = "SELECT 1 FROM DUAL";

/* error handling as it should be done regardless of the plugin */
if (!($res = $link->query($sql))) {
    /* plugin specific: check for connection error */
    switch ($link->errno) {
    case 2002:
    case 2003:
    case 2005:
        printf("Connection error - trying next slave!\n");
        /* load balancer will pick next slave */
        $res = $link->query($sql);
        break;
    default:
        /* no connection error, failover is unlikely to help */
        die(sprintf("SQL error: [%d] %s", $link->errno, $link->error));
        break;
    }
}
if ($res) {
    var_dump($res->fetch_assoc());
}
?>
```

7.4.11 Partitioning and Sharding

Database clustering is done for various reasons. Clusters can improve availability, fault tolerance, and increase performance by applying a divide and conquer approach as work is distributed over many machines. Clustering is sometimes combined with partitioning and sharding to further break up a large complex task into smaller, more manageable units.

The mysqlnd_ms plugin aims to support a wide variety of MySQL database clusters. Some flavors of MySQL database clusters have built-in methods for partitioning and sharding, which could be transparent to use. The plugin supports the two most common approaches: MySQL Replication table filtering, and Sharding (application based partitioning).

MySQL Replication supports partitioning as filters that allow you to create slaves that replicate all or specific databases of the master, or tables. It is then in the responsibility of the application to choose a slave according to the filter rules. You can either use the mysqlnd_ms `node_groups` filter to manually support this, or use the experimental table filter.

Manual partitioning or sharding is supported through the node grouping filter, and SQL hints as of 1.5.0. The node_groups filter lets you assign a symbolic name to a group of master and slave servers. In the example, the master `master_0` and `slave_0` form a group with the name `Partition_A`. It is entirely up to you to decide what makes up a group. For example, you may use node groups for sharding, and use the group names to address shards like `Shard_A_Range_0_100`.

Example 7.41 Cluster node groups

```
{
 "myapp": {
      "master": {
           "master_0": {
```

```
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "simulate_slave_failure",
                "port": "0"
            },
            "slave_1": {
                "host": "127.0.0.1",
                "port": 3311
            }
        },
        "filters": {
            "node_groups": {
                "Partition_A" : {
                    "master": ["master_0"],
                    "slave": ["slave_0"]
                }
            },
           "roundrobin": []
        }
    }
}
```

Example 7.42 Manual partitioning using SQL hints

```
<?php
function select($mysqli, $msg, $hint = '')
{
    /* Note: weak test, two connections to two servers may have the same thread id */
    $sql = sprintf("SELECT CONNECTION_ID() AS _thread, '%s' AS _hint FROM DUAL", $msg);
    if ($hint) {
        $sql = $hint . $sql;
    }
    if (!($res = $mysqli->query($sql))) {
        printf("[%d] %s", $mysqli->errno, $mysqli->error);
        return false;
    }
    $row =  $res->fetch_assoc();
    printf("%d - %s - %s\n", $row['_thread'], $row['_hint'], $sql);
    return true;
}

$mysqli = new mysqli("myapp", "user", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* All slaves allowed */
select($mysqli, "slave_0");
select($mysqli, "slave_1");

/* only servers of node group "Partition_A" allowed */
select($mysqli, "slave_1", "/*Partition_A*/");
select($mysqli, "slave_1", "/*Partition_A*/");
?>
```

```
6804 - slave_0 - SELECT CONNECTION_ID() AS _thread, 'slave1' AS _hint FROM DUAL
2442 - slave_1 - SELECT CONNECTION_ID() AS _thread, 'slave2' AS _hint FROM DUAL
6804 - slave_0 - /*Partition_A*/SELECT CONNECTION_ID() AS _thread, 'slave1' AS _hint FROM DUAL
6804 - slave_0 - /*Partition_A*/SELECT CONNECTION_ID() AS _thread, 'slave1' AS _hint FROM DUAL
```

By default, the plugin will use all configured master and slave servers for query execution. But if a query begins with a SQL hint like `/*node_group*/`, the plugin will only consider the servers listed in the `node_group` for query execution. Thus, `SELECT` queries prefixed with `/*Partition_A*/` will only be executed on `slave_0`.

7.4.12 MySQL Fabric

> **Version requirement and status**
>
> Work on supporting MySQL Fabric started in version 1.6. Please, consider the support to be of pre-alpha quality. The manual may not list all features or feature limitations. This is work in progress.
>
> Sharding is the only use case supported by the plugin to date.

> **MySQL Fabric concepts**
>
> Please, check the MySQL reference manual for more information about MySQL Fabric and how to set it up. The PHP manual assumes that you are familiar with the basic concepts and ideas of MySQL Fabric.

MySQL Fabric is a system for managing farms of MySQL servers to achive High Availability and optionally support sharding. Technically, it is a middleware to manage and monitor MySQL servers.

Clients query MySQL Fabric to obtain lists of MySQL servers, their state and their roles. For example, clients can request a list of slaves for a MySQL Replication group and whether they are ready to handle SQL requests. Another example is a cluster of sharded MySQL servers where the client seeks to know which shard to query for a given table and shard key. If configured to use Fabric, the plugin uses XML RCP over HTTP to obtain the list at runtime from a MySQL Fabric host. The XML remote procedure call itself is done in the background and transparent from a developers point of view.

Instead of listing MySQL servers directly in the plugins configuration file it contains a list of one or more MySQL Fabric hosts

Example 7.43 Plugin config: Fabric hosts instead of MySQL servers

```
{
    "myapp": {
        "fabric": {
            "hosts": [
                {
                    "host" : "127.0.0.1",
                    "port" : 8080
                }
            ]
        }
    }
}
```

Users utilize the new functions `mysqlnd_ms_fabric_select_shard` and `mysqlnd_ms_fabric_select_global` to switch to the set of servers responsible for a given shard key. Then, the plugin picks an appropriate server for running queries on. When doing so, the plugin takes care of additional load balancing rules set.

The below example assumes that MySQL Fabric has been setup to shard the table `test.fabrictest` using the `id` column of the table as a shard key.

Example 7.44 Manual partitioning using SQL hints

```
<?php
$mysqli = new mysqli("myapp", "user", "password", "database");
if (!$mysqli) {
    /* Of course, your error handling is nicer... */
    die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));
}

/* Create a global table - a table available on all shards */
mysqlnd_ms_fabric_select_global($mysqli, "test.fabrictest");
if (!$mysqli->query("CREATE TABLE test.fabrictest(id INT NOT NULL PRIMARY KEY)")) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Switch connection to appropriate shard and insert record */
mysqlnd_ms_fabric_select_shard($mysqli, "test.fabrictest", 10);
if (!($res = $mysqli->query("INSERT INTO fabrictest(id) VALUES (10)"))) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}

/* Try to read newly inserted record */
mysqlnd_ms_fabric_select_shard($mysqli, "test.fabrictest", 10);
if (!($res = $mysqli->query("SELECT id FROM test WHERE id = 10"))) {
    die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
}
?>
```

The example creates the sharded table, inserts a record and reads the record thereafter. All SQL data definition language (DDL) operations on a sharded table must be applied to the so called global server group. Prior to creating or altering a sharded table, `mysqlnd_ms_fabric_select_global` is called to switch the given connection to the corresponding servers of the global group. Data manipulation (DML) SQL statements must be sent to the shards directly. The `mysqlnd_ms_fabric_select_shard` switches a connection to shards handling a certain shard key.

7.5 Concepts

This explains the architecture and related concepts for this plugin, and describes the impact that MySQL replication and this plugin have on developmental tasks while using a database cluster. Reading and understanding these concepts is required, in order to use this plugin with success.

7.5.1 Architecture

The mysqlnd replication and load balancing plugin is implemented as a PHP extension. It is written in C and operates under the hood of PHP. During the startup of the PHP interpreter, in the module init phase of the PHP engine, it gets registered as a mysqlnd plugin to replace selected mysqlnd C methods.

At PHP runtime, it inspects queries sent from mysqlnd (PHP) to the MySQL server. If a query is recognized as read-only, it will be sent to one of the configured slave servers. Statements are considered read-only if they either start with `SELECT`, the SQL hint `/*ms=slave*/` or a slave had been chosen for running the previous query, and the query started with the SQL hint `/*ms=last_used*/`. In all other cases, the query will be sent to the MySQL replication master server.

For better portability, applications should use the `MYSQLND_MS_MASTER_SWITCH`, `MYSQLND_MS_SLAVE_SWITCH`, and `MYSQLND_MS_LAST_USED_SWITCH` predefined mysqlnd_ms constants, instead of their literal values, such as `/*ms=slave*/`.

The plugin handles the opening and closing of database connections to both master and slave servers. From an application point of view, there continues to be only one connection handle. However, internally, this one public connection handle represents a pool of network connections that are managed by the plugin. The plugin proxies queries to the master server, and to the slaves using multiple connections.

Database connections have a state consisting of, for example, transaction status, transaction settings, character set settings, and temporary tables. The plugin will try to maintain the same state among all internal connections, whenever this can be done in an automatic and transparent way. In cases where it is not easily possible to maintain state among all connections, such as when using `BEGIN TRANSACTION`, the plugin leaves it to the user to handle.

7.5.2 Connection pooling and switching

The replication and load balancing plugin changes the semantics of a PHP MySQL connection handle. The existing API of the PHP MySQL extensions (mysqli, mysql, and PDO_MYSQL) are not changed in a way that functions are added or removed. But their behavior changes when using the plugin. Existing applications do not need to be adapted to a new API, but they may need to be modified because of the behavior changes.

The plugin breaks the one-by-one relationship between a mysqli, mysql, and PDO_MYSQL connection handle and a MySQL network connection. And a mysqli, mysql, and PDO_MYSQL connection handle represents a local pool of connections to the configured MySQL replication master and MySQL replication slave servers. The plugin redirects queries to the master and slave servers. At some point in time one and the same PHP connection handle may point to the MySQL master server. Later on, it may point to one of the slave servers or still the master. Manipulating and replacing the network connection referenced by a PHP MySQL connection handle is not a transparent operation.

Every MySQL connection has a state. The state of the connections in the connection pool of the plugin can differ. Whenever the plugin switches from one wire connection to another, the current state of the user connection may change. The applications must be aware of this.

The following list shows what the connection state consists of. The list may not be complete.

- Transaction status
- Temporary tables
- Table locks
- Session system variables and session user variables

- The current database set using `USE` and other state chaining SQL commands
- Prepared statements
- `HANDLER` variables
- Locks acquired with `GET_LOCK()`

Connection switches happen right before queries are executed. The plugin does not switch the current connection until the next statement is executed.

> **Replication issues**
>
> See also the MySQL reference manual chapter about replication features and related issues. Some restrictions may not be related to the PHP plugin, but are properties of the MySQL replication system.

Broadcasted messages

The plugins philosophy is to align the state of connections in the pool only if the state is under full control of the plugin, or if it is necessary for security reasons. Just a few actions that change the state of the connection fall into this category.

The following is a list of connection client library calls that change state, and are broadcasted to all open connections in the connection pool.

If any of the listed calls below are to be executed, the plugin loops over all open master and slave connections. The loop continues until all servers have been contacted, and the loop does not break if a server indicates a failure. If possible, the failure will propagate to the called user API function, which may be detected depending on which underlying library function was triggered.

Library call	Notes	Version
`change_user()`	Called by the `mysqli_change_user` user API call. Also triggered upon reuse of a persistent `mysqli` connection.	Since 1.0.0.
`select_db()`	Called by the following user API calls: `mysql_select_db`, `mysql_list_tables`, `mysql_db_query`, `mysql_list_fields`, `mysqli_select_db`. Note, that SQL `USE` is not monitored.	Since 1.0.0.
`set_charset()`	Called by the following user API calls: `mysql_set_charset`. `mysqli_set_charset`. Note, that SQL `SET NAMES` is not monitored.	Since 1.0.0.
`set_server_option()`	Called by the following user API calls: `mysqli_multi_query`, `mysqli_real_query`, `mysqli_query`, `mysql_query`.	Since 1.0.0.
`set_client_option()`	Called by the following user API calls: `mysqli_options`, `mysqli_ssl_set`, `mysqli_connect`, `mysql_connect`, `mysql_pconnect`.	Since 1.0.0.
`set_autocommit()`	Called by the following user API calls: `mysqli_autocommit`, `PDO::setAttribute(PDO::ATTR_AUTOCOMMIT)`.	Since 1.0.0. PHP >= 5.4.0.
`ssl_set()`	Called by the following user API calls: `mysqli_ssl_set`.	Since 1.1.0.

Broadcasting and lazy connections

The plugin does not proxy or “remember” all settings to apply them on connections opened in the future. This is important to remember, if using lazy connections. Lazy connections are connections which are not opened before the client sends the first connection. Use of lazy connections is the default plugin action.

The following connection library calls each changed state, and their execution is recorded for later use when lazy connections are opened. This helps ensure that the connection state of all connections in the connection pool are comparable.

Library call	Notes	Version
`change_user()`	User, password and database recorded for future use.	Since 1.1.0.
`select_db`	Database recorded for future use.	Since 1.1.0.
`set_charset()`	Calls `set_client_option(MYSQL_SET_CHARSET_NAME, charset)` on lazy connection to ensure `charset` will be used upon opening the lazy connection.	Since 1.1.0.
`set_autocommit()`	Adds `SET AUTOCOMMIT=0|1` to the list of init commands of a lazy connection using `set_client_option(MYSQL_INIT_COMMAND, "SET AUTOCOMMIT=...%quot;)`.	Since 1.1.0. PHP >= 5.4.0.

> **Connection state**
>
> The connection state is not only changed by API calls. Thus, even if PECL mysqlnd_ms monitors all API calls, the application must still be aware. Ultimately, it is the applications responsibility to maintain the connection state, if needed.

Charsets and string escaping

Due to the use of lazy connections, which are a default, it can happen that an application tries to escape a string for use within SQL statements before a connection has been established. In this case string escaping is not possible. The string escape function does not know what charset to use before a connection has been established.

To overcome the problem a new configuration setting `server_charset` has been introduced in version 1.4.0.

Attention has to be paid on escaping strings with a certain charset but using the result on a connection that uses a different charset. Please note, that PECL/mysqlnd_ms manipulates connections and one application level connection represents a pool of multiple connections that all may have different default charsets. It is recommended to configure the servers involved to use the same default charsets. The configuration setting `server_charset` does help with this situation as well. If using `server_charset`, the plugin will set the given charset on all newly opened connections.

7.5.3 Local transaction handling

Transaction handling is fundamentally changed. An SQL transaction is a unit of work that is run on one database server. The unit of work consists of one or more SQL statements.

By default the plugin is not aware of SQL transactions. The plugin may switch connections for load balancing at any point in time. Connection switches may happen in the middle of a transaction. This is against the nature of an SQL transaction. By default, the plugin is not transaction safe.

Any kind of MySQL load balancer must be hinted about the begin and end of a transaction. Hinting can either be done implicitly by monitoring API calls or using SQL hints. Both options are supported by the plugin, depending on your PHP version. API monitoring requires PHP 5.4.0 or newer. The plugin, like any other MySQL load balancer, cannot detect transaction boundaries based on the MySQL Client Server

Protocol. Thus, entirely transparent transaction aware load balancing is not possible. The least intrusive option is API monitoring, which requires little to no application changes, depending on your application.

Please, find examples of using SQL hints or the API monitoring in the examples section. The details behind the API monitoring, which makes the plugin transaction aware, are described below.

Beginning with PHP 5.4.0, the mysqlnd library allows this plugin to subclass the library C API call `set_autocommit()`, to detect the status of `autocommit` mode.

The PHP MySQL extensions either issue a query (such as `SET AUTOCOMMIT=0|1`), or use the mysqlnd library call `set_autocommit()` to control the `autocommit` setting. If an extension makes use of `set_autocommit()`, the plugin can be made transaction aware. Transaction awareness cannot be achieved if using SQL to set the autocommit mode. The library function `set_autocommit()` is called by the `mysqli_autocommit` and `PDO::setAttribute(PDO::ATTR_AUTOCOMMIT)` user API calls.

The plugin configuration option trx_stickiness=master can be used to make the plugin transactional aware. In this mode, the plugin stops load balancing if autocommit becomes disabled, and directs all statements to the master until autocommit gets enabled.

An application that does not want to set SQL hints for transactions but wants to use the transparent API monitoring to avoid application changes must make sure that the autocommit settings is changed exclusively through the listed API calls.

API based transaction boundary detection has been improved with PHP 5.5.0 and PECL/mysqlnd_ms 1.5.0 to cover not only calls to `mysqli_autocommit` but also `mysqli_begin`, `mysqli_commit` and `mysqli_rollback`.

7.5.4 Error handling

Applications using PECL/mysqlnd_ms should implement proper error handling for all user API calls. And because the plugin changes the semantics of a connection handle, API calls may return unexpected errors. If using the plugin on a connection handle that no longer represents an individual network connection, but a connection pool, an error code and error message will be set on the connection handle whenever an error occurs on any of the network connections behind.

If using lazy connections, which is the default, connections are not opened until they are needed for query execution. Therefore, an API call for a statement execution may return a connection error. In the example below, an error is provoked when trying to run a statement on a slave. Opening a slave connection fails because the plugin configuration file lists an invalid host name for the slave.

Example 7.45 Provoking a connection error

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": {
            "slave_0": {
                "host": "invalid_host_name",
            }
        },
```

```
        "lazy_connections": 1
    }
}
```

The explicit activation of lazy connections is for demonstration purpose only.

Example 7.46 Connection error on query execution

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (mysqli_connect_errno())
  /* Of course, your error handling is nicer... */
  die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));

/* Connection 1, connection bound SQL user variable, no SELECT thus run on master */
if (!$mysqli->query("SET @myrole='master'")) {
 printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}

/* Connection 2, run on slave because SELECT, provoke connection error */
if (!($res = $mysqli->query("SELECT @myrole AS _role"))) {
 printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
} else {
 $row = $res->fetch_assoc();
 $res->close();
 printf("@myrole = '%s'\n", $row['_role']);
}
$mysqli->close();
?>
```

The above example will output something similar to:

```
PHP Warning:  mysqli::query(): php_network_getaddresses: getaddrinfo failed: Name or service not known in %s o
PHP Warning:  mysqli::query(): [2002] php_network_getaddresses: getaddrinfo failed: Name or service not known
[2002] php_network_getaddresses: getaddrinfo failed: Name or service not known
```

Applications are expected to handle possible connection errors by implementing proper error handling.

Depending on the use case, applications may want to handle connection errors differently from other errors. Typical connection errors are `2002 (CR_CONNECTION_ERROR) - Can't connect to local MySQL server through socket '%s' (%d)`, `2003 (CR_CONN_HOST_ERROR) - Can't connect to MySQL server on '%s' (%d)` and `2005 (CR_UNKNOWN_HOST) - Unknown MySQL server host '%s' (%d)`. For example, the application may test for the error codes and manually perform a fail over. The plugins philosophy is not to offer automatic fail over, beyond master fail over, because fail over is not a transparent operation.

Example 7.47 Provoking a connection error

```
{
    "myapp": {
        "master": {
            "master_0": {
```

```
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "invalid_host_name"
            },
            "slave_1": {
                "host": "192.168.78.136"
            }
        },
        "lazy_connections": 1,
        "filters": {
            "roundrobin": [

            ]
        }
    }
}
```

Explicitly activating lazy connections is done for demonstration purposes, as is round robin load balancing as opposed to the default `random once` type.

Example 7.48 Most basic failover

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (mysqli_connect_errno())
  /* Of course, your error handling is nicer... */
  die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));

/* Connection 1, connection bound SQL user variable, no SELECT thus run on master */
if (!$mysqli->query("SET @myrole='master'")) {
 printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}

/* Connection 2, first slave */
$res = $mysqli->query("SELECT VERSION() AS _version");
/* Hackish manual fail over */
if (2002 == $mysqli->errno || 2003 == $mysqli->errno || 2004 == $mysqli->errno) {
  /* Connection 3, first slave connection failed, trying next slave */
  $res = $mysqli->query("SELECT VERSION() AS _version");
}

if (!$res) {
  printf("ERROR, [%d] '%s'\n", $mysqli->errno, $mysqli->error);
} else {
 /* Error messages are taken from connection 3, thus no error */
 printf("SUCCESS, [%d] '%s'\n", $mysqli->errno, $mysqli->error);
 $row = $res->fetch_assoc();
 $res->close();
 printf("version = %s\n", $row['_version']);
}
$mysqli->close();
?>
```

The above example will output something similar to:

```

```

```
[1045] Access denied for user 'username'@'localhost' (using password: YES)
PHP Warning:  mysqli::query(): php_network_getaddresses: getaddrinfo failed: Name or service not known in %s o
PHP Warning:  mysqli::query(): [2002] php_network_getaddresses: getaddrinfo failed: Name or service not known
SUCCESS, [0] ''
version = 5.6.2-m5-log
```

In some cases, it may not be easily possible to retrieve all errors that occur on all network connections through a connection handle. For example, let's assume a connection handle represents a pool of three open connections. One connection to a master and two connections to the slaves. The application changes the current database using the user API call `mysqli_select_db`, which then calls the mysqlnd library function to change the schemata. mysqlnd_ms monitors the function, and tries to change the current database on all connections to harmonize their state. Now, assume the master succeeds in changing the database, and both slaves fail. Upon the initial error from the first slave, the plugin will set an appropriate error on the connection handle. The same is done when the second slave fails to change the database. The error message from the first slave is lost.

Such cases can be debugged by either checking for errors of the type `E_WARNING` (see above) or, if no other option, investigation of the mysqlnd_ms debug and trace log.

7.5.5 Transient errors

Some distributed database clusters make use of transient errors. A transient error is a temporary error that is likely to disappear soon. By definition it is safe for a client to ignore a transient error and retry the failed operation on the same database server. The retry is free of side effects. Clients are not forced to abort their work or to fail over to another database server immediately. They may enter a retry loop before to wait for the error to disappear before giving up on the database server. Transient errors can be seen, for example, when using MySQL Cluster. But they are not bound to any specific clustering solution per se.

`PECL/mysqlnd_ms` can perform an automatic retry loop in case of a transient error. This increases distribution transparency and thus makes it easier to migrate an application running on a single database server to run on a cluster of database servers without having to change the source of the application.

The automatic retry loop will repeat the requested operation up to a user configurable number of times and pause between the attempts for a configurable amount of time. If the error disappears during the loop, the application will never see it. If not, the error is forwarded to the application for handling.

In the example below a duplicate key error is provoked to make the plugin retry the failing query two times before the error is passed to the application. Between the two attempts the plugin sleeps for 100 milliseconds.

Example 7.49 Provoking a transient error

```
mysqlnd_ms.enable=1
mysqlnd_ms.collect_statistics=1
```

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
```

```
            }
         },
         "slave": {
             "slave_0": {
                 "host": "192.168.78.136",
                 "port": "3306"
             }
       },
       "transient_error": {
          "mysql_error_codes": [
            1062
          ],
          "max_retries": 2,
          "usleep_retry": 100
       }
    }
}
```

Example 7.50 Transient error retry loop

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
if (mysqli_connect_errno())
  /* Of course, your error handling is nicer... */
  die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT PRIMARY KEY)") ||
    !$mysqli->query("INSERT INTO test(id) VALUES (1)")) {
  printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}

/* Retry loop is completely transparent. Checking statistics is
 the only way to know about implicit retries */
$stats = mysqlnd_ms_get_stats();
printf("Transient error retries before error: %d\n", $stats['transient_error_retries']);

/* Provoking duplicate key error to see statistics change */
if (!$mysqli->query("INSERT INTO test(id) VALUES (1)")) {
  printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
}

$stats = mysqlnd_ms_get_stats();
printf("Transient error retries after error: %d\n", $stats['transient_error_retries']);

$mysqli->close();
?>
```

The above example will output something similar to:

```
Transient error retries before error: 0
[1062] Duplicate entry '1' for key 'PRIMARY'
Transient error retries before error: 2
```

Because the execution of the retry loop is transparent from a users point of view, the example checks the statistics provided by the plugin to learn about it.

As the example shows, the plugin can be instructed to consider any error transient regardless of the database servers error semantics. The only error that a stock MySQL server considers temporary has the error code `1297`. When configuring other error codes but `1297` make sure your configuration reflects the semantics of your clusters error codes.

The following mysqlnd C API calls are monitored by the plugin to check for transient errors: `query()`, `change_user()`, `select_db()`, `set_charset()`, `set_server_option()` `prepare()`, `execute()`, `set_autocommit()`, `tx_begin()`, `tx_commit()`, `tx_rollback()`, `tx_commit_or_rollback()`. The corresponding user API calls have similar names.

The maximum time the plugin may sleep during the retry loop depends on the function in question. The a retry loop for `query()`, `prepare()` or `execute()` will sleep for up to `max_retries * usleep_retry` milliseconds.

However, functions that control connection state are dispatched to all connections. The retry loop settings are applied to every connection on which the command is to be run. Thus, such a function may interrupt program execution for longer than a function that is run on one server only. For example, `set_autocommit()` is dispatched to connections and may sleep up to `(max_retries * usleep_retry) * number_of_open_connections)` milliseconds. Please, keep this in mind when setting long sleep times and large retry numbers. Using the default settings of `max_retries=1`, `usleep_retry=100` and `lazy_connections=1` it is unlikely that you will ever see a delay of more than 1 second.

7.5.6 Failover

By default, connection failover handling is left to the user. The application is responsible for checking return values of the database functions it calls and reacting to possible errors. If, for example, the plugin recognizes a query as a read-only query to be sent to the slave servers, and the slave server selected by the plugin is not available, the plugin will raise an error after not executing the statement.

Default: manual failover

It is up to the application to handle the error and, if required, re-issue the query to trigger the selection of another slave server for statement execution. The plugin will make no attempts to failover automatically, because the plugin cannot ensure that an automatic failover will not change the state of the connection. For example, the application may have issued a query which depends on SQL user variables which are bound to a specific connection. Such a query might return incorrect results if the plugin would switch the connection implicitly as part of automatic failover. To ensure correct results, the application must take care of the failover, and rebuild the required connection state. Therefore, by default, no automatic failover is performed by the plugin.

A user that does not change the connection state after opening a connection may activate automatic failover. Please note, that automatic failover logic is limited to connection attempts. Automatic failover is not used for already established connections. There is no way to instruct the plugin to attempt failover on a connection that has been connected to MySQL already in the past.

Automatic failover

The failover policy is configured in the plugins configuration file, by using the failover configuration directive.

Automatic and silent failover can be enabled through the failover configuration directive. Automatic failover can either be configured to try exactly one master after a slave failure or, alternatively, loop over slaves

and masters before returning an error to the user. The number of connection attempts can be limited and failed hosts can be excluded from future load balancing attempts. Limiting the number of retries and remembering failed hosts are considered experimental features, albeit being reasonable stable. Syntax and semantics may change in future versions.

Please note, since version 1.5.0 automatic failover is disabled for the duration of a transaction if transaction stickiness is enabled and transaction boundaries have been detected. The plugin will not switch connections for the duration of a transaction. It will also not perform automatic and silent failover. Instead an error will be thrown. It is then left to the user to handle the failure of the transaction. Please check, the `trx_stickiness` documentation how to do this.

A basic manual failover example is provided within the error handling section.

Standby servers

Using weighted load balancing, introduced in PECL/mysqlnd 1.4.0, it is possible to configure standby servers that are sparsely used during normal operations. A standby server that is primarily used as a worst-case standby failover target can be assigned a very low weight/priority in relation to all other servers. As long as all servers are up and running the majority of the workload is assigned to the servers which have hight weight values. Few requests will be directed to the standby system which has a very low weight value.

Upon failure of the servers with a high priority, you can still failover to the standby, which has been given a low load balancing priority by assigning a low weight to it. Failover can be some manually or automatically. If done automatically, you may want to combine it with the `remember_failed` option.

At this point, it is not possible to instruct the load balancer to direct no requests at all to a standby. This may not be much of a limitation given that the highest weight you can assign to a server is 65535. Given two slaves, of which one shall act as a standby and has been assigned a weight of 1, the standby will have to handle far less than one percent of the overall workload.

Failover and primary copy

Please note, if using a primary copy cluster, such as MySQL Replication, it is difficult to do connection failover in case of a master failure. At any time there is only one master in the cluster for a given dataset. The master is a single point of failure. If the master fails, clients have no target to fail over write requests. In case of a master outage the database administrator must take care of the situation and update the client configurations, if need be.

7.5.7 Load balancing

Four load balancing strategies are supported to distribute statements over the configured MySQL slave servers:

random	Chooses a random server whenever a statement is executed.
random once (default)	Chooses a random server after the first statement is executed, and uses the decision for the rest of the PHP request. It is the default, and the lowest impact on the connection state.
round robin	Iterates over the list of configured servers.
user-defined via callback	Is used to implement any other strategy.

The load balancing policy is configured in the plugins configuration file using the random, roundrobin, and user filters.

Servers can be prioritized assigning a weight. A server that has been given a weight of two will get twice as many requests as a server that has been given the default weight of one. Prioritization can be handy in heterogenous environments. For example, you may want to assign more requests to a powerful machine than to a less powerful. Or, you may have configured servers that are close or far from the client, thus expose different latencies.

7.5.8 Read-write splitting

The plugin executes read-only statements on the configured MySQL slaves, and all other queries on the MySQL master. Statements are considered read-only if they either start with `SELECT`, the SQL hint `/*ms=slave*/`, or if a slave had been chosen for running the previous query and the query starts with the SQL hint `/*ms=last_used*/`. In all other cases, the query will be sent to the MySQL replication master server. It is recommended to use the constants `MYSQLND_MS_SLAVE_SWITCH`, `MYSQLND_MS_MASTER_SWITCH` and `MYSQLND_MS_LAST_USED_SWITCH` instead of `/*ms=slave*/`. See also the list of mysqlnd_ms constants.

SQL hints are a special kind of standard compliant SQL comments. The plugin does check every statement for certain SQL hints. The SQL hints are described within the mysqlnd_ms constants documentation, constants that are exported by the extension. Other systems involved with the statement processing, such as the MySQL server, SQL firewalls, and SQL proxies, are unaffected by the SQL hints, because those systems are designed to ignore SQL comments.

The built-in read-write splitter can be replaced by a user-defined filter, see also the user filter documentation.

A user-defined read-write splitter can request the built-in logic to send a statement to a specific location, by invoking `mysqlnd_ms_is_select`.

> **Note**
>
> The built-in read-write splitter is not aware of multi-statements. Multi-statements are seen as one statement. The splitter will check the beginning of the statement to decide where to run the statement. If, for example, a multi-statement begins with `SELECT 1 FROM DUAL; INSERT INTO test(id) VALUES (1); ...` the plugin will run it on a slave although the statement is not read-only.

7.5.9 Filter

> **Version requirement**
>
> Filters exist as of mysqlnd_ms version 1.1.0-beta.

filters. PHP applications that implement a MySQL replication cluster must first identify a group of servers in the cluster which could execute a statement before the statement is executed by one of the candidates. In other words: a defined list of servers must be filtered until only one server is available.

The process of filtering may include using one or more filters, and filters can be chained. And they are executed in the order they are defined in the plugins configuration file.

> **Explanation: comparing filter chaining to pipes**
>
> The concept of chained filters can be compared to using pipes to connect command line utilities on an operating system command shell. For example, an input stream is passed to a processor, filtered, and then transferred to be output. Then, the output is passed as input to the next command, which is connected to the previous using the pipe operator.

Available filters:

- Load balancing filters: random and roundrobin.
- Selection filter: user, user_multi, quality_of_service.

The `random` filter implements the 'random' and 'random once' load balancing policies. The 'round robin' load balancing can be configured through the `roundrobin` filter. Setting a 'user defined callback' for server selection is possible with the `user` filter. The `quality_of_service` filter finds cluster nodes capable of delivering a certain service, for example, read-your-writes or, not lagging more seconds behind the master than allowed.

Filters can accept parameters to change their behavior. The `random` filter accepts an optional `sticky` parameter. If set to true, the filter changes load balancing from random to random once. Random picks a random server every time a statement is to be executed. Random once picks a random server when the first statement is to be executed and uses the same server for the rest of the PHP request.

One of the biggest strength of the filter concept is the possibility to chain filters. This strength does not become immediately visible because the `random`, `roundrobin` and `user` filters are supposed to output no more than one server. If a filter reduces the list of candidates for running a statement to only one server, it makes little sense to use that one server as input for another filter for further reduction of the list of candidates.

An example filter sequence that will fail:

- Statement to be executed: `SELECT 1 FROM DUAL`. Passed to all filters.
- All configured nodes are passed as input to the first filter. Master nodes: `master_0`. Slave nodes:`slave_0`, `slave_1`
- Filter: `random`, argument `sticky=1`. Picks a random slave once to be used for the rest of the PHP request. Output: `slave_0`.
- Output of `slave_0` and the statement to be executed is passed as input to the next filter. Here: `roundrobin`, server list passed to filter is: `slave_0`.
- Filter: `roundrobin`. Server list consists of one server only, round robin will always return the same server.

If trying to use such a filter sequence, the plugin may emit a warning like `(mysqlnd_ms) Error while creating filter '%s' . Non-multi filter '%s' already created. Stopping in %s on line %d`. Furthermore, an appropriate error on the connection handle may be set.

A second type of filter exists: multi filter. A multi filter emits zero, one or multiple servers after processing. The `quality_of_service` filter is an example. If the service quality requested sets an upper limit for the slave lag and more than one slave is lagging behind less than the allowed number of seconds, the filter returns more than one cluster node. A multi filter must be followed by other to further reduce the list of candidates for statement execution until a candidate is found.

A filter sequence with the `quality_of_service` multi filter followed by a load balancing filter.

- Statement to be executed: `SELECT sum(price) FROM orders WHERE order_id = 1`. Passed to all filters.
- All configured nodes are passed as input to the first filter. Master nodes: `master_0`. Slave nodes: `slave_0`, `slave_1`, `slave_2`, `slave_3`
- Filter: `quality_of_service`, rule set: session_consistency (read-your-writes) Output: `master_0`
- Output of `master_0` and the statement to be executed is passed as input to the next filter, which is `roundrobin`.
- Filter: `roundrobin`. Server list consists of one server. Round robin selects `master_0`.

A filter sequence must not end with a multi filter. If trying to use a filter sequence which ends with a multi filter the plugin may emit a warning like `(mysqlnd_ms) Error in configuration. Last filter is multi filter. Needs to be non-multi one. Stopping in %s on line %d`. Furthermore, an appropriate error on the connection handle may be set.

> **Speculation towards the future: MySQL replication filtering**
>
> In future versions, there may be additional multi filters. For example, there may be a `table` filter to support MySQL replication filtering. This would allow you to define rules for which database or table is to be replicated to which node of a replication cluster. Assume your replication cluster consists of four slaves (`slave_0`, `slave_1`, `slave_2`, `slave_3`) two of which replicate a database named `sales` (`slave_0`, `slave_1`). If the application queries the database `slaves`, the hypothetical `table` filter reduces the list of possible servers to `slave_0` and `slave_1`. Because the output and list of candidates consists of more than one server, it is necessary and possible to add additional filters to the candidate list, for example, using a load balancing filter to identify a server for statement execution.

7.5.10 Service level and consistency

> **Version requirement**
>
> Service levels have been introduced in mysqlnd_ms version 1.2.0-alpha. `mysqlnd_ms_set_qos` requires PHP 5.4.0 or newer.

The plugin can be used with different kinds of MySQL database clusters. Different clusters can deliver different levels of service to applications. The service levels can be grouped by the data consistency levels that can be achieved. The plugin knows about:

- eventual consistency
- session consistency
- strong consistency

Depending how a cluster is used it may be possible to achieve higher service levels than the default one. For example, a read from an asynchronous MySQL replication slave is eventual consistent. Thus, one may say the default consistency level of a MySQL replication cluster is eventual consistency. However, if the master only is used by a client for reading and writing during a session, session consistency (read your

writes) is given. PECL mysqlnd 1.2.0 abstracts the details of choosing an appropriate node for any of the above service levels from the user.

Service levels can be set through the qualify-of-service filter in the plugins configuration file and at runtime using the function `mysqlnd_ms_set_qos`.

The plugin defines the different service levels as follows.

Eventual consistency is the default service provided by an asynchronous cluster, such as classical MySQL replication. A read operation executed on an arbitrary node may or may not return stale data. The applications view of the data is eventual consistent.

Session consistency is given if a client can always read its own writes. An asynchronous MySQL replication cluster can deliver session consistency if clients always use the master after the first write or never query a slave which has not yet replicated the clients write operation.

The plugins understanding of strong consistency is that all clients always see the committed writes of all other clients. This is the default when using MySQL Cluster or any other cluster offering synchronous data distribution.

Service level parameters

Eventual consistency and session consistency service level accept parameters.

Eventual consistency is the service provided by classical MySQL replication. By default, all nodes qualify for read requests. An optional `age` parameter can be given to filter out nodes which lag more than a certain number of seconds behind the master. The plugin is using `SHOW SLAVE STATUS` to measure the lag. Please, see the MySQL reference manual to learn about accuracy and reliability of the `SHOW SLAVE STATUS` command.

Session consistency (read your writes) accepts an optional `GTID` parameter to consider reading not only from the master but also from slaves which already have replicated a certain write described by its transaction identifier. This way, when using asynchronous MySQL replication, read requests may be load balanced over slaves while still ensuring session consistency.

The latter requires the use of client-side global transaction id injection.

Advantages of the new approach

The new approach supersedes the use of SQL hints and the configuration option `master_on_write` in some respects. If an application running on top of an asynchronous MySQL replication cluster cannot accept stale data for certain reads, it is easier to tell the plugin to choose appropriate nodes than prefixing all read statements in question with the SQL hint to enforce the use of the master. Furthermore, the plugin may be able to use selected slaves for reading.

The `master_on_write` configuration option makes the plugin use the master after the first write (session consistency, read your writes). In some cases, session consistency may not be needed for the rest of the session but only for some, few read operations. Thus, `master_on_write` may result in more read load on the master than necessary. In those cases it is better to request a higher than default service level only for those reads that actually need it. Once the reads are done, the application can return to default service level. Switching between service levels is only possible using `mysqlnd_ms_set_qos`.

Performance considerations

A MySQL replication cluster cannot tell clients which slaves are capable of delivering which level of service. Thus, in some cases, clients need to query the slaves to check their status. PECL mysqlnd_ms transparently runs the necessary SQL in the background. However, this is an expensive and slow

operation. SQL statements are run if eventual consistency is combined with an age (slave lag) limit and if session consistency is combined with a global transaction ID.

If eventual consistency is combined with an maximum age (slave lag), the plugin selects candidates for statement execution and load balancing for each statement as follows. If the statement is a write all masters are considered as candidates. Slaves are not checked and not considered as candidates. If the statement is a read, the plugin transparently executes `SHOW SLAVE STATUS` on every slaves connection. It will loop over all connections, send the statement and then start checking for results. Usually, this is slightly faster than a loop over all connections in which for every connection a query is send and the plugin waits for its results. A slave is considered a candidate if `SHOW SLAVE STATUS` reports `Slave_IO_Running=Yes`, `Slave_SQL_Running=Yes` and `Seconds_Behind_Master` is less or equal than the allowed maximum age. In case of an SQL error, the plugin emits a warning but does not set an error on the connection. The error is not set to make it possible to use the plugin as a drop-in.

If session consistency is combined with a global transaction ID, the plugin executes the SQL statement set with the `fetch_last_gtid` entry of the `global_transaction_id_injection` section from the plugins configuration file. Further details are identical to those described above.

In version 1.2.0 no additional optimizations are done for executing background queries. Future versions may contain optimizations, depending on user demand.

If no parameters and options are set, no SQL is needed. In that case, the plugin consider all nodes of the type shown below.

- eventual consistency, no further options set: all masters, all slaves
- session consistency, no further options set: all masters
- strong consistency (no options allowed): all masters

Throttling

The quality of service filter can be combined with Global transaction IDs to throttle clients. Throttling does reduce the write load on the master by slowing down clients. If session consistency is requested and global transactions identifier are used to check the status of a slave, the check can be done in two ways. By default a slave is checked and skipped immediately if it does not match the criteria for session consistency. Alternatively, the plugin can wait for a slave to catch up to the master until session consistency is possible. To enable the throttling, you have to set wait_for_gtid_timeout configuration option.

7.5.11 Global transaction IDs

> **Version requirement**
>
> Client side global transaction ID injection exists as of mysqlnd_ms version 1.2.0-alpha. Transaction boundaries are detected by monitoring API calls. This is possible as of PHP 5.4.0. Please, see also Transaction handling.
>
> As of MySQL 5.6.5-m8 the MySQL server features built-in global transaction identifiers. The MySQL built-in global transaction ID feature is supported by `PECL/mysqlnd_ms` 1.3.0-alpha or later. Neither are client-side transaction boundary monitoring nor any setup activities required if using the server feature.
>
> Please note, all MySQL 5.6 production versions do not provide clients with enough information to use GTIDs for enforcing session consistency. In the worst case, the plugin will choose the master only.

Idea and client-side emulation

`PECL/mysqlnd_ms` can do client-side transparent global transaction ID injection. In its most basic form, a global transaction identifier is a counter which is incremented for every transaction executed on the master. The counter is held in a table on the master. Slaves replicate the counter table.

In case of a master failure a database administrator can easily identify the most recent slave for promoting it as a new master. The most recent slave has the highest transaction identifier.

Application developers can ask the plugin for the global transaction identifier (GTID) for their last successful write operation. The plugin will return an identifier that refers to an transaction no older than that of the clients last write operation. Then, the GTID can be passed as a parameter to the quality of service (QoS) filter as an option for session consistency. Session consistency ensures read your writes. The filter ensures that all reads are either directed to a master or a slave which has replicated the write referenced by the GTID.

When injection is done

The plugin transparently maintains the GTID table on the master. In autocommit mode the plugin injects an `UPDATE` statement before executing the users statement for every master use. In manual transaction mode, the injection is done before the application calls `commit()` to close a transaction. The configuration option `report_error` of the GTID section in the plugins configuration file is used to control whether a failed injection shall abort the current operation or be ignored silently (default).

Please note, the PHP version requirements for transaction boundary monitoring and their limits.

Limitations

Client-side global transaction ID injection has shortcomings. The potential issues are not specific to `PECL/mysqlnd_ms` but are rather of general nature.

- Global transaction ID tables must be deployed on all masters and replicas.
- The GTID can have holes. Only PHP clients using the plugin will maintain the table. Other clients will not.
- Client-side transaction boundary detection is based on API calls only.
- Client-side transaction boundary detection does not take implicit commit into account. Some MySQL SQL statements cause an implicit commit and cannot be rolled back.

Using server-side global transaction identifier

Starting with `PECL/mysqlnd_ms` 1.3.0-alpha the MySQL 5.6.5-m8 or newer built-in global transaction identifier feature is supported. Use of the server feature lifts all of the above listed limitations. Please, see the MySQL Reference Manual for limitations and preconditions for using server built-in global transaction identifiers.

Whether to use the client-side emulation or the server built-in functionality is a question not directly related to the plugin, thus it is not discussed in depth. There are no plans to remove the client-side emulation and you can continue to use it, if the server-side solution is no option. This may be the case in heterogenous environments with old MySQL server or, if any of the server-side solution limitations is not acceptable.

From an applications perspective there is hardly a difference in using one or the other approach. The following properties differ.

- Client-side emulation, as shown in the manual, is using an easy to compare sequence number for global transactions. Multi-master is not handled to keep the manual examples easy.

Server-side built-in feature is using a combination of a server identifier and a sequence number as a global transaction identifier. Comparison cannot use numeric algebra. Instead a SQL function must be used. Please, see the MySQL Reference Manual for details.

Server-side built-in feature of MySQL 5.6 cannot be used to ensure session consistency under all circumstances. Do not use it for the quality-of-service feature. Here is a simple example why it will not give reliable results. There are more edge cases that cannot be covered with limited functionality exported by the server. Currently, clients can ask a MySQL replication master for a list of all executed global transaction IDs only. If a slave is configured not to replicate all transactions, for example, because replication filters are set, then the slave will never show the same set of executed global transaction IDs. Albeit the slave may have replicated a clients writes and it may be a candidate for a consistent read, it will never be considered by the plugin. Upon write the plugin learns from the master that the servers complete transaction history consists of GTID=1..3. There is no way for the plugin to ask for the GTID of the write transaction itself, say GTID=3. Assume that a slave does not replicate the transactions GTID=1..2 but only GTID=3 because of a replication feature. Then, the slaves transaction history is GTID=3. However, the plugin tries to find a node which has a transaction history of GITD=1...3. Albeit the slave has replicated the clients write and session consistency may be achieved when reading from the slave, it will not be considered by the plugin. This is not a fault of the plugin implementation but a feature gap on the server side. Please note, this is a trivial case to illustrate the issue there are other issues. In sum you are asked not to attempt using MySQL 5.6 built-in GTIDs for enforcing session consistency. Sooner or later the load balancing will stop working properly and the plugin will direct all session consistency requests to the master.

- Plugin global transaction ID statistics are only available with client-side emulation because they monitor the emulation.

> **Global transaction identifiers in distributed systems**
>
> Global transaction identifiers can serve multiple purposes in the context of distributed systems, such as a database cluster. Global transaction identifiers can be used for, for example, system wide identification of transactions, global ordering of transactions, heartbeat mechanism and for checking the replication status of replicas. `PECL/mysqlnd_ms`, a clientside driver based software, does focus on using GTIDs for tasks that can be handled at the client, such as checking the replication status of replicas for asynchronous replication setups.

7.5.12 Cache integration

> **Version requirement**
>
> The feature requires use of `PECL/mysqlnd_ms` 1.3.0-beta or later, and `PECL/mysqlnd_qc` 1.1.0-alpha or newer. `PECL/mysqlnd_ms` must be compiled to support the feature. PHP 5.4.0 or newer is required.

> **Setup: extension load order**
>
> `PECL/mysqlnd_ms` must be loaded before `PECL/mysqlnd_qc`, when using shared extensions.

> **Feature stability**
>
> The cache integration is of beta quality.

> **Suitable MySQL clusters**
>
> The feature is targeted for use with MySQL Replication (primary copy). Currently, no other kinds of MySQL clusters are supported. Users of such cluster must control PECL/mysqlnd_qc manually if they are interested in client-side query caching.

Support for MySQL replication clusters (asynchronous primary copy) is the main focus of `PECL/mysqlnd_ms`. The slaves of a MySQL replication cluster may or may not reflect the latest updates from the master. Slaves are asynchronous and can lag behind the master. A read from a slave is eventual consistent from a cluster-wide perspective.

The same level of consistency is offered by a local cache using time-to-live (TTL) invalidation strategy. Current data or stale data may be served. Eventually, data searched for in the cache is not available and the source of the cache needs to be accessed.

Given that both a MySQL Replication slave (asynchronous secondary) and a local TTL-driven cache deliver the same level of service it is possible to transparently replace a remote database access with a local cache access to gain better possibility.

As of `PECL/mysqlnd_ms` 1.3.0-beta the plugin is capable of transparently controlling `PECL/mysqlnd_ms` 1.1.0-alpha or newer to cache a read-only query if explicitly allowed by setting an appropriate quality of service through `mysqlnd_ms_set_qos`. P lease, see the quickstart for a code example. Both plugins must be installed, `PECL/mysqlnd_ms` must be compiled to support the cache feature and PHP 5.4.0 or newer has to be used.

Applications have full control of cache usage and can request fresh data at any time, if need be. The cache usage can be enabled and disabled time during the execution of a script. The cache will be used if `mysqlnd_ms_set_qos` sets the quality of service to eventual consistency and enables cache usage. Cache usage is disabled by requesting higher consistency levels, for example, session consistency (read your writes). Once the quality of service has been relaxed to eventual consistency the cache can be used again.

If caching is enabled for a read-only statement, `PECL/mysqlnd_ms` may inject SQL hints to control caching by PECL/mysqlnd_qc. It may modify the SQL statement it got from the application. Subsequent SQL processors are supposed to ignore the SQL hints. A SQL hint is a SQL comment. Comments must not be ignored, for example, by the database server.

The TTL of a cache entry is computed on a per statement basis. Applications set an maximum age for the data they want to retrieve using `mysqlnd_ms_set_qos`. The age sets an approximate upper limit of how many seconds the data returned may lag behind the master.

The following logic is used to compute the actual TTL if caching is enabled. The logic takes the estimated slave lag into account for choosing a TTL. If, for example, there are two slaves lagging 5 and 10 seconds behind and the maximum age allowed is 60 seconds, the TTL is set to 50 seconds. Please note, the age setting is no more than an estimated guess.

- Check whether the statement is read-only. If not, don't cache.
- If caching is enabled, check the slave lag of all configured slaves. Establish slave connections if none exist so far and lazy connections are used.
- Send `SHOW SLAVE STATUS` to all slaves. Do not wait for the first slave to reply before sending to the second slave. Clients often wait long for replies, thus we send out all requests in a burst before fetching in a second stage.
- Loop over all slaves. For every slave wait for its reply. Do not start checking another slave before the currently waited for slave has replied. Check for `Slave_IO_Running=Yes` and

`Slave_SQL_Running=Yes`. If both conditions hold true, fetch the value of `Seconds_Behind_Master`. In case of any errors or if conditions fail, set an error on the slave connection. Skip any such slave connection for the rest of connection filtering.

- Search for the maximum value of `Seconds_Behind_Master` from all slaves that passed the previous conditions. Subtract the value from the maximum age provided by the user with `mysqlnd_ms_set_qos`. Use the result as a TTL.
- The filtering may sort out all slaves. If so, the maximum age is used as TTL, because the maximum lag found equals zero. It is perfectly valid to sort out all slaves. In the following it is up to subsequent filter to decide what to do. The built-in load balancing filter will pick the master.
- Inject the appropriate SQL hints to enable caching by `PECL/mysqlnd_qc`.
- Proceed with the connection filtering, e.g. apply load balancing rules to pick a slave.
- `PECL/mysqlnd_qc` is loaded after `PECL/mysqlnd_ms` by PHP. Thus, it will see all query modifications of `PECL/mysqlnd_ms` and cache the query if instructed to do so.

The algorithm may seem expensive. `SHOW SLAVE STATUS` is a very fast operation. Given a sufficient number of requests and cache hits per second the cost of checking the slaves lag can easily outweigh the costs of the cache decision.

Suggestions on a better algorithm are always welcome.

7.5.13 Supported clusters

Any application using any kind of MySQL cluster is faced with the same tasks:

- Identify nodes capable of executing a given statement with the required service level
- Load balance requests within the list of candidates
- Automatic fail over within candidates, if needed

The plugin is optimized for fulfilling these tasks in the context of a classical asynchronous MySQL replication cluster consisting of a single master and many slaves (primary copy). When using classical, asynchronous MySQL replication all of the above listed tasks need to be mastered at the client side.

Other types of MySQL cluster may have lower requirements on the application side. For example, if all nodes in the cluster can answer read and write requests, no read-write splitting needs to be done (multi-master, update-all). If all nodes in the cluster are synchronous, they automatically provide the highest possible quality of service which makes choosing a node easier. In this case, the plugin may serve the application after some reconfiguration to disable certain features, such as built-in read-write splitting.

> **Documentation focus**
>
> The documentation focusses describing the use of the plugin with classical asynchronous MySQL replication clusters (primary copy). Support for this kind of cluster has been the original development goal. Use of other clusters is briefly described below. Please note, that this is still work in progress.

Primary copy (MySQL Replication)

This is the primary use case of the plugin. Follow the hints given in the descriptions of each feature.

- Configure one master and one or more slaves. Server configuration details are given in the setup section.
- Use random load balancing policy together with the sticky flag.
- If you do not plan to use the service level API calls, add the master on write flag.
- Please, make yourself aware of the properties of automatic failover before adding a failover directive.
- Consider the use of trx_stickiness to execute transactions on the primary only. Please, read carefully how it works before you rely on it.

Example 7.51 Enabling the plugin (php.ini)

```
mysqlnd_ms.enable=1
mysqlnd_ms.config_file=/path/to/mysqlnd_ms_plugin.ini
```

Example 7.52 Basic plugin configuration (mysqlnd_ms_plugin.ini) for MySQL Replication

```
{
  "myapp": {
    "master": {
      "master_1": {
        "host": "localhost",
        "socket": "\/tmp\/mysql57.sock"
      }
    },
    "slave": {
      "slave_0": {
        "host": "127.0.0.1",
        "port": 3308
      },
      "slave_1": {
        "host": "192.168.2.28",
        "port": 3306
      }
    },
    "filters": {
      "random": {
        "sticky": "1"
      }
    }
  }
}
```

Primary copy with multi primaries (MMM - MySQL Multi Master)

MySQL Replication allows you to create cluster topologies with multiple masters (primaries). Write-write conflicts are not handled by the replication system. This is no update anywhere setup. Thus, data must be partitioned manually and clients must redirected in accordance to the partitioning rules. The recommended setup is equal to the sharding setup below.

Manual sharding, possibly combined with primary copy and multiple primaries

Use SQL hints and the node group filter for clusters that use data partitioning but leave query redirection to the client. The example configuration shows a multi master setup with two shards.

Example 7.53 Multiple primaries - multi master (php.ini)

```
mysqlnd_ms.enable=1
mysqlnd_ms.config_file=/path/to/mysqlnd_ms_plugin.ini
mysqlnd_ms.multi_master=1
```

Example 7.54 Primary copy with multiple primaries and paritioning

```
{
  "myapp": {
    "master": {
      "master_1": {
        "host": "localhost",
        "socket": "\/tmp\/mysql57.sock"
      }
      "master_2": {
        "host": "192.168.2.27",
        "socket": "3306"
      }
    },
    "slave": {
      "slave_1": {
        "host": "127.0.0.1",
        "port": 3308
      },
      "slave_2": {
        "host": "192.168.2.28",
        "port": 3306
      }
    },
    "filters": {
      "node_groups": {
        "Partition_A" : {
          "master": ["master_1"],
          "slave": ["slave_1"]
        },
        "Partition_B" : {
          "master": ["master_2"],
          "slave": ["slave_2"]
        }
      },
      "roundrobin": []
    }
  }
}
```

The plugin can also be used with a loose collection of unrelated shards. For such a cluster, configure masters only and disable read write splitting. The nodes of such a cluster are called masters in the plugin configuration as they accept both reads and writes for their partition.

Using synchronous update everywhere clusters such as MySQL Cluster

MySQL Cluster is a synchronous cluster solution. All cluster nodes accept read and write requests. In the context of the plugin, all nodes shall be considered as masters.

Use the load balancing and fail over features only.

- Disable the plugins built-in read-write splitting.
- Configure masters only.
- Consider random once load balancing strategy, which is the plugins default. If random once is used, only masters are configured and no SQL hints are used to force using a certain node, no connection switches will happen for the duration of a web request. Thus, no special handling is required for transactions. The plugin will pick one master at the beginning of the PHP script and use it until the script terminates.
- Do not set the quality of service. All nodes have all the data. This automatically gives you the highest possible service quality (strong consistency).
- Do not enable client-side global transaction injection. It is neither required to help with server-side fail over nor to assist the quality of service filter choosing an appropriate node.

Disabling built-in read-write splitting.

- Set `mysqlnd_ms.disable_rw_split=1`
- Do not use SQL hints to enforce the use of slaves

Configure masters only.

- Set `mysqlnd_ms.multi_master=1`.
- Do not configure any slaves.
- Set `failover=loop_before_master` in the plugins configuration file to avoid warnings about the empty slave list and to make the failover logic loop over all configured masters before emitting an error.

  Please, note the warnings about automatic failover given in the previous sections.

Example 7.55 Multiple primaries - multi master (php.ini)

```
mysqlnd_ms.enable=1
mysqlnd_ms.config_file=/path/to/mysqlnd_ms_plugin.ini
mysqlnd_ms.multi_master=1
mysqlnd_ms.disable_rw_split=1
```

Example 7.56 Synchronous update anywhere cluster

```
"myapp": {
  "master": {
    "master_1": {
      "host": "localhost",
      "socket": "\/tmp\/mysql57.sock"
    },
    "master_2": {
      "host": "192.168.2.28",
      "port": 3306
    }
  },
  "slave": {
  },
```

```
    "filters": {
      "roundrobin": {
      }
    },
    "failover": {
      "strategy": "loop_before_master",
      "remember_failed": true
    }
  }
}
```

If running an update everywhere cluster that has no built-in partitioning to avoid hot spots and high collision rates, consider using the node groups filter to keep updates on a frequently accessed table on one of the nodes. This may help to reduce collision rates and thus improve performance.

7.5.14 XA/Distributed transactions

> **Version requirement**
>
> XA related functions have been introduced in `PECL/mysqlnd_ms` version 1.6.0-alpha.

> **Early adaptors wanted**
>
> The feature is currently under development. There may be issues and/or feature limitations. Do not use in production environments, although early lab tests indicate reasonable quality.
>
> Please, contact the development team if you are interested in this feature. We are looking for real life feedback to complement the feature.
>
> Below is a list of some feature restrictions.
>
> - The feature is not yet compatible with the MySQL Fabric support . This limitation is soon to be lifted.
>
> XA transaction identifier are currently restricted to numbers. This limitation will be lifted upon request, it is a simplification used during the initial implementation.

> **MySQL server restrictions**
>
> The XA support by the MySQL server has some restrictions. Most noteably, the servers binary log may lack changes made by XA transactions in case of certain errors. Please, see the MySQL manual for details.

XA/Distributed transactions can spawn multiple MySQL servers. Thus, they may seem like a perfect tool for sharded MySQL clusters, for example, clusters managed with MySQL Fabric. `PECL/mysqlnd_ms` hides most of the SQL commands to control XA transactions and performs automatic administrative tasks in cases of errors, to provide the user with a comprehensive API. Users should setup the plugin carefully and be well aware of server restrictions prior to using the feature.

Example 7.57 General pattern for XA transactions

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");

/* BEGIN */
mysqlnd_ms_xa_begin($mysqli, 1 /* xa id */);

/* run queries on various servers */
$mysqli->query("UPDATE some_table SET col_a = 1");
...

/* COMMIT */
mysqlnd_ms_xa_commit($link, 1);
?>
```

XA transactions use the two-phase commit protocol. The two-phase commit protocol is a blocking protocol. During the first phase participating servers begin a transaction and the client carries out its work. This phase is followed by a second voting phase. During voting, the servers first make a firm promise that they are ready to commit the work even in case of their possible unexpected failure. Should a server crash in this phase, it will still recall the aborted transaction after recover and wait for the client to decide on whether it shall be committed or rolled back.

Should a client that has initiated a global transaction crash after all the participating servers gave their promise to be ready to commit, then the servers must wait for a decision. The servers are not allowed to unilaterally decide on the transaction.

A client crash or disconnect from a participant, a server crash or server error during the fist phase of the protocol is uncritical. In most cases, the server will forget about the XA transaction and its work is rolled back. Additionally, the plugin tries to reach out to as many participants as it can to instruct the server to roll back the work immediately. It is not possible to disable this implicit rollback carried out by `PECL/mysqlnd_ms` in case of errors during the first phase of the protocol. This design decision has been made to keep the implementation simple.

An error during the second phase of the commit protocol can develop into a more severe situation. The servers will not forget about prepared but unfinished transactions in all cases. The plugin will not attempt to solve these cases immediately but waits for optional background garbage collection to ensure progress of the commit protocol. It is assumed that a solution will take significant time as it may include waiting for a participating server to recover from a crash. This time span may be longer than a developer and end user expects when trying to commit a global transaction with `mysqlnd_ms_xa_commit`. Thus, the function returns with the unfinished global transaction still requiring attention. Please, be warned that at this point, it is not yet clear whether the global transaction will be committed or rolled back later on.

Errors during the second phase can be ignored, handled by yourself or solved by the build-int garbage collection logic. Ignoring them is not recommended as you may experience unfinished global transactions on your servers that block resources virtually indefinitely. Handling the errors requires knowing the participants, checking their state and issuing appropriate SQL commands on them. There are no user API calls to expose this very information. You will have to configure a state store and make the plugin record its actions in it to receive the desired facts.

Please, see the quickstart and related plugin configuration file settings for an example how to configure a state. In addition to configuring a state store, you have to setup some SQL tables. The table definitions are given in the description of the plugin configuration settings.

Setting up and configuring a state store is also a precondition for using the built-in garbage collection for XA transactions that fail during the second commit phase. Recording information about ongoing XA transactions is an unavoidable extra task. The extra task consists of updating the state store after each

and every operation that changes the state of the global transaction itself (started, committed, rolled back, errors and aborts), the addition of participants (host, optionally user and password required to connect) and any changes to a participants state. Please note, depending on configuration and your security policies, these recordings may be considered sensitive. It is therefore recommended to restrict access to the state store. Unless the state store itself becomes overloaded, writing the state information may contribute noteworthy to the runtime but should overall be only a minor factor.

It is possible that the effort it takes to implement your own routines for handling XA transactions that failed during the second commit phase exceeds the benefits of using the XA feature of `PECL/mysqlnd_ms` in the first place. Thus, the manual focussed on using the built-on garbage collection only.

Garbage collection can be triggered manually or automatically in the background. You may want to call `mysqlnd_ms_xa_gc` immediately after a commit failure to attempt to solve any failed but still open global transactions as soon as possible. You may also decide to disable the automatic background garbage collection, implement your own rule set for invoking the built-in garbage collection and trigger it when desired.

By default the plugin will start the garbage collection with a certain probability in the extensions internal `RSHUTDOWN` method. The request shutdown is called after your script finished. Whether the garbage collection will be triggered is determined by computing a random value between `1...1000` and comparing it with the configuration setting `probability` (default: 5). If the setting is greater or equal to the random value, the garbage collection will be triggered.

Once started, the garbage collection acts upon up to `max_transactions_per_run` (default: 100) global transactions recorded. Records include successfully finished but also unfinished XA transactions. Records for successful transactions are removed and unfinished transactions are attempted to be solved. There are no statistics that help you finding the right balance between keeping garbage collection runs short by limiting the number of transactions considered per run and preventing the garbage collection to fall behind, resulting in many records.

For each failed XA transaction the garbage collection makes `max_retries` (default: 5) attempts to finish it. After that `PECL/mysqlnd_ms` gives up. There are two possible reasons for this. Either a participating server crashed and has not become accessible again within `max_retries` invocations of the garbage collection, or there is a situation that the built-in garbage collection cannot cope with. Likely, the latter would be considered a bug. However, you can manually force more garbage collection runs calling `mysqlnd_ms_xa_gc` with the appropriate parameter set. Should even those function runs fail to solve the situation, then the problem must be solved by an operator.

The function `mysqlnd_ms_get_stats` provides some statistics on how many XA transactions have been started, committed, failed or rolled back.

7.6 Installing/Configuring

7.6.1 Requirements

`PHP 5.3.6` or newer. Some advanced functionality requires `PHP 5.4.0` or newer.

The `mysqlnd_ms` replication and load balancing plugin supports all PHP applications and all available PHP MySQL extensions (mysqli, mysql, PDO_MYSQL). The PHP MySQL extension must be configured to use mysqlnd in order to be able to use the `mysqlnd_ms` plugin for mysqlnd.

7.6.2 Installation

This PECL extension is not bundled with PHP.

Information for installing this PECL extension may be found in the manual chapter titled Installation of PECL extensions. Additional information such as new releases, downloads, source files, maintainer information, and a CHANGELOG, can be located here: http://pecl.php.net/package/mysqlnd_ms

A DLL for this PECL extension is currently unavailable. See also the building on Windows section.

7.6.3 Runtime Configuration

The behaviour of these functions is affected by settings in `php.ini`.

Table 7.1 Mysqlnd_ms Configure Options

Name	Default	Changeable	Changelog
mysqlnd_ms.enable	0	PHP_INI_SYSTEM	
mysqlnd_ms.force_config_usage	0	PHP_INI_SYSTEM	
mysqlnd_ms.ini_file	""	PHP_INI_SYSTEM	
mysqlnd_ms.config_file	""	PHP_INI_SYSTEM	
mysqlnd_ms.collect_statistics	0	PHP_INI_SYSTEM	
mysqlnd_ms.multi_master	0	PHP_INI_SYSTEM	
mysqlnd_ms.disable_rw_split	0	PHP_INI_SYSTEM	

Here's a short explanation of the configuration directives.

`mysqlnd_ms.enable` integer
: Enables or disables the plugin. If disabled, the extension will not plug into mysqlnd to proxy internal mysqlnd C API calls.

`mysqlnd_ms.force_config_usage` integer
: If enabled, the plugin checks if the host (server) parameters value of any MySQL connection attempt, matches a section name from the plugin configuration file. If not, the connection attempt is blocked.

  This setting is not only useful to restrict PHP to certain servers but also to debug configuration file problems. The configuration file validity is checked at two different stages. The first check is performed when PHP begins to handle a web request. At this point the plugin reads and decodes the configuration file. Errors thrown at this early stage in an extensions life cycle may not be shown properly to the user. Thus, the plugin buffers the errors, if any, and additionally displays them when establishing a connection to MySQL. By default a buffered startup error will emit an error of type `E_WARNING`. If `force_config_usage` is set, the error type used is `E_RECOVERABLE_ERROR`.

  Please, see also configuration file debugging notes.

`mysqlnd_ms.ini_file` string
: Plugin specific configuration file. This setting has been renamed to `mysqlnd_ms.config_file` in version 1.4.0.

mysqlnd_ms.config_file string	Plugin specific configuration file. This setting superseeds `mysqlnd_ms.ini_file` since 1.4.0.
mysqlnd_ms.collect_statistics integer	Enables or disables the collection of statistics. The collection of statistics is disabled by default for performance reasons. Statistics are returned by the function `mysqlnd_ms_get_stats`.
mysqlnd_ms.multi_master integer	Enables or disables support of MySQL multi master replication setups. Please, see also supported clusters.
mysqlnd_ms.disable_rw_split integer	Enables or disables built-in read write splitting. Controls whether load balancing and lazy connection functionality can be used independently of read write splitting. If read write splitting is disabled, only servers from the master list will be used for statement execution. All configured slave servers will be ignored. The SQL hint `MYSQLND_MS_USE_SLAVE` will not be recognized. If found, the statement will be redirected to a master. Disabling read write splitting impacts the return value of `mysqlnd_ms_query_is_select`. The function will no longer propose query execution on slave servers.

> **Multiple master servers**
>
> Setting `mysqlnd_ms.multi_master=1` allows the plugin to use multiple master servers, instead of only the first master server of the master list.
>
> Please, see also supported clusters.

7.6.4 Plugin configuration file (>=1.1.x)

The following documentation applies to PECL/mysqlnd_ms >= 1.1.0-beta. It is not valid for prior versions. For documentation covering earlier versions, see the configuration documentation for mysqlnd_ms 1.0.x and below.

7.6.4.1 Introduction

> **Changelog: Feature was added in PECL/mysqlnd_ms 1.1.0-beta**
>
> The below description applies to PECL/mysqlnd_ms >= 1.1.0-beta. It is not valid for prior versions.

The plugin uses its own configuration file. The configuration file holds information about the MySQL replication master server, the MySQL replication slave servers, the server pick (load balancing) policy, the failover strategy, and the use of lazy connections.

The plugin loads its configuration file at the beginning of a web request. It is then cached in memory and used for the duration of the web request. This way, there is no need to restart PHP after deploying the configuration file. Configuration file changes will become active almost instantly.

The PHP configuration directive `mysqlnd_ms.config_file` is used to set the plugins configuration file. Please note, that the PHP configuration directive may not be evaluated for every web request. Therefore, changing the plugins configuration file name or location may require a PHP restart. However, no restart is required to read changes if an already existing plugin configuration file is updated.

Using and parsing JSON is efficient, and using JSON makes it easier to express hierarchical data structures than the standard `php.ini` format.

Example 7.58 Converting a PHP array (hash) into JSON format

Or alternatively, a developer may be more familiar with the PHP array syntax, and prefer it. This example demonstrates how a developer might convert a PHP array to JSON.

```
<?php
$config = array(
  "myapp" => array(
    "master" => array(
      "master_0" => array(
        "host"   => "localhost",
        "socket" => "/tmp/mysql.sock",
      ),
    ),
    "slave" => array(),
  ),
);

file_put_contents("mysqlnd_ms.ini", json_encode($config, JSON_PRETTY_PRINT));
printf("mysqlnd_ms.ini file created...\n");
printf("Dumping file contents...\n");
printf("%s\n", str_repeat("-", 80));
echo file_get_contents("mysqlnd_ms.ini");
printf("\n%s\n", str_repeat("-", 80));
?>
```

The above example will output:

```
mysqlnd_ms.ini file created...
Dumping file contents...
--------------------------------------------------------------------------------
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost",
                "socket": "\/tmp\/mysql.sock"
            }
        },
        "slave": [

        ]
    }
}
--------------------------------------------------------------------------------
```

A plugin configuration file consists of one or more sections. Sections are represented by the top-level object properties of the object encoded in the JSON file. Sections could also be called *configuration names*.

Applications reference sections by their name. Applications use section names as the host (server) parameter to the various connect methods of the mysqli, mysql and PDO_MYSQL extensions. Upon connect, the mysqlnd plugin compares the hostname with all of the section names from the plugin configuration file. If the hostname and section name match, then the plugin will load the settings for that section.

Example 7.59 Using section names example

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.2.27"
            },
            "slave_1": {
                "host": "192.168.2.27",
                "port": 3306
            }
        }
    },
    "localhost": {
        "master": [
            {
                "host": "localhost",
                "socket": "\/path\/to\/mysql.sock"
            }
        ],
        "slave": [
            {
                "host": "192.168.3.24",
                "port": "3305"
            },
            {
                "host": "192.168.3.65",
                "port": "3309"
            }
        ]
    }
}
```

```
<?php
/* All of the following connections will be load balanced */
$mysqli = new mysqli("myapp", "username", "password", "database");
$pdo = new PDO('mysql:host=myapp;dbname=database', 'username', 'password');
$mysql = mysql_connect("myapp", "username", "password");

$mysqli = new mysqli("localhost", "username", "password", "database");
?>
```

Section names are strings. It is valid to use a section name such as `192.168.2.1`, `127.0.0.1` or `localhost`. If, for example, an application connects to `localhost` and a plugin configuration section `localhost` exists, the semantics of the connect operation are changed. The application will no longer only

use the MySQL server running on the host `localhost`, but the plugin will start to load balance MySQL queries following the rules from the `localhost` configuration section. This way you can load balance queries from an application without changing the applications source code. Please keep in mind, that such a configuration may not contribute to overall readability of your applications source code. Using section names that can be mixed up with host names should be seen as a last resort.

Each configuration section contains, at a minimum, a list of master servers and a list of slave servers. The master list is configured with the keyword `master`, while the slave list is configured with the `slave` keyword. Failing to provide a slave list will result in a fatal `E_ERROR` level error, although a slave list may be empty. It is possible to allow no slaves. However, this is only recommended with synchronous clusters, please see also supported clusters. The main part of the documentation focusses on the use of asynchronous MySQL replication clusters.

The master and slave server lists can be optionally indexed by symbolic names for the servers they describe. Alternatively, an array of descriptions for slave and master servers may be used.

Example 7.60 List of anonymous slaves

```
"slave": [
    {
        "host": "192.168.3.24",
        "port": "3305"
    },
    {
        "host": "192.168.3.65",
        "port": "3309"
    }
]
```

An anonymous server list is encoded by the `JSON array` type. Optionally, symbolic names may be used for indexing the slave or master servers of a server list, and done so using the `JSON object` type.

Example 7.61 Master list using symbolic names

```
"master": {
    "master_0": {
        "host": "localhost"
    }
}
```

It is recommended to index the server lists with symbolic server names. The alias names will be shown in error messages.

The order of servers is preserved and taken into account by mysqlnd_ms. If, for example, you configure round robin load balancing strategy, the first `SELECT` statement will be executed on the slave that appears first in the slave server list.

A configured server can be described with the `host`, `port`, `socket`, `db`, `user`, `password` and `connect_flags`. It is mandatory to set the database server host using the `host` keyword. All other settings are optional.

Example 7.62 Keywords to configure a server

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "db_server_host",
                "port": "db_server_port",
                "socket": "db_server_socket",
                "db": "database_resp_schema",
                "user": "user",
                "password": "password",
                "connect_flags": 0
            }
        },
        "slave": {
            "slave_0": {
                "host": "db_server_host",
                "port": "db_server_port",
                "socket": "db_server_socket"
            }
        }
    }
}
```

If a setting is omitted, the plugin will use the value provided by the user API call used to open a connection. Please, see the using section names example above.

The configuration file format has been changed in version 1.1.0-beta to allow for chained filters. Filters are responsible for filtering the configured list of servers to identify a server for execution of a given statement. Filters are configured with the `filter` keyword. Filters are executed by mysqlnd_ms in the order of their appearance. Defining filters is optional. A configuration section in the plugins configuration file does not need to have a `filters` entry.

Filters replace the `pick[]` setting from prior versions. The new `random` and `roundrobin` provide the same functionality.

Example 7.63 New `roundrobin` filter, old functionality

```
  {
   "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.78.136",
                "port": "3306"
            },
            "slave_1": {
                "host": "192.168.78.137",
                "port": "3306"
            }
        },
        "filters": {
            "roundrobin": [

            ]
        }
    }
```

```
}
```

The function `mysqlnd_ms_set_user_pick_server` has been removed. Setting a callback is now done with the `user` filter. Some filters accept parameters. The `user` filter requires and accepts a mandatory `callback` parameter to set the callback previously set through the function `mysqlnd_ms_set_user_pick_server`.

Example 7.64 The `user` filter replaces `mysqlnd_ms_set_user_pick_server`

```
"filters": {
    "user": {
        "callback": "pick_server"
    }
}
```

The validity of the configuration file is checked both when reading the configuration file and later when establishing a connection. The configuration file is read during PHP request startup. At this early stage a PHP extension may not display error messages properly. In the worst case, no error is shown and a connection attempt fails without an adequate error message. This problem has been cured in version 1.5.0.

Example 7.65 Common error message in case of configuration file issues (upto version 1.5.0)

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
?>
```

The above example will output:

```
Warning: mysqli::mysqli(): (mysqlnd_ms) (mysqlnd_ms) Failed to parse config file [s1.json]. Please, verify

Warning: mysqli::mysqli(): (HY000/2002): php_network_getaddresses: getaddrinfo failed: Name or service not

Warning: mysqli::query(): Couldn't fetch mysqli in Command line code on line 1

Fatal error: Call to a member function fetch_assoc() on a non-object in Command line code on line 1
```

Since version 1.5.0 startup errors are additionally buffered and emitted when a connection attempt is made. Use the configuration directive `mysqlnd_ms.force_config_usage` to set the error type used to display buffered errors. By default an error of type `E_WARNING` will be emitted.

Example 7.66 Improved configuration file validation since 1.5.0

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
?>
```

```
```

The above example will output:

```
Warning: mysqli::mysqli(): (mysqlnd_ms) (mysqlnd_ms) Failed to parse config file [s1.json]. Please, verify the
```

It can be useful to set `mysqlnd_ms.force_config_usage = 1` when debugging potential configuration file errors. This will not only turn the type of buffered startup errors into `E_RECOVERABLE_ERROR` but also help detecting misspelled section names.

Example 7.67 Possibly more precise error due to `mysqlnd_ms.force_config_usage=1`

```
mysqlnd_ms.force_config_usage=1
```

```
<?php
$mysqli = new mysqli("invalid_section", "username", "password", "database");
?>
```

The above example will output:

```
Warning: mysqli::mysqli(): (mysqlnd_ms) Exclusive usage of configuration enforced but did not find the correct
```

7.6.4.2 Configuration Directives

Here is a short explanation of the configuration directives that can be used.

`master` array or object

List of MySQL replication master servers. The list of either of the `JSON type array` to declare an anonymous list of servers or of the `JSON type object`. Please, see above for examples.

Setting at least one master server is mandatory. The plugin will issue an error of type `E_ERROR` if the user has failed to provide a master server list for a configuration section. The fatal error may read `(mysqlnd_ms) Section [master] doesn't exist for host [name_of_a_config_section] in %s on line %d`.

A server is described with the `host`, `port`, `socket`, `db`, `user`, `password` and `connect_flags`. It is mandatory to provide at a value for `host`. If any of the other values is not given, it will be taken from the user API connect call, please, see also: using section names example.

Table of server configuration keywords.

Keyword	Description	Version
host	Database server host. This is a mandatory setting. Failing to provide, will cause an error of type E_RECOVERABLE_ERROR when the plugin tries to connect to the server. The error message may read (mysqlnd_ms) Cannot find [host] in [%s] section in config in %s on line %d.	Since 1.1.0.
port	Database server TCP/IP port.	Since 1.1.0.
socket	Database server Unix domain socket.	Since 1.1.0.
db	Database (schemata).	Since 1.1.0.
user	MySQL database user.	Since 1.1.0.
password	MySQL database user password.	Since 1.1.0.
connect_flags	Connection flags.	Since 1.1.0.

The plugin supports using only one master server. An experimental setting exists to enable multi-master support. The details are not documented. The setting is meant for development only.

slave array or object

List of one or more MySQL replication slave servers. The syntax is identical to setting master servers, please, see master above for details.

The plugin supports using one or more slave servers.

Setting a list of slave servers is mandatory. The plugin will report an error of the type E_ERROR if slave is not given for a configuration section. The fatal error message may read (mysqlnd_ms) Section [slave] doesn't exist for host [%s] in %s on line %d. Note, that it is valid to use an empty slave server list. The error has been introduced to prevent accidently setting no slaves by forgetting about the slave setting. A master-only setup is still possible using an empty slave server list.

If an empty slave list is configured and an attempt is made to execute a statement on a slave the plugin may emit a warning like mysqlnd_ms) Couldn't find the appropriate slave connection. 0 slaves to choose from. upon statement execution. It is possible that another warning follows such as (mysqlnd_ms) No connection selected by the last filter.

global_transaction_id_injection array or object

Global transaction identifier configuration related to both the use of the server built-in global transaction ID feature and the client-side emulation.

Keyword	Description	Version
fetch_last_gtid	SQL statement for accessing the latest global transaction identifier. The SQL statement is run if the plugin needs to know the most recent global transaction identifier. This can be the case, for example, when checking	Since 1.2.0.

Keyword	Description	Version
	MySQL Replication slave status. Also used with `mysqlnd_ms_get_last_gtid`.	
`check_for_gtid`	SQL statement for checking if a replica has replicated all transactions up to and including ones searched for. The SQL statement is run when searching for replicas which can offer a higher level of consistency than eventual consistency. The statement must contain a placeholder `#GTID` which is to be replaced with the global transaction identifier searched for by the plugin. Please, check the quickstart for examples.	Since 1.2.0.
`report_errors`	Whether to emit an error of type warning if an issue occurs while executing any of the configured SQL statements.	Since 1.2.0.
`on_commit`	Client-side global transaction ID emulation only. SQL statement to run when a transaction finished to update the global transaction identifier sequence number on the master. Please, see the quickstart for examples.	Since 1.2.0.
`wait_for_gtid_timeout`	Instructs the plugin to wait up to `wait_for_gtid_timeout` seconds for a slave to catch up when searching for slaves that can deliver session consistency. The setting limits the time spend for polling the slave status. If polling the status takes very long, the total clock time spend waiting may exceed `wait_for_gtid_timeout`. The plugin calls `sleep(1)` to sleep one second between each two polls. The setting can be used both with the plugins client-side emulation and the server-side global transaction identifier feature of MySQL 5.6. Waiting for a slave to replicate a certain GTID needed for session consistency also means throttling the client. By throttling the client the write load on the master is reduced indirectly. A primary copy based replication system, such as MySQL Replication, is given more time to reach a consistent state. This can be desired, for example, to increase the number of data copies for high availability considerations or to prevent the master from being overloaded.	Since 1.4.0.

`fabric` object	MySQL Fabric related settings. If the plugin is used together with MySQL Fabric, then the plugins configuration file no longer contains lists of MySQL servers. Instead, the plugin will ask MySQL Fabric which list of servers to use to perform a certain task.

A minimum plugin configuration for use with MySQL Fabric contains a list of one or more MySQL Fabric hosts that the plugin can query. If more than one MySQL Fabric host is configured, the plugin will use a roundrobin strategy to choose among them. Other strategies are currently not available.

Example 7.68 Minimum pluging configuration for use with MySQL Fabric

```
{
    "myapp": {
        "fabric": {
            "hosts": [
                {
                    "host" : "127.0.0.1",
                    "port" : 8080
                }
            ]
        }
    }
}
```

Each MySQL Fabric host is described using a JSON object with the following members.

Keyword	Description	Version
`host`	Host name of the MySQL Fabric host.	Since 1.6.0.
`port`	The TCP/IP port on which the MySQL Fabric host listens for remote procedure calls sent by clients such as the plugin.	Since 1.6.0.

The plugin is using PHP streams to communicate with MySQL Fabric through XML RPC over HTTP. By default no timeouts are set for the network communication. Thus, the plugin defaults to PHP stream default timeouts. Those defaults are out of control of the plugin itself.

An optional timeout value can be set to overrule the PHP streams default timeout setting. Setting the timeout in the plugins configuration file has the same effect as setting a timeout for a PHP user space HTTP connection established through PHP streams.

The plugins Fabric timeout value unit is seconds. The allowed value range is from 0 to 65535. The setting exists since version 1.6.

Example 7.69 Optional timeout for communication with Fabric

```
{
    "myapp": {
        "fabric": {
            "hosts": [
                {
                    "host" : "127.0.0.1",
```

```
                    "port" : 8080
                }
            ],
            "timeout": 2
        }
    }
}
```

Transaction stickiness and MySQL Fabric logic can collide. The stickiness option disables switching between servers for the duration of a transaction. When using Fabric and sharding the user may (erroneously) start a local transaction on one share and then attempt to switch to a different shard using either `mysqlnd_ms_fabric_select_shard` or `mysqlnd_ms_fabric_select_global`. In this case, the plugin will not reject the request to switch servers in the middle of a transaction but allow the user to switch to another server regardless of the transaction stickiness setting used. It is clearly a user error to write such code.

If transaction stickiness is enabled and you would like to get an error of type warning when calling `mysqlnd_ms_fabric_select_shard` or `mysqlnd_ms_fabric_select_global`, set the boolean flag `trx_warn_server_list_changes`.

Example 7.70 Warnings about the violation of transaction boundaries

```
{
    "myapp": {
        "fabric": {
            "hosts": [
                {
                    "host" : "127.0.0.1",
                    "port" : 8080
                }
            ],
            "trx_warn_serverlist_changes": 1
        },
        "trx_stickiness": "on"
    }
}
```

```
<?php
$link = new mysqli("myapp", "root", "", "test");
/*
  For the demo the call may fail.
  Failed or not we get into the state
  needed for the example.
*/
@mysqlnd_ms_fabric_select_global($link, 1);
$link->begin_transaction();
@$link->query("DROP TABLE IF EXISTS test");
/*
  Switching servers/shards is a mistake due to open
  local transaction!
```

```
*/
mysqlnd_ms_select_global($link, 1);
?>
```

The above example will output:

```
PHP Warning: mysqlnd_ms_fabric_select_global(): (mysqlnd_ms) Fabric server
```

Please, consider the feature experimental. Changes to syntax and semantics may happen.

`filters` object

List of filters. A filter is responsible to filter the list of available servers for executing a given statement. Filters can be chained. The `random` and `roundrobin` filter replace the `pick[]` directive used in prior version to select a load balancing policy. The `user` filter replaces the `mysqlnd_ms_set_user_pick_server` function.

Filters may accept parameters to refine their actions.

If no load balancing policy is set, the plugin will default to `random_once`. The `random_once` policy picks a random slave server when running the first read-only statement. The slave server will be used for all read-only statements until the PHP script execution ends. No load balancing policy is set and thus, defaulting takes place, if neither the `random` nor the `roundrobin` are part of a configuration section.

If a filter chain is configured so that a filter which output no more than once server is used as input for a filter which should be given more than one server as input, the plugin may emit a warning upon opening a connection. The warning may read: `(mysqlnd_ms) Error while creating filter '%s' . Non-multi filter '%s' already created. Stopping in %s on line %d`. Furthermore, an error of the error code `2000`, the sql state `HY000` and an error message similar to the warning may be set on the connection handle.

Example 7.71 Invalid filter sequence

```
   {
"myapp": {
    "master": {
        "master_0": {
            "host": "localhost"
        }
    },
    "slave": {
        "slave_0": {
            "host": "192.168.78.136",
            "port": "3306"
        }
    },
    "filters": [
        "roundrobin",
```

```
            "random"
        ]
    }
}
```

```
<?php
$link = new mysqli("myapp", "root", "", "test");
printf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error());
$link->query("SELECT 1 FROM DUAL");
?>
```

The above example will output:

```
PHP Warning:  mysqli::mysqli(): (HY000/2000): (mysqlnd_ms) Error while creatin
[2000] (mysqlnd_ms) Error while creating filter 'random' . Non-multi filter 'r
PHP Warning:  mysqli::query(): Couldn't fetch mysqli in filter_warning.php on
```

Filter: *`random`* object

The `random` filter features the random and random once load balancing policies, set through the `pick[]` directive in older versions.

The random policy will pick a random server whenever a read-only statement is to be executed. The random once strategy picks a random slave server once and continues using the slave for the rest of the PHP web request. Random once is a default, if load balancing is not configured through a filter.

If the `random` filter is not given any arguments, it stands for random load balancing policy.

Example 7.72 Random load balancing with `random` filter

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.78.136",
                "port": "3306"
            },
            "slave_1": {
                "host": "192.168.78.137",
                "port": "3306"
            }
        },
        "filters": [
            "random"
        ]
    }
```

```
}
```

Optionally, the `sticky` argument can be passed to the filter. If the parameter `sticky` is set to the string `1`, the filter follows the random once load balancing strategy.

Example 7.73 Random once load balancing with `random` filter

```
{
    "filters": {
        "random": {
            "sticky": "1"
        }
    }
}
```

Both the `random` and `roundrobin` filters support setting a priority, a weight for a server, since PECL/mysqlnd_ms 1.4.0. If the `weight` argument is passed to the filter, it must assign a weight for all servers. Servers must be given an alias name in the `slave` respectively `master` server lists. The alias must be used to reference servers for assigning a priority with `weight`.

Example 7.74 Referencing error

```
[E_RECOVERABLE_ERROR] mysqli_real_connect(): (mysqlnd_ms) Unknown server '
```

Using a wrong alias name with `weight` may result in an error similar to the shown above.

If `weight` is omitted, the default weight of all servers is one.

Example 7.75 Assigning a `weight` for load balancing

```
{
   "myapp": {
       "master": {
           "master1":{
               "host":"localhost",
               "socket":"\/var\/run\/mysql\/mysql.sock"
           }
       },
       "slave": {
           "slave1": {
               "host":"192.168.2.28",
               "port":3306
           },
           "slave2": {
               "host":"192.168.2.29",
               "port":3306
```

```
            },
            "slave3": {
                "host":"192.0.43.10",
                "port":3306
            },
        },
        "filters": {
            "random": {
                "weights": {
                    "slave1":8,
                    "slave2":4,
                    "slave3":1,
                    "master1":1
                }
            }
        }
    }
}
```

At the average a server assigned a weight of two will be selected twice as often as a server assigned a weight of one. Different weights can be assigned to reflect differently sized machines, to prefer co-located slaves which have a low network latency or, to configure a standby failover server. In the latter case, you may want to assign the standby server a very low weight in relation to the other servers. For example, given the configuration above `slave3` will get only some eight percent of the requests in the average. As long as `slave1` and `slave2` are running, it will be used sparsely, similar to a standby failover server. Upon failure of `slave1` and `slave2`, the usage of `slave3` increases. Please, check the notes on failover before using `weight` this way.

Valid weight values range from 1 to 65535.

Unknown arguments are ignored by the filter. No warning or error is given.

The filter expects one or more servers as input. Outputs one server. A filter sequence such as `random`, `roundrobin` may cause a warning and an error message to be set on the connection handle when executing a statement.

List of filter arguments.

Keyword	Description	Version
`sticky`	Enables or disabled random once load balancing policy. See above.	Since 1.2.0.
`weight`	Assigns a load balancing weight/priority to a server. Please, see above for a description.	Since 1.4.0.

Filter: *`roundrobin`* object

If using the `roundrobin` filter, the plugin iterates over the list of configured slave servers to pick a server for statement execution. If the plugin reaches the end of the list, it wraps around to the beginning of the list and picks the first configured slave server.

Example 7.76 `roundrobin` filter

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.78.136",
                "port": "3306"
            }
        },
        "filters": [
            "roundrobin"
        ]
    }
}
```

Expects one or more servers as input. Outputs one server. A filter sequence such as `roundrobin`, `random` may cause a warning and an error message to be set on the connection handle when executing a statement.

List of filter arguments.

Keyword	Description	Version
`weight`	Assigns a load balancing weight/priority to a server. Please, find a description above.	Since 1.4.0.

Filter: *`user`* object

The `user` replaces `mysqlnd_ms_set_user_pick_server` function, which was removed in 1.1.0-beta. The filter sets a callback for user-defined read/write splitting and server selection.

The plugins built-in read/write query split mechanism decisions can be overwritten in two ways. The easiest way is to prepend a query string with the SQL hints `MYSQLND_MS_MASTER_SWITCH`, `MYSQLND_MS_SLAVE_SWITCH` or `MYSQLND_MS_LAST_USED_SWITCH`. Using SQL hints one can control, for example, whether a query shall be send to the MySQL replication master server or one of the slave servers. By help of SQL hints it is not possible to pick a certain slave server for query execution.

Full control on server selection can be gained using a callback function. Use of a callback is recommended to expert users only because the callback has to cover all cases otherwise handled by the plugin.

The plugin will invoke the callback function for selecting a server from the lists of configured master and slave servers. The callback function inspects the query to run and picks a server for query execution by returning the hosts URI, as found in the master and slave list.

If the lazy connections are enabled and the callback chooses a slave server for which no connection has been established so far and establishing the connection to the slave fails, the plugin will return an error upon the next action on the failed connection, for example, when running a query. It is the responsibility of the application developer to handle the error. For example, the application can re-run the query to trigger a new server selection and callback invocation. If so, the callback must make sure to select a different slave, or check slave availability, before returning to the plugin to prevent an endless loop.

Example 7.77 Setting a callback

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.78.136",
                "port": "3306"
            }
        },
        "filters": {
            "user": {
                "callback": "pick_server"
            }
        }
    }
}
```

The callback is supposed to return a host to run the query on. The host URI is to be taken from the master and slave connection lists passed to the callback function. If callback returns a value neither found in the master nor in the slave connection lists the plugin will emit an error of the type `E_RECOVERABLE_ERROR` The error may read like `(mysqlnd_ms) User filter callback has returned an unknown server. The server 'server that is not in master or slave list' can neither be found in the master list nor in the slave list`. If the application catches the error to ignore it, follow up errors may be set on the connection handle, for example, `(mysqlnd_ms) No connection selected by the last filter` with the error code `2000` and the sqlstate `HY000`. Furthermore a warning may be emitted.

Referencing a non-existing function as a callback will result in any error of the type `E_RECOVERABLE_ERROR` whenever the plugin tries to callback function. The error message may reads like: `(mysqlnd_ms) Specified callback (pick_server) is not a valid callback`. If the application catches the error to ignore it, follow up errors may be set on the connection handle, for example, `(mysqlnd_ms) Specified callback (pick_server) is not`

`a valid callback` with the error code `2000` and the sqlstate `HY000`. Furthermore a warning may be emitted.

The following parameters are passed from the plugin to the callback.

Parameter	Description	Version
`connected_host`	URI of the currently connected database server.	Since 1.1.0.
`query`	Query string of the statement for which a server needs to be picked.	Since 1.1.0.
`masters`	List of master servers to choose from. Note, that the list of master servers may not be identical to the list of configured master servers if the filter is not the first in the filter chain. Previously run filters may have reduced the master list already.	Since 1.1.0.
`slaves`	List of slave servers to choose from. Note, that the list of master servers may not be identical to the list of configured master servers if the filter is not the first in the filter chain. Previously run filters may have reduced the master list already.	Since 1.1.0.
`last_used_connection`	URI of the server of the connection used to execute the previous statement on.	Since 1.1.0.
`in_transaction`	Boolean flag indicating whether the statement is part of an open transaction. If autocommit mode is turned off, this will be set to `TRUE`. Otherwise it is set to `FALSE`. Transaction detection is based on monitoring the mysqlnd library call `set_autocommit`. Monitoring is not possible before PHP 5.4.0. Please, see connection pooling and switching concepts discussion for further details.	Since 1.1.0.

Example 7.78 Using a callback

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.2.27",
                "port": "3306"
            },
            "slave_1": {
                "host": "192.168.78.136",
                "port": "3306"
            }
        },
        "filters": {
            "user": {
                "callback": "pick_server"
```

```
            }
        }
    }
}
```

```
<?php
function pick_server($connected, $query, $masters, $slaves, $last_used_connect
{
 static $slave_idx = 0;
 static $num_slaves = NULL;
 if (is_null($num_slaves))
  $num_slaves = count($slaves);

 /* default: fallback to the plugins build-in logic */
 $ret = NULL;

 printf("User has connected to '%s'...\n", $connected);
 printf("... deciding where to run '%s'\n", $query);

 $where = mysqlnd_ms_query_is_select($query);
 switch ($where)
 {
  case MYSQLND_MS_QUERY_USE_MASTER:
   printf("... using master\n");
   $ret = $masters[0];
   break;
  case MYSQLND_MS_QUERY_USE_SLAVE:
   /* SELECT or SQL hint for using slave */
   if (stristr($query, "FROM table_on_slave_a_only"))
   {
    /* a table which is only on the first configured slave  */
    printf("... access to table available only on slave A detected\n");
    $ret = $slaves[0];
   }
   else
   {
    /* round robin */
    printf("... some read-only query for a slave\n");
    $ret = $slaves[$slave_idx++ % $num_slaves];
   }
   break;
  case MYSQLND_MS_QUERY_LAST_USED:
   printf("... using last used server\n");
   $ret = $last_used_connection;
   break;
 }

 printf("... ret = '%s'\n", $ret);
 return $ret;
}

$mysqli = new mysqli("myapp", "root", "", "test");

if (!($res = $mysqli->query("SELECT 1 FROM DUAL")))
 printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
else
 $res->close();

if (!($res = $mysqli->query("SELECT 2 FROM DUAL")))
 printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
else
 $res->close();
```

```
if (!($res = $mysqli->query("SELECT * FROM table_on_slave_a_only")))
 printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
else
 $res->close();

$mysqli->close();
?>
```

The above example will output:

```
User has connected to 'myapp'...
... deciding where to run 'SELECT 1 FROM DUAL'
... some read-only query for a slave
... ret = 'tcp://192.168.2.27:3306'
User has connected to 'myapp'...
... deciding where to run 'SELECT 2 FROM DUAL'
... some read-only query for a slave
... ret = 'tcp://192.168.78.136:3306'
User has connected to 'myapp'...
... deciding where to run 'SELECT * FROM table_on_slave_a_only'
... access to table available only on slave A detected
... ret = 'tcp://192.168.2.27:3306'
```

Filter: *`user_multi`* object

The `user_multi` differs from the `user` only in one aspect. Otherwise, their syntax is identical. The `user` filter must pick and return exactly one node for statement execution. A filter chain usually ends with a filter that emits only one node. The filter chain shall reduce the list of candidates for statement execution down to one. This, only one node left, is the case after the `user` filter has been run.

The `user_multi` filter is a multi filter. It returns a list of slave and a list of master servers. This list needs further filtering to identify exactly one node for statement execution. A multi filter is typically placed at the top of the filter chain. The `quality_of_service` filter is another example of a multi filter.

The return value of the callback set for `user_multi` must be an array with two elements. The first element holds a list of selected master servers. The second element contains a list of selected slave servers. The lists shall contain the keys of the slave and master servers as found in the slave and master lists passed to the callback. The below example returns random master and slave lists extracted from the functions input.

Example 7.79 Returning random masters and slaves

```
<?php
function pick_server($connected, $query, $masters, $slaves, $last_used_con
{
  $picked_masters = array()
  foreach ($masters as $key => $value) {
    if (mt_rand(0, 2) > 1)
      $picked_masters[] = $key;
```

```
  }
  $picked_slaves = array()
  foreach ($slaves as $key => $value) {
    if (mt_rand(0, 2) > 1)
      $picked_slaves[] = $key;
  }
  return array($picked_masters, $picked_slaves);
}
?>
```

The plugin will issue an error of type `E_RECOVERABLE` if the callback fails to return a server list. The error may read `(mysqlnd_ms) User multi filter callback has not returned a list of servers to use. The callback must return an array in %s on line %d`. In case the server list is not empty but has invalid servers key/ids in it, an error of type `E_RECOVERABLE` will the thrown with an error message like `(mysqlnd_ms) User multi filter callback has returned an invalid list of servers to use. Server id is negative in %s on line %d`, or similar.

Whether an error is emitted in case of an empty slave or master list depends on the configuration. If an empty master list is returned for a write operation, it is likely that the plugin will emit a warning that may read `(mysqlnd_ms) Couldn't find the appropriate master connection. 0 masters to choose from. Something is wrong in %s on line %d`. Typically a follow up error of type `E_ERROR` will happen. In case of a read operation and an empty slave list the behavior depends on the fail over configuration. If fail over to master is enabled, no error should appear. If fail over to master is deactivated the plugin will emit a warning that may read `(mysqlnd_ms) Couldn't find the appropriate slave connection. 0 slaves to choose from. Something is wrong in %s on line %d`.

Filter: *`node_groups`* object

The `node_groups` filter lets you group cluster nodes and query selected groups, for example, to support data partitioning. Data partitioning can be required for manual sharding, primary copy based clusters running multiple masters, or to avoid hot spots in update everywhere clusters that have no built-in partitioning. The filter is a multi filter which returns zero, one or multiple of its input servers. Thus, it must be followed by other filters to reduce the number of candidates down to one for statement execution.

Keyword	Description	Version
`user defined node group name`	One or more node groups must be defined. A node group can have an arbitrary user defined name. The name is used in combination with a SQL hint to restrict query execution to the nodes listed for the node group. To run a query on any of the servers of a node group, the query must begin with the SQL hint `/*user defined node group name*/`. Please note, no white space is allowed around `user defined node group name`. Because `user defined node`	Since 1.5.0.

Keyword	Description	Version
	`group name` is used as-is as part of a SQL hint, you should choose the name that is compliant with the SQL language. Each node group entry must contain a list of `master` servers. Additional `slave` servers are allowed. Failing to provide a list of `master` for a node group `name_of_group` may cause an error of type `E_RECOVERABLE_ERROR` like `(mysqlnd_ms) No masters configured in node group 'name_of_group' for 'node_groups' filter`. The list of master and slave servers must reference corresponding entries in the global master respectively slave server list. Referencing an unknown server in either of the both server lists may cause an `E_RECOVERABLE_ERROR` error like `(mysqlnd_ms) Unknown master 'server_alias_name' (section 'name_of_group') in 'node_groups' filter configuration`. **Example 7.80 Manual partitioning** `{ "myapp": { "master": { "master_0": { "host": "localhost", "socket": "\/tmp\/mysql.sock" } }, "slave": { "slave_0": { "host": "192.168.2.28", "port": 3306 }, "slave_1": { "host": "127.0.0.1", "port": 3311 } }, "filters": { "node_groups": { "Partition_A" : { "master": ["master_0"], "slave": ["slave_0"] } }, "roundrobin": [] } } }`	

Keyword	Description	Version
	Please note, if a filter chain generates an empty slave list and the PHP configuration directive `mysqlnd_ms.multi_master=0` is used, the plugin may emit a warning.	

Filter: *`quality_of_service`* object

The `quality_of_service` identifies cluster nodes capable of delivering a certain quality of service. It is a multi filter which returns zero, one or multiple of its input servers. Thus, it must be followed by other filters to reduce the number of candidates down to one for statement execution.

The `quality_of_service` filter has been introduced in 1.2.0-alpha. In the 1.2 series the filters focus is on the consistency aspect of service quality. Different types of clusters offer different default data consistencies. For example, an asynchronous MySQL replication slave offers eventual consistency. The slave may not be able to deliver requested data because it has not replicated the write, it may serve stale database because its lagging behind or it may serve current information. Often, this is acceptable. In some cases higher consistency levels are needed for the application to work correct. In those cases, the `quality_of_service` can filter out cluster nodes which cannot deliver the necessary quality of service.

The `quality_of_service` filter can be replaced or created at runtime. A successful call to `mysqlnd_ms_set_qos` removes all existing `qos` filter entries from the filter list and installs a new one at the very beginning. All settings that can be made through `mysqlnd_ms_set_qos` can also be in the plugins configuration file. However, use of the function is by far the most common use case. Instead of setting session consistency and strong consistency service levels in the plugins configuration file it is recommended to define only masters and no slaves. Both service levels will force the use of masters only. Using an empty slave list shortens the configuration file, thus improving readability. The only service level for which there is a case of defining in the plugins configuration file is the combination of eventual consistency and maximum slave lag.

Keyword	Description	Version
`eventual_consistency`	Request eventual consistency. Allows the use of all master and slave servers. Data returned may or may not be current. Eventual consistency accepts an optional `age` parameter. If `age` is given the plugin considers only slaves for reading for which MySQL replication reports a slave lag less or equal to `age`. The replication lag is measure using `SHOW SLAVE STATUS`. If the plugin fails to fetch the replication lag, the slave tested is skipped. Implementation details and tips are given in the quality of service concepts section.	Since 1.2.0.

Keyword	Description	Version
	Please note, if a filter chain generates an empty slave list and the PHP configuration directive `mysqlnd_ms.multi_master=0` is used, the plugin may emit a warning. **Example 7.81 Global limit on slave lag** `{ "myapp": { "master": { "master_0": { "host": "localhost" } }, "slave": { "slave_0": { "host": "192.168.2.27", "port": "3306" }, "slave_1": { "host": "192.168.78.136", "port": "3306" } }, "filters": { "quality_of_service": { "eventual_consistency": { "age":123 } } } } }`	
`session_consistency`	Request session consistency (read your writes). Allows use of all masters and all slaves which are in sync with the master. If no further parameters are given slaves are filtered out as there is no reliable way to test if a slave has caught up to the master or is lagging behind. Please note, if a filter chain generates an empty slave list and the PHP configuration directive `mysqlnd_ms.multi_master=0` is used, the plugin may emit a warning. Session consistency temporarily requested using `mysqlnd_ms_set_qos` is a valuable alternative to using `master_on_write`. `master_on_write` is likely to send more statements to the master than needed. The application may be able to continue operation at a lower consistency level after it has done some critical reads.	Since 1.1.0.

Keyword	Description	Version
`strong_consistency`	Request strong consistency. Only masters will be used.	Since 1.2.0.

`failover` Up to and including 1.3.x: string. Since 1.4.0: object.

Failover policy. Supported policies: `disabled` (default), `master`, `loop_before_master` (Since 1.4.0).

If no failover policy is set, the plugin will not do any automatic failover (`failover=disabled`). Whenever the plugin fails to connect a server it will emit a warning and set the connections error code and message. Thereafter it is up to the application to handle the error and, for example, resent the last statement to trigger the selection of another server.

Please note, the automatic failover logic is applied when opening connections only. Once a connection has been opened no automatic attempts are made to reopen it in case of an error. If, for example, the server a connection is connected to is shut down and the user attempts to run a statement on the connection, no automatic failover will be tried. Instead, an error will be reported.

If using `failover=master` the plugin will implicitly failover to a master, if available. Please check the concepts documentation to learn about potential pitfalls and risks of using `failover=master`.

Example 7.82 Optional master failover when failing to connect to slave (PECL/mysqlnd_ms < 1.4.0)

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.78.136",
                "port": "3306"
            }
        },
        "failover": "master"
    }
}
```

Since PECL/mysqlnd_ms 1.4.0 the failover configuration keyword refers to an object.

Example 7.83 New syntax since 1.4.0

```
{
    "myapp": {
        "master": {
            "master_0": {
```

```
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.78.136",
                "port": "3306"
            }
        },
        "failover": {"strategy": "master" }
    }
}
```

Keyword	Description	Version
`strategy`	Failover policy. Possible values: `disabled` (default), `master`, `loop_before_master` A value of `disabled` disables automatic failover. Setting `master` instructs the plugin to try to connect to a master in case of a slave connection error. If the master connection attempt fails, the plugin exists the failover loop and returns an error to the user. If using `loop_before_master` and a slave request is made, the plugin tries to connect to other slaves before failing over to a master. If multiple master are given and multi master is enabled, the plugin also loops over the list of masters and attempts to connect before returning an error to the user.	Since 1.4.0.
`remember_failed`	Remember failures for the duration of a web request. Default: `false`. If set to `true` the plugin will remember failed hosts and skip the hosts in all future load balancing made for the duration of the current web request.	Since 1.4.0. The feature is only available together with the `random` and `roundrobin` load balancing filter. Use of the setting is recommended.
`max_retries`	Maximum number of connection attempts before skipping host. Default: `0` (no limit). The setting is used to prevent hosts from being dropped of the host list upon the first failure. If set to `n > 0`, the plugin will keep the node in the node list even after a failed connection attempt. The node will not be removed immediately from the slave respectively master lists after the first	Since 1.4.0. The feature is only available together with the `random` and `roundrobin` load

Keyword	Description	Version
	connection failure but instead be tried to connect to up to `n` times in future load balancing rounds before being removed.	balancing filter.

Setting `failover` to any other value but `disabled`, `master` or `loop_before_master` will not emit any warning or error.

`lazy_connections` bool

Controls the use of lazy connections. Lazy connections are connections which are not opened before the client sends the first connection. Lazy connections are a default.

It is strongly recommended to use lazy connections. Lazy connections help to keep the number of open connections low. If you disable lazy connections and, for example, configure one MySQL replication master server and two MySQL replication slaves, the plugin will open three connections upon the first call to a connect function although the application might use the master connection only.

Lazy connections bare a risk if you make heavy use of actions which change the state of a connection. The plugin does not dispatch all state changing actions to all connections from the connection pool. The few dispatched actions are applied to already opened connections only. Lazy connections opened in the future are not affected. Only some settings are "remembered" and applied when lazy connections are opened.

Example 7.84 Disabling lazy connection

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.78.136",
                "port": "3306"
            }
        },
        "lazy_connections": 0
    }
}
```

Please, see also `server_charset` to overcome potential problems with string escaping and servers using different default charsets.

`server_charset` string

The setting has been introduced in 1.4.0. It is recommended to set it if using lazy connections.

The `server_charset` setting serves two purposes. It acts as a fallback charset to be used for string escaping done before a connection has been established and it helps to avoid escaping pitfalls

in heterogeneous environments which servers using different default charsets.

String escaping takes a connections charset into account. String escaping is not possible before a connection has been opened and the connections charset is known. The use of lazy connections delays the actual opening of connections until a statement is send.

An application using lazy connections may attempt to escape a string before sending a statement. In fact, this should be a common case as the statement string may contain the string that is to be escaped. However, due to the lazy connection feature no connection has been opened yet and escaping fails. The plugin may report an error of the type `E_WARNING` and a message like `(mysqlnd_ms) string escaping doesn't work without established connection. Possible solution is to add server_charset to your configuration` to inform you of the pitfall.

Setting `server_charset` makes the plugin use the given charset for string escaping done on lazy connection handles before establishing a network connection to MySQL. Furthermore, the plugin will enforce the use of the charset when the connection is established.

Enforcing the use of the configured charset used for escaping is done to prevent tapping into the pitfall of using a different charset for escaping than used later for the connection. This has the additional benefit of removing the need to align the charset configuration of all servers used. No matter what the default charset on any of the servers is, the plugin will set the configured one as a default.

The plugin does not stop the user from changing the charset at any time using the `set_charset` call or corresponding SQL statements. Please, note that the use of SQL is not recommended as it cannot be monitored by the plugin. The user can, for example, change the charset on a lazy connection handle after escaping a string and before the actual connection is opened. The charset set by the user will be used for any subsequent escaping before the connection is established. The connection will be established using the configured charset, no matter what the server charset is or what the user has set before. Once a connection has been opened, `set_charset` is of no meaning anymore.

Example 7.85 String escaping on a lazy connection handle

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.78.136",
                "port": "3306"
            }
```

```
        },
        "lazy_connections": 1,
        "server_charset" : "utf8"
    }
}
```

```
<?php
$mysqli = new mysqli("myapp", "username", "password", "database");
$mysqli->real_escape("this will be escaped using the server_charset setting -
$mysqli->set_charset("latin1");
$mysqli->real_escape("this will be escaped using latin1");
/* server_charset implicitly set - utf8 connection */
$mysqli->query("SELECT 'This connection will be set to server_charset upon est
/* latin1 used from now on */
$mysqli->set_charset("latin1");
?>
```

`master_on_write` bool

If set, the plugin will use the master server only after the first statement has been executed on the master. Applications can still send statements to the slaves using SQL hints to overrule the automatic decision.

The setting may help with replication lag. If an application runs an `INSERT` the plugin will, by default, use the master to execute all following statements, including `SELECT` statements. This helps to avoid problems with reads from slaves which have not replicated the `INSERT` yet.

Example 7.86 Master on write for consistent reads

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.78.136",
                "port": "3306"
            }
        },
        "master_on_write": 1
    }
}
```

Please, note the `quality_of_service` filter introduced in version 1.2.0-alpha. It gives finer control, for example, for achieving read-your-writes and, it offers additional functionality introducing service levels.

All transaction stickiness settings, including `trx_stickiness=on`, are overruled by `master_on_write=1`.

trx_stickiness string

Transaction stickiness policy. Supported policies: `disabled` (default), `master`.

The setting requires 5.4.0 or newer. If used with PHP older than 5.4.0, the plugin will emit a warning like `(mysqlnd_ms) trx_stickiness strategy is not supported before PHP 5.3.99`.

If no transaction stickiness policy is set or, if setting `trx_stickiness=disabled`, the plugin is not transaction aware. Thus, the plugin may load balance connections and switch connections in the middle of a transaction. The plugin is not transaction safe. SQL hints must be used avoid connection switches during a transaction.

As of PHP 5.4.0 the mysqlnd library allows the plugin to monitor the `autocommit` mode set by calls to the libraries `set_autocommit()` function. If setting `set_stickiness=master` and `autocommit` gets disabled by a PHP MySQL extension invoking the `mysqlnd` library internal function call `set_autocommit()`, the plugin is made aware of the begin of a transaction. Then, the plugin stops load balancing and directs all statements to the master server until `autocommit` is enabled. Thus, no SQL hints are required.

An example of a PHP MySQL API function calling the `mysqlnd` library internal function call `set_autocommit()` is `mysqli_autocommit`.

Although setting `trx_stickiness=master`, the plugin cannot be made aware of `autocommit` mode changes caused by SQL statements such as `SET AUTOCOMMIT=0` or `BEGIN`.

As of PHP 5.5.0, the mysqlnd library features additional C API calls to control transactions. The level of control matches the one offered by SQL statements. The `mysqli` API has been modified to use these calls. Since version 1.5.0, PECL/mysqlnd_ms can monitor not only `mysqli_autocommit`, but also `mysqli_begin`, `mysqli_commit` and `mysqli_rollback` to detect transaction boundaries and stop load balancing for the duration of a transaction.

Example 7.87 Using master to execute transactions

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.78.136",
                "port": "3306"
            }
        },
        "trx_stickiness": "master"
    }
}
```

Since version 1.5.0 automatic and silent failover is disabled for the duration of a transaction. If the boundaries of a transaction have been properly detected, transaction stickiness is enabled and a server fails, the plugin will not attempt to fail over to the next server, if any, regardless of the failover policy configured. The user must handle the error manually. Depending on the configuration, the plugin may emit an error of type `E_WARNING` reading like `(mysqlnd_ms) Automatic failover is not permitted in the middle of a transaction`. This error may then be overwritten by follow up errors such as `(mysqlnd_ms) No connection selected by the last filter`. Those errors will be generated by the failing query function.

Example 7.88 No automatic failover, error handling pitfall

```
<?php
/* assumption: automatic failover configured */
$mysqli = new mysqli("myapp", "username", "password", "database");

/* sets plugin internal state in_trx = 1 */
$mysqli->autocommit(false);

/* assumption: server fails */
if (!($res = $mysqli->query("SELECT 'Assume this query fails' AS _msg FROM DUA
 /* handle failure of transaction, plugin internal state is still in_trx = 1 *
 printf("[%d] %s", $mysqli->errno, $mysqli->error);
 /*
  If using autocommit() based transaction detection it is a
  MUST to call autocommit(true). Otherwise the plugin assumes
  the current transaction continues and connection
  changes remain forbidden.
 */
 $mysqli->autocommit(true);
 /* Likewise, you'll want to start a new transaction */
 $mysqli->autocommit(false);
}
/* latin1 used from now on */
$mysqli->set_charset("latin1");
?>
```

If a server fails in the middle of a transaction the plugin continues to refuse to switch connections until the current transaction has been finished. Recall that the plugin monitors API calls to detect transaction boundaries. Thus, you have to, for example, enable auto commit mode to end the current transaction before the plugin continues load balancing and switches the server. Likewise, you will want to start a new transaction immediately thereafter and disable auto commit mode again.

Not handling failed queries and not ending a failed transaction using API calls may cause all following commands emit errors such as `Commands out of sync; you can't run this command now`. Thus, it is important to handle all errors.

`transient_error` object — The setting has been introduced in 1.6.0.

A database cluster node may reply a transient error to a client. The client can then repeat the operation on the same node, fail over to a different node or abort the operation. Per definition is it safe for a client to retry the same operation on the same node before giving up.

PECL/mysqlnd_ms can perform the retry loop on behalf of the application. By configuring transient_error the plugin can be instructed to repeat operations failing with a certain error code for a certain maximum number of times with a pause between the retries. If the transient error disappears during loop execution, it is hidden from the application. Otherwise, the error is forwarded to the application by the end of the loop.

Example 7.89 Retry loop for transient errors

```
{
    "myapp": {
        "master": {
            "master_0": {
                "host": "localhost"
            }
        },
        "slave": {
            "slave_0": {
                "host": "192.168.78.136",
                "port": "3306"
            }
       },
       "transient_error": {
          "mysql_error_codes": [
            1297
          ],
          "max_retries": 2,
          "usleep_retry": 100
       }
    }
}
```

Keyword	Description	Version
mysql_error_codes	List of transient error codes. You may add any MySQL error code to the list. It is possible to consider any error as transient not only 1297 (HY000 (ER_GET_TEMPORARY_ERRMSG), Message: Got temporary error %d '%s' from %s). Before adding other codes but 1297 to the list, make sure your cluster supports a new attempt without impacting the state of your application.	Since 1.6.0.
max_retries	How often to retry an operation which fails with a transient error before forwarding the failure to the user. Default: 1	Since 1.6.0.

Keyword	Description	Version
`usleep_interval`	Milliseconds to sleep between transient error retries. The value is passed to the C function `usleep`, hence the name. Default: `100`	Since 1.6.0.

xa object

The setting has been introduced in 1.6.0.

Experimental

The feature is currently under development. There may be issues and/or feature limitations. Do not use in production environments.

state_store

record_participant_credentials

Whether to store the username and password of a global transaction participant in the participants table. If disabled, the garbage collection will use the default username and password when connecting to the participants. Unless you are using a different

username and password for each of your MySQL servers, you can use the default and avoid storing the sensible information in state store.

Please note, username and password are stored in clear text when using the MySQL state store, which is the only one available. It is in your responsibil to protect this

		sensible information. Default: `false`
	participant_localhost_ip	During XA garbage collection the plugin may find a participant server for which the host `localhost` has been recorded. If the garbage collection takes place on another host but the host that has written the participant record to the state store, the host name `localhost` now resolves to

a different host. Therefore, when recording a participant servers host name in the state store, a value of `localhos` must be replaced with the actual IP address of `localhos`

Setting `particip` should be considered only if using `localhos` cannot be avoided. From a garbage collection point of view only, it is preferrable not

		to configure any socket connection but to provide an IP address and port for a node.
	mysql	The MySQL state store is the only state store available.
		global_trx_tabl

participant_

garbage_c

host

user

password

db

port

socket

rollback_on_close	Whether to automatically rollback an open global transaction when a connection is closed. If enabled, it mimics the default behaviour of local transactions. Should a client disconnect, the server rolls back any open and unfinished transactions. Default: `true`

garbage_collection	max_retries	Maximum number of garbage collection runs before giving up. Allowed values are from `0` to `100`. A setting of `0` means no limit, unless the state store enforces a limit. Should the state store enforce a limit, it can be supposed to be significantly higher than `100`. Available since 1.6.0. Please note,

it is important to end failed XA transactions within reasonable time to make participating servers free resources bound to the transaction. The built-in garbage collection is not expected to fail for a long period as long as crashed servers become available again quickly. Still, a situation may arise where a human is required

		to act because the built-in garbage collection stopped or failed. In this case, you may first want to check if the transaction still cannot be fixed by forcing `mysqlnd_` to ignore the setting, prior to handling it manually. Default: `5`
	probability	Garbage collection probability. Allowed values are from `0` to `1000`. A

setting of `0` disables automatic background garbage collection. Despite a setting of `0` it is still possible to trigger garbage collection by calling `mysqlnd_ms_` Available since 1.6.0.

The automatic garbage collection of stalled XA transaction is only available if a state store have been configured. The state store is responsible to keep track

of XA transaction Based on its recordings it can find blocked XA transaction where the client has crashed, connect to the participants and rollback the unfinished transaction

The garbage collection is triggered as part of PHP's request shutdown procedure at the end of a web request. That is after your PHP script has

		finished working. Do decide whether to run the garbage collection a random value between `0` and `1000` is computed. If the `probability` value is higher or equal to the random value, the state stores garbage collection routines are invoked. Default: `5`
	max_transactions_per_run	Maximum number of unfinished XA transactions considered by the garbage collection

during one run. Allowed values are from 1 to 32768. Available since 1.6.0.

Cleaning up an unfinished XA transaction takes considerab amounts of time and resources. The garbage collection routine may have to connect to several participants of a failed global transaction to issue the SQL commands for rolling back the unfinished tranaction.

Default: `100`

7.6.4.3 Plugin configuration file (<= 1.0.x)

> **Note**
>
> The below description applies to PECL/mysqlnd_ms < 1.1.0-beta. It is not valid for later versions.

The plugin is using its own configuration file. The configuration file holds information on the MySQL replication master server, the MySQL replication slave servers, the server pick (load balancing) policy, the failover strategy and the use of lazy connections.

The PHP configuration directive `mysqlnd_ms.ini_file` is used to set the plugins configuration file.

The configuration file mimics standard the `php.ini` format. It consists of one or more sections. Every section defines its own unit of settings. There is no global section for setting defaults.

Applications reference sections by their name. Applications use section names as the host (server) parameter to the various connect methods of the mysqli, mysql and PDO_MYSQL extensions. Upon connect the mysqlnd plugin compares the hostname with all section names from the plugin configuration file. If hostname and section name match, the plugin will load the sections settings.

Example 7.93 Using section names example

```
[myapp]
master[] = localhost
slave[] = 192.168.2.27
slave[] = 192.168.2.28:3306
[localhost]
master[] = localhost:/tmp/mysql/mysql.sock
slave[] = 192.168.3.24:3305
slave[] = 192.168.3.65:3309
```

```
<?php
/* All of the following connections will be load balanced */
$mysqli = new mysqli("myapp", "username", "password", "database");
$pdo = new PDO('mysql:host=myapp;dbname=database', 'username', 'password');
$mysql = mysql_connect("myapp", "username", "password");

$mysqli = new mysqli("localhost", "username", "password", "database");
?>
```

Section names are strings. It is valid to use a section name such as `192.168.2.1`, `127.0.0.1` or `localhost`. If, for example, an application connects to `localhost` and a plugin configuration section `[localhost]` exists, the semantics of the connect operation are changed. The application will no longer only use the MySQL server running on the host `localhost` but the plugin will start to load balance

MySQL queries following the rules from the `[localhost]` configuration section. This way you can load balance queries from an application without changing the applications source code.

The `master[]`, `slave[]` and `pick[]` configuration directives use a list-like syntax. Configuration directives supporting list-like syntax may appear multiple times in a configuration section. The plugin maintains the order in which entries appear when interpreting them. For example, the below example shows two `slave[]` configuration directives in the configuration section `[myapp]`. If doing round-robin load balancing for read-only queries, the plugin will send the first read-only query to the MySQL server `mysql_slave_1` because it is the first in the list. The second read-only query will be send to the MySQL server `mysql_slave_2` because it is the second in the list. Configuration directives supporting list-like syntax result are ordered from top to bottom in accordance to their appearance within a configuration section.

Example 7.94 List-like syntax

```
[myapp]
master[] = mysql_master_server
slave[] = mysql_slave_1
slave[] = mysql_slave_2
```

Here is a short explanation of the configuration directives that can be used.

`master[]` string

URI of a MySQL replication master server. The URI follows the syntax `hostname[:port|unix_domain_socket]`.

The plugin supports using only one master server.

Setting a master server is mandatory. The plugin will report a warning upon connect if the user has failed to provide a master server for a configuration section. The warning may read `(mysqlnd_ms) Cannot find master section in config`. Furthermore the plugin may set an error code for the connection handle such as `HY000/2000 (CR_UNKNOWN_ERROR)`. The corresponding error message depends on your language settings.

`slave[]` string

URI of one or more MySQL replication slave servers. The URI follows the syntax `hostname[:port|unix_domain_socket]`.

The plugin supports using one or more slave servers.

Setting a slave server is mandatory. The plugin will report a warning upon connect if the user has failed to provide at least one slave server for a configuration section. The warning may read `(mysqlnd_ms) Cannot find slaves section in config`. Furthermore the plugin may set an error code for the connection handle such as `HY000/2000 (CR_UNKNOWN_ERROR)`. The corresponding error message depends on your language settings.

`pick[]` string

Load balancing (server picking) policy. Supported policies: `random`, `random_once` (default), `roundrobin`, `user`.

If no load balancing policy is set, the plugin will default to `random_once`. The `random_once` policy picks a random slave server

when running the first read-only statement. The slave server will be used for all read-only statements until the PHP script execution ends.

The `random` policy will pick a random server whenever a read-only statement is to be executed.

If using `roundrobin` the plugin iterates over the list of configured slave servers to pick a server for statement execution. If the plugin reaches the end of the list, it wraps around to the beginning of the list and picks the first configured slave server.

Setting more than one load balancing policy for a configuration section makes only sense in conjunction with `user` and `mysqlnd_ms_set_user_pick_server`. If the user defined callback fails to pick a server, the plugin falls back to the second configured load balancing policy.

`failover` string

Failover policy. Supported policies: `disabled` (default), `master`.

If no failover policy is set, the plugin will not do any automatic failover (`failover=disabled`). Whenever the plugin fails to connect a server it will emit a warning and set the connections error code and message. Thereafter it is up to the application to handle the error and, for example, resent the last statement to trigger the selection of another server.

If using `failover=master` the plugin will implicitly failover to a slave, if available. Please check the concepts documentation to learn about potential pitfalls and risks of using `failover=master`.

`lazy_connections` bool

Controls the use of lazy connections. Lazy connections are connections which are not opened before the client sends the first connection.

It is strongly recommended to use lazy connections. Lazy connections help to keep the number of open connections low. If you disable lazy connections and, for example, configure one MySQL replication master server and two MySQL replication slaves, the plugin will open three connections upon the first call to a connect function although the application might use the master connection only.

Lazy connections bare a risk if you make heavy use of actions which change the state of a connection. The plugin does not dispatch all state changing actions to all connections from the connection pool. The few dispatched actions are applied to already opened connections only. Lazy connections opened in the future are not affected. If, for example, the connection character set is changed using a PHP MySQL API call, the plugin will change the character set of all currently opened connection. It will not remember the character set change to apply it on lazy connections opened in the future. As a result the internal connection pool would hold connections using different character sets. This is not desired. Remember that character sets are taken into account for escaping.

`master_on_write` bool

If set, the plugin will use the master server only after the first statement has been executed on the master. Applications can still send

statements to the slaves using SQL hints to overrule the automatic decision.

The setting may help with replication lag. If an application runs an `INSERT` the plugin will, by default, use the master to execute all following statements, including `SELECT` statements. This helps to avoid problems with reads from slaves which have not replicated the `INSERT` yet.

`trx_stickiness` string

Transaction stickiness policy. Supported policies: `disabled` (default), `master`.

Experimental feature.

The setting requires 5.4.0 or newer. If used with PHP older than 5.4.0, the plugin will emit a warning like `(mysqlnd_ms) trx_stickiness strategy is not supported before PHP 5.3.99`.

If no transaction stickiness policy is set or, if setting `trx_stickiness=disabled`, the plugin is not transaction aware. Thus, the plugin may load balance connections and switch connections in the middle of a transaction. The plugin is not transaction safe. SQL hints must be used avoid connection switches during a transaction.

As of PHP 5.4.0 the mysqlnd library allows the plugin to monitor the `autocommit` mode set by calls to the libraries `trx_autocommit()` function. If setting `trx_stickiness=master` and `autocommit` gets disabled by a PHP MySQL extension invoking the `mysqlnd` library internal function call `trx_autocommit()`, the plugin is made aware of the begin of a transaction. Then, the plugin stops load balancing and directs all statements to the master server until `autocommit` is enabled. Thus, no SQL hints are required.

An example of a PHP MySQL API function calling the `mysqlnd` library internal function call `trx_autocommit()` is `mysqli_autocommit`.

Although setting `trx_stickiness=master`, the plugin cannot be made aware of `autocommit` mode changes caused by SQL statements such as `SET AUTOCOMMIT=0`.

7.6.4.4 Testing

> **Note**
>
> The section applies to mysqlnd_ms 1.1.0 or newer, not the 1.0 series.

The PECL/mysqlnd_ms test suite is in the `tests/` directory of the source distribution. The test suite consists of standard phpt tests, which are described on the PHP Quality Assurance Teams website.

Running the tests requires setting up one to four MySQL servers. Some tests don't connect to MySQL at all. Others require one server for testing. Some require two distinct servers. In some cases two servers are used to emulate a replication setup. In other cases a master and a slave of an existing MySQL replication setup are required for testing. The tests will try to detect how many servers and what kind of servers are given. If the required servers are not found, the test will be skipped automatically.

Before running the tests, edit `tests/config.inc` to configure the MySQL servers to be used for testing.

The most basic configuration is as follows.

```
putenv("MYSQL_TEST_HOST=localhost");
putenv("MYSQL_TEST_PORT=3306");
putenv("MYSQL_TEST_USER=root");
putenv("MYSQL_TEST_PASSWD=");
putenv("MYSQL_TEST_DB=test");
putenv("MYSQL_TEST_ENGINE=MyISAM");
putenv("MYSQL_TEST_SOCKET=");

putenv("MYSQL_TEST_SKIP_CONNECT_FAILURE=1");
putenv("MYSQL_TEST_CONNECT_FLAGS=0");
putenv("MYSQL_TEST_EXPERIMENTAL=0");

/* replication cluster emulation */
putenv("MYSQL_TEST_EMULATED_MASTER_HOST=". getenv("MYSQL_TEST_HOST"));
putenv("MYSQL_TEST_EMULATED_SLAVE_HOST=". getenv("MYSQL_TEST_HOST"));

/* real replication cluster */
putenv("MYSQL_TEST_MASTER_HOST=". getenv("MYSQL_TEST_EMULATED_MASTER_HOST"));
putenv("MYSQL_TEST_SLAVE_HOST=". getenv("MYSQL_TEST_EMULATED_SLAVE_HOST"));
```

`MYSQL_TEST_HOST`, `MYSQL_TEST_PORT` and `MYSQL_TEST_SOCKET` define the hostname, TCP/IP port and Unix domain socket of the default database server. `MYSQL_TEST_USER` and `MYSQL_TEST_PASSWD` contain the user and password needed to connect to the database/schema configured with `MYSQL_TEST_DB`. All configured servers must have the same database user configured to give access to the test database.

Using `host`, `host:port` or `host:/path/to/socket` syntax one can set an alternate host, host and port or host and socket for any of the servers.

```
putenv("MYSQL_TEST_SLAVE_HOST=192.168.78.136:3307"));
putenv("MYSQL_TEST_MASTER_HOST=myserver_hostname:/path/to/socket"));
```

7.6.4.5 Debugging and Tracing

The mysqlnd debug log can be used to debug and trace the actitivities of PECL/mysqlnd_ms. As a mysqlnd PECL/mysqlnd_ms adds trace information to the mysqlnd library debug file. Please, see the `mysqlnd.debug` PHP configuration directive documentation for a detailed description on how to configure the debug log.

Configuration setting example to activate the debug log:

```
mysqlnd.debug=d:t:x:O,/tmp/mysqlnd.trace
```

> **Note**
>
> This feature is only available with a debug build of PHP. Works on Microsoft Windows if using a debug build of PHP and PHP was built using Microsoft Visual C version 9 and above.

The debug log shows mysqlnd library and PECL/mysqlnd_ms plugin function calls, similar to a trace log. Mysqlnd library calls are usually prefixed with `mysqlnd_`. PECL/mysqlnd internal calls begin with `mysqlnd_ms`.

Example excerpt from the debug log (connect):

```
[...]
>mysqlnd_connect
| info : host=myapp user=root db=test port=3306 flags=131072
| >mysqlnd_ms::connect
| | >mysqlnd_ms_config_json_section_exists
| | | info : section=[myapp] len=[5]
| | | >mysqlnd_ms_config_json_sub_section_exists
| | | | info : section=[myapp] len=[5]
| | | | info : ret=1
| | | <mysqlnd_ms_config_json_sub_section_exists
| | | info : ret=1
| | <mysqlnd_ms_config_json_section_exists
[...]
```

The debug log is not only useful for plugin developers but also to find the cause of user errors. For example, if your application does not do proper error handling and fails to record error messages, checking the debug and trace log may help finding the cause. Use of the debug log to debug application issues should be considered only if no other option is available. Writing the debug log to disk is a slow operation and may have negative impact on the application performance.

Example excerpt from the debug log (connection failure):

```
[...]
| | | | | | | info : adding error [Access denied for user 'root'@'localhost' (using password: YES)] to the
| | | | | | | info : PACKET_FREE(0)
| | | | | | | info : PACKET_FREE(0x7f3ef6323f50)
| | | | | | | info : PACKET_FREE(0x7f3ef6324080)
| | | | | | <mysqlnd_auth_handshake
| | | | | | info : switch_to_auth_protocol=n/a
| | | | | | info : conn->error_info.error_no = 1045
| | | | | <mysqlnd_connect_run_authentication
| | | | | info : PACKET_FREE(0x7f3ef63236d8)
| | | | | >mysqlnd_conn::free_contents
| | | | | | >mysqlnd_net::free_contents
| | | | | | <mysqlnd_net::free_contents
| | | | | | info : Freeing memory of members
| | | | | | info : scheme=unix:///tmp/mysql.sock
| | | | | | >mysqlnd_error_list_pdtor
| | | | | | <mysqlnd_error_list_pdtor
| | | | | <mysqlnd_conn::free_contents
| | | | <mysqlnd_conn::connect
[...]
```

The trace log can also be used to verify correct behaviour of PECL/mysqlnd_ms itself, for example, to check which server has been selected for query execution and why.

Example excerpt from the debug log (plugin decision):

```
[...]
>mysqlnd_ms::query
| info : query=DROP TABLE IF EXISTS test
| >_mysqlnd_plugin_get_plugin_connection_data
| | info : plugin_id=5
| <_mysqlnd_plugin_get_plugin_connection_data
| >mysqlnd_ms_pick_server_ex
| | info : conn_data=0x7fb6a7d3e5a0 *conn_data=0x7fb6a7d410d0
| | >mysqlnd_ms_select_servers_all
| | <mysqlnd_ms_select_servers_all
| | >mysqlnd_ms_choose_connection_rr
| | | >mysqlnd_ms_query_is_select
[...]
| | | <mysqlnd_ms_query_is_select
[...]
| | | info : Init the master context
| | | info : list(0x7fb6a7d3f598) has 1
| | | info : Using master connection
| | | >mysqlnd_ms_advanced_connect
| | | | >mysqlnd_conn::connect
| | | | | info : host=localhost user=root db=test port=3306 flags=131072 persistent=0 state=0
```

In this case the statement `DROP TABLE IF EXISTS test` has been executed. Note that the statement string is shown in the log file. You may want to take measures to restrict access to the log for security considerations.

The statement has been load balanced using round robin policy, as you can easily guess from the functions name `>mysqlnd_ms_choose_connection_rr`. It has been sent to a master server running on `host=localhost user=root db=test port=3306 flags=131072 persistent=0 state=0`.

7.6.4.6 Monitoring

Plugin activity can be monitored using the mysqlnd trace log, mysqlnd statistics, mysqlnd_ms plugin statistics and external PHP debugging tools. Use of the trace log should be limited to debugging. It is recommended to use the plugins statistics for monitoring.

Writing a trace log is a slow operation. If using an external PHP debugging tool, please refer to the vendors manual about its performance impact and the type of information collected. In many cases, external debugging tools will provide call stacks. Often, a call stack or a trace log is more difficult to interpret than the statistics provided by the plugin.

Plugin statistics tell how often which kind of cluster node has been used (slave or master), why the node was used, if lazy connections have been used and if global transaction ID injection has been performed. The monitoring information provided enables user to verify plugin decisions and to plan their cluster resources based on usage pattern. The function `mysqlnd_ms_get_stats` is used to access the statistics. Please, see the functions description for a list of available statistics.

Statistics are collected on a per PHP process basis. Their scope is a PHP process. Depending on the PHP deployment model a process may serve one or multiple web requests. If using CGI model, a PHP process serves one web request. If using FastCGI or pre-fork web server models, a PHP process usually serves

multiple web requests. The same is the case with a threaded web server. Please, note that threads running in parallel can update the statistics in parallel. Thus, if using a threaded PHP deployment model, statistics can be changed by more than one script at a time. A script cannot rely on the fact that it sees only its own changes to statistics.

Example 7.95 Verify plugin activity in a non-threaded deployment model

```
mysqlnd_ms.enable=1
mysqlnd_ms.collect_statistics=1
```

```
<?php
/* Load balanced following "myapp" section rules from the plugins config file (not shown) */
$mysqli = new mysqli("myapp", "username", "password", "database");
if (mysqli_connect_errno())
  /* Of course, your error handling is nicer... */
  die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));

$stats_before = mysqlnd_ms_get_stats();
if ($res = $mysqli->query("SELECT 'Read request' FROM DUAL")) {
  var_dump($res->fetch_all());
}
$stats_after = mysqlnd_ms_get_stats();
if ($stats_after['use_slave'] <= $stats_before['use_slave']) {
  echo "According to the statistics the read request has not been run on a slave!";
}
?>
```

Statistics are aggregated for all plugin activities and all connections handled by the plugin. It is not possible to tell how much a certain connection handle has contributed to the overall statistics.

Utilizing PHPs `register_shutdown_function` function or the `auto_append_file` PHP configuration directive it is easily possible to dump statistics into, for example, a log file when a script finishes. Instead of using a log file it is also possible to send the statistics to an external monitoring tool for recording and display.

Example 7.96 Recording statistics during shutdown

```
mysqlnd_ms.enable=1
mysqlnd_ms.collect_statistics=1
error_log=/tmp/php_errors.log
```

```
<?php
function check_stats() {
  $msg = str_repeat("-", 80) . "\n";
  $msg .= var_export(mysqlnd_ms_get_stats(), true) . "\n";
  $msg .= str_repeat("-", 80) . "\n";
  error_log($msg);
}
register_shutdown_function("check_stats");
?>
```

7.7 Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

SQL hint related

Example 7.97 Example demonstrating the usage of mysqlnd_ms constants

The mysqlnd replication and load balancing plugin (`mysqlnd_ms`) performs read/write splitting. This directs write queries to a MySQL master server, and read-only queries to the MySQL slave servers. The plugin has a built-in read/write split logic. All queries which start with `SELECT` are considered read-only queries, which are then sent to a MySQL slave server that is listed in the plugin configuration file. All other queries are directed to the MySQL master server that is also specified in the plugin configuration file.

User supplied SQL hints can be used to overrule automatic read/write splitting, to gain full control on the process. SQL hints are standards compliant SQL comments. The plugin will scan the beginning of a query string for an SQL comment for certain commands, which then control query redirection. Other systems involved in the query processing are unaffected by the SQL hints because other systems will ignore the SQL comments.

The plugin supports three SQL hints to direct queries to either the MySQL slave servers, the MySQL master server, or the last used MySQL server. SQL hints must be placed at the beginning of a query to be recognized by the plugin.

For better portability, it is recommended to use the string constants `MYSQLND_MS_MASTER_SWITCH`, `MYSQLND_MS_SLAVE_SWITCH` and `MYSQLND_MS_LAST_USED_SWITCH` instead of their literal values.

```
<?php
/* Use constants for maximum portability */
$master_query = "/*" . MYSQLND_MS_MASTER_SWITCH . "*/SELECT id FROM test";

/* Valid but less portable: using literal instead of constant */
$slave_query = "/*ms=slave*/SHOW TABLES";

printf("master_query = '%s'\n", $master_query);
printf("slave_query = '%s'\n", $slave_query);
?>
```

The above examples will output:

```
master_query = /*ms=master*/SELECT id FROM test
slave_query = /*ms=slave*/SHOW TABLES
```

`MYSQLND_MS_MASTER_SWITCH` (string)	SQL hint used to send a query to the MySQL replication master server.
`MYSQLND_MS_SLAVE_SWITCH` (string)	SQL hint used to send a query to one of the MySQL replication slave servers.

`MYSQLND_MS_LAST_USED_SWITCH` (string) SQL hint used to send a query to the last used MySQL server. The last used MySQL server can either be a master or a slave server in a MySQL replication setup.

`mysqlnd_ms_query_is_select` related

`MYSQLND_MS_QUERY_USE_MASTER` (integer) If `mysqlnd_ms_is_select` returns `MYSQLND_MS_QUERY_USE_MASTER` for a given query, the built-in read/write split mechanism recommends sending the query to a MySQL replication master server.

`MYSQLND_MS_QUERY_USE_SLAVE` (integer) If `mysqlnd_ms_is_select` returns `MYSQLND_MS_QUERY_USE_SLAVE` for a given query, the built-in read/write split mechanism recommends sending the query to a MySQL replication slave server.

`MYSQLND_MS_QUERY_USE_LAST_USED` (integer) If `mysqlnd_ms_is_select` returns `MYSQLND_MS_QUERY_USE_LAST_USED` for a given query, the built-in read/write split mechanism recommends sending the query to the last used server.

`mysqlnd_ms_set_qos`, quality of service filter and service level related

`MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL` (integer) Use to request the service level eventual consistency from the `mysqlnd_ms_set_qos`. Eventual consistency is the default quality of service when reading from an asynchronous MySQL replication slave. Data returned in this service level may or may not be stale, depending on whether the selected slaves happen to have replicated the latest changes from the MySQL replication master or not.

`MYSQLND_MS_QOS_CONSISTENCY_SESSION` (integer) Use to request the service level session consistency from the `mysqlnd_ms_set_qos`. Session consistency is defined as read your writes. The client is guaranteed to see his latest changes.

`MYSQLND_MS_QOS_CONSISTENCY_STRONG` (integer) Use to request the service level strong consistency from the `mysqlnd_ms_set_qos`. Strong consistency is used to ensure all clients see each others changes.

`MYSQLND_MS_QOS_OPTION_GTID` (integer) Used as a service level option with `mysqlnd_ms_set_qos` to parameterize session consistency.

`MYSQLND_MS_QOS_OPTION_AGE` (integer) Used as a service level option with `mysqlnd_ms_set_qos` to parameterize eventual consistency.

Other

The plugins version number can be obtained using `MYSQLND_MS_VERSION` or `MYSQLND_MS_VERSION_ID`. `MYSQLND_MS_VERSION` is the string representation of the numerical version number `MYSQLND_MS_VERSION_ID`, which is an integer such as 10000. Developers can calculate the version number as follows.

Version (part)	**Example**
Major*10000	1*10000 = 10000
Minor*100	0*100 = 0
Patch	0 = 0
MYSQLND_MS_VERSION_ID	10000

`MYSQLND_MS_VERSION` (string)	Plugin version string, for example, “1.0.0-prototype”.
`MYSQLND_MS_VERSION_ID` (integer)	Plugin version number, for example, 10000.

7.8 Mysqlnd_ms Functions

7.8.1 `mysqlnd_ms_dump_servers`

- `mysqlnd_ms_dump_servers`

  Returns a list of currently configured servers

Description

```
array mysqlnd_ms_dump_servers(
  mixed connection);
```

Returns a list of currently configured servers.

Parameters

`connection`	A MySQL connection handle obtained from any of the connect functions of the mysqli, mysql or PDO_MYSQL extensions.

Return Values

`FALSE` on error. Otherwise, returns an array with two entries `masters` and `slaves` each of which contains an array listing all corresponding servers.

The function can be used to check and debug the list of servers currently used by the plugin. It is mostly useful when the list of servers changes at runtime, for example, when using MySQL Fabric.

`masters` and `slaves` server entries

Key	Description	Version
`name_from_config`	Server entry name from config, if appliciable. NULL if no configuration name is available.	Since 1.6.0.
`hostname`	Host name of the server.	Since 1.6.0.
`user`	Database user used to authenticate against the server.	Since 1.6.0.
`port`	TCP/IP port of the server.	Since 1.6.0.
`socket`	Unix domain socket of the server.	Since 1.6.0.

Notes

> **Note**
>
> `mysqlnd_ms_dump_servers` requires PECL mysqlnd_ms >> 1.6.0.

Examples

Example 7.98 `mysqlnd_ms_dump_servers` example

```
{
    "myapp": {
        "master": {
            "master1": {
                "host":"master1_host",
                "port":"master1_port",
                "socket":"master1_socket",
                "db":"master1_db",
                "user":"master1_user",
                "password":"master1_pw"
            }
        },
        "slave": {
             "slave_0": {
                 "host":"slave0_host",
                 "port":"slave0_port",
                 "socket":"slave0_socket",
                 "db":"slave0_db",
                 "user":"slave0_user",
                 "password":"slave0_pw"
             },
             "slave_1": {
                 "host":"slave1_host"
             }
        }
    }
}
```

```
<?php
$link = mysqli_connect("myapp", "global_user", "global_pass", "global_db", 1234, "global_socket");
var_dump(mysqlnd_ms_dump_servers($link);
?>
```

The above example will output:

```
array(2) {
  ["masters"]=>
  array(1) {
    [0]=>
    array(5) {
      ["name_from_config"]=>
      string(7) "master1"
      ["hostname"]=>
      string(12) "master1_host"
      ["user"]=>
      string(12) "master1_user"
      ["port"]=>
      int(3306)
      ["socket"]=>
      string(14) "master1_socket"
    }
  }
  ["slaves"]=>
  array(2) {
    [0]=>
```

```
    array(5) {
      ["name_from_config"]=>
      string(7) "slave_0"
      ["hostname"]=>
      string(11) "slave0_host"
      ["user"]=>
      string(11) "slave0_user"
      ["port"]=>
      int(3306)
      ["socket"]=>
      string(13) "slave0_socket"
    }
    [1]=>
    array(5) {
      ["name_from_config"]=>
      string(7) "slave_1"
      ["hostname"]=>
      string(11) "slave1_host"
      ["user"]=>
      string(12) "gloabal_user"
      ["port"]=>
      int(1234)
      ["socket"]=>
      string(13) "global_socket"
    }
  }
}
```

7.8.2 mysqlnd_ms_fabric_select_global

- mysqlnd_ms_fabric_select_global

  Switch to global sharding server for a given table

Description

```
array mysqlnd_ms_fabric_select_global(
  mixed connection,
  mixed table_name);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

MySQL Fabric related.

Switch the connection to the nodes handling global sharding queries for the given table name.

Parameters

connection	A MySQL connection handle obtained from any of the connect functions of the mysqli, mysql or PDO_MYSQL extensions.
table_name	The table name to ask Fabric about.

Return Values

FALSE on error. Otherwise, TRUE

Notes

> **Note**
>
> mysqlnd_ms_fabric_select_global requires PECL mysqlnd_ms >> 1.6.0.

7.8.3 mysqlnd_ms_fabric_select_shard

- mysqlnd_ms_fabric_select_shard

  Switch to shard

Description

```
array mysqlnd_ms_fabric_select_shard(
  mixed connection,
  mixed table_name,
  mixed shard_key);
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

MySQL Fabric related.

Switch the connection to the shards responsible for the given table name and shard key.

Parameters

connection	A MySQL connection handle obtained from any of the connect functions of the mysqli, mysql or PDO_MYSQL extensions.
table_name	The table name to ask Fabric about.
shard_key	The shard key to ask Fabric about.

Return Values

FALSE on error. Otherwise, TRUE

Notes

> **Note**
>
> mysqlnd_ms_fabric_select_shard requires PECL mysqlnd_ms >> 1.6.0.

7.8.4 mysqlnd_ms_get_last_gtid

- mysqlnd_ms_get_last_gtid

  Returns the latest global transaction ID

Description

```
string mysqlnd_ms_get_last_gtid(
  mixed connection);
```

Returns a global transaction identifier which belongs to a write operation no older than the last write performed by the client. It is not guaranteed that the global transaction identifier is identical to that one created for the last write transaction performed by the client.

Parameters

connection	A PECL/mysqlnd_ms connection handle to a MySQL server of the type PDO_MYSQL, mysqli> or ext/mysql. The connection handle is obtained when opening a connection with a host name that matches a mysqlnd_ms configuration file entry using any of the above three MySQL driver extensions.

Return Values

Returns a global transaction ID (GTID) on success. Otherwise, returns FALSE.

The function mysqlnd_ms_get_last_gtid returns the GTID obtained when executing the SQL statement from the fetch_last_gtid entry of the global_transaction_id_injection section from the plugins configuration file.

The function may be called after the GTID has been incremented.

Notes

> **Note**
>
> mysqlnd_ms_get_last_gtid requires PHP >= 5.4.0 and PECL mysqlnd_ms >= 1.2.0. Internally, it is using a mysqlnd library C functionality not available with PHP 5.3.
>
> Please note, all MySQL 5.6 production versions do not provide clients with enough information to use GTIDs for enforcing session consistency. In the worst case, the plugin will choose the master only.

Examples

Example 7.99 mysqlnd_ms_get_last_gtid example

```
<?php
/* Open mysqlnd_ms connection using mysqli, PDO_MySQL or mysql extension */
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli)
  /* Of course, your error handling is nicer... */
  die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));

/* auto commit mode, transaction on master, GTID must be incremented */
if (!$mysqli->query("DROP TABLE IF EXISTS test"))
  die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));

printf("GTID after transaction %s\n", mysqlnd_ms_get_last_gtid($mysqli));

/* auto commit mode, transaction on master, GTID must be incremented */
if (!$mysqli->query("CREATE TABLE test(id INT)"))
  die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));

printf("GTID after transaction %s\n", mysqlnd_ms_get_last_gtid($mysqli));
?>
```

See Also

Global Transaction IDs

7.8.5 mysqlnd_ms_get_last_used_connection

- mysqlnd_ms_get_last_used_connection

  Returns an array which describes the last used connection

Description

```
array mysqlnd_ms_get_last_used_connection(
  mixed connection);
```

Returns an array which describes the last used connection from the plugins connection pool currently pointed to by the user connection handle. If using the plugin, a user connection handle represents a pool of database connections. It is not possible to tell from the user connection handles properties to which database server from the pool the user connection handle points.

The function can be used to debug or monitor PECL mysqlnd_ms.

Parameters

connection	A MySQL connection handle obtained from any of the connect functions of the mysqli, mysql or PDO_MYSQL extensions.

Return Values

FALSE on error. Otherwise, an array which describes the connection used to execute the last statement on.

Array which describes the connection.

Property	Description	Version
scheme	Connection scheme. Either tcp://host:port or unix://host:socket. If you want to distinguish connections from each other use a combination of scheme and thread_id as a unique key. Neither scheme nor thread_id alone are sufficient to distinguish two connections from each other. Two servers may assign the same thread_id to two different connections. Thus, connections in the pool may have the same thread_id. Also, do not rely on uniqueness of scheme in a pool. Your QA engineers may use the same MySQL server instance for two distinct logical roles and add it multiple times to the pool. This hack is used, for example, in the test suite.	Since 1.1.0.
host	Database server host used with the connection. The host is only set with TCP/IP connections. It is empty with Unix domain or Windows named pipe connections,	Since 1.1.0.
host_info	A character string representing the server hostname and the connection type.	Since 1.1.2.
port	Database server port used with the connection.	Since 1.1.0.
socket_or_pipe	Unix domain socket or Windows named pipe used with the connection. The value is empty for TCP/IP connections.	Since 1.1.2.
thread_id	Connection thread id.	Since 1.1.0.
last_message	Info message obtained from the MySQL C API function mysql_info(). Please, see mysqli_info for a description.	Since 1.1.0.
errno	Error code.	Since 1.1.0.

Property	Description	Version
error	Error message.	Since 1.1.0.
sqlstate	Error SQLstate code.	Since 1.1.0.

Notes

> **Note**
>
> mysqlnd_ms_get_last_used_connection requires PHP >= 5.4.0 and PECL mysqlnd_ms >> 1.1.0. Internally, it is using a mysqlnd library C call not available with PHP 5.3.

Examples

The example assumes that myapp refers to a plugin configuration file section and represents a connection pool.

Example 7.100 mysqlnd_ms_get_last_used_connection example

```
<?php
$link = new mysqli("myapp", "user", "password", "database");
$res = $link->query("SELECT 1 FROM DUAL");
var_dump(mysqlnd_ms_get_last_used_connection($link));
?>
```

The above example will output:

```
array(10) {
  ["scheme"]=>
  string(22) "unix:///tmp/mysql.sock"
  ["host_info"]=>
  string(25) "Localhost via UNIX socket"
  ["host"]=>
  string(0) ""
  ["port"]=>
  int(3306)
  ["socket_or_pipe"]=>
  string(15) "/tmp/mysql.sock"
  ["thread_id"]=>
  int(46253)
  ["last_message"]=>
  string(0) ""
  ["errno"]=>
  int(0)
  ["error"]=>
  string(0) ""
  ["sqlstate"]=>
  string(5) "00000"
}
```

7.8.6 mysqlnd_ms_get_stats

- mysqlnd_ms_get_stats

Returns query distribution and connection statistics

Description

```
array mysqlnd_ms_get_stats();
```

Returns an array of statistics collected by the replication and load balancing plugin.

The PHP configuration setting `mysqlnd_ms.collect_statistics` controls the collection of statistics. The collection of statistics is disabled by default for performance reasons.

The scope of the statistics is the `PHP` process. Depending on your deployment model a `PHP` process may handle one or multiple requests.

Statistics are aggregated for all connections and all storage handler. It is not possible to tell how much queries originating from `mysqli`, `PDO_MySQL` or `mysql` API calls have contributed to the aggregated data values.

Parameters

This function has no parameters.

Return Values

Returns `NULL` if the PHP configuration directive `mysqlnd_ms.enable` has disabled the plugin. Otherwise, returns array of statistics.

Array of statistics

Statistic	Description	Version
`use_slave`	The semantics of this statistic has changed between 1.0.1 - 1.1.0. The meaning for version 1.0.1 is as follows. Number of statements considered as read-only by the built-in query analyzer. Neither statements which begin with a SQL hint to force use of slave nor statements directed to a slave by an user-defined callback are included. The total number of statements sent to the slaves is `use_slave` + `use_slave_sql_hint` + `use_slave_callback`. PECL/mysqlnd_ms 1.1.0 introduces a new concept of chained filters. The statistics is now set by the internal load balancing filter. With version 1.1.0 the load balancing filter is always the last in the filter chain, if used. In future versions a load balancing filter may be followed by other filters causing another change in the meaning of the statistic. If, in the future, a load balancing filter is followed by another filter it is no longer guaranteed that the statement, which increments `use_slave`, will be executed on the slaves. The meaning for version 1.1.0 is as follows. Number of statements sent to the slaves. Statements directed to a slave by the user filter (an user-defined callback) are not included. The latter are counted by `use_slave_callback`.	Since 1.0.0.
`use_master`	The semantics of this statistic has changed between 1.0.1 - 1.1.0. The meaning for version 1.0.1 is as follows. Number of statements not considered as read-only by the built-in query analyzer. Neither statements which begin with a SQL hint to force use of master nor	Since 1.0.0.

Statistic	Description	Version
	statements directed to a master by an user-defined callback are included. The total number of statements sent to the master is `use_master` + `use_master_sql_hint` + `use_master_callback`. PECL/mysqlnd_ms 1.1.0 introduces a new concept of chained filters. The statictics is now set by the internal load balancing filter. With version 1.1.0 the load balancing filter is always the last in the filter chain, if used. In future versions a load balancing filter may be followed by other filters causing another change in the meaning of the statistic. If, in the future, a load balancing filter is followed by another filter it is no longer guaranteed that the statement, which increments `use_master`, will be executed on the slaves. The meaning for version 1.1.0 is as follows. Number of statements sent to the masters. Statements directed to a master by the user filter (an user-defined callback) are not included. The latter are counted by `use_master_callback`.	
`use_slave_guess`	Number of statements the built-in query analyzer recommends sending to a slave because they contain no SQL hint to force use of a certain server. The recommendation may be overruled in the following. It is not guaranteed whether the statement will be executed on a slave or not. This is how often the internal `is_select` function has guessed that a slave shall be used. Please, see also the user space function `mysqlnd_ms_query_is_select`.	Since 1.1.0.
`use_master_guess`	Number of statements the built-in query analyzer recommends sending to a master because they contain no SQL hint to force use of a certain server. The recommendation may be overruled in the following. It is not guaranteed whether the statement will be executed on a slave or not. This is how often the internal `is_select` function has guessed that a master shall be used. Please, see also the user space function `mysqlnd_ms_query_is_select`.	Since 1.1.0.
`use_slave_sql_hint`	Number of statements sent to a slave because statement begins with the SQL hint to force use of slave.	Since 1.0.0.
`use_master_sql_hint`	Number of statements sent to a master because statement begins with the SQL hint to force use of master.	Since 1.0.0.
`use_last_used_sql_hint`	Number of statements sent to server which has run the previous statement, because statement begins with the SQL hint to force use of previously used server.	Since 1.0.0.
`use_slave_callback`	Number of statements sent to a slave because an user-defined callback has chosen a slave server for statement execution.	Since 1.0.0.
`use_master_callback`	Number of statements sent to a master because an user-defined callback has chosen a master server for statement execution.	Since 1.0.0.
`non_lazy_connections_slave_success`	Number of successfully opened slave connections from configurations not using `lazy connections`. The total number of successfully opened slave connections is `non_lazy_connections_slave_success` + `lazy_connections_slave_success`	Since 1.0.0.
`non_lazy_connections_slave_failure`	Number of failed slave connection attempts from configurations not using `lazy connections`. The total number of failed slave	Since 1.0.0.

Statistic	Description	Version
	connection attempts is non_lazy_connections_slave_failure + lazy_connections_slave_failure	
non_lazy_connections_master_success	Number of successfully opened master connections from configurations not using lazy connections. The total number of successfully opened master connections is non_lazy_connections_master_success + lazy_connections_master_success	Since 1.0.0.
non_lazy_connections_master_failure	Number of failed master connection attempts from configurations not using lazy connections. The total number of failed master connection attempts is non_lazy_connections_master_failure + lazy_connections_master_failure	Since 1.0.0.
lazy_connections_slave_success	Number of successfully opened slave connections from configurations using lazy connections.	Since 1.0.0.
lazy_connections_slave_failure	Number of failed slave connection attempts from configurations using lazy connections.	Since 1.0.0.
lazy_connections_master_success	Number of successfully opened master connections from configurations using lazy connections.	Since 1.0.0.
lazy_connections_master_failure	Number of failed master connection attempts from configurations using lazy connections.	Since 1.0.0.
trx_autocommit_on	Number of autocommit mode activations via API calls. This figure may be used to monitor activity related to the plugin configuration setting trx_stickiness. If, for example, you want to know if a certain API call invokes the mysqlnd library function trx_autocommit(), which is a requirement for trx_stickiness, you may call the user API function in question and check if the statistic has changed. The statistic is modified only by the plugins internal subclassed trx_autocommit() method.	Since 1.0.0.
trx_autocommit_off	Number of autocommit mode deactivations via API calls.	Since 1.0.0.
trx_master_statements	Number of statements redirected to the master while trx_stickiness=master and autocommit mode is disabled.	Since 1.0.0.
gtid_autocommit_injections_success	Number of successful SQL injections in autocommit mode as part of the plugins client-side global transaction id emulation.	Since 1.2.0.
gtid_autocommit_injections_failure	Number of failed SQL injections in autocommit mode as part of the plugins client-side global transaction id emulation.	Since 1.2.0.
gtid_commit_injections_success	Number of successful SQL injections in commit mode as part of the plugins client-side global transaction id emulation.	Since 1.2.0.
gtid_commit_injections_failure	Number of failed SQL injections in commit mode as part of the plugins client-side global transaction id emulation.	Since 1.2.0.
gtid_implicit_commit_injections_success	Number of successful SQL injections when implicit commit is detected as part of the plugins client-side global transaction id emulation. Implicit commit happens, for example, when autocommit has been turned off, a query is executed and autocommit is enabled again. In that case, the statement will be committed by the server and SQL to maintain is injected before the autocommit is re-enabled. Another sequence causing an implicit commit is begin(), query(), begin(). The second call to begin() will implicitly commit the transaction started by the first call to begin(). begin() refers to internal library calls not actual PHP user API calls.	Since 1.2.0.

Statistic	Description	Version
gtid_implicit_commit_injections_failed	Number of failed SQL injections when implicit commit is detected as part of the plugins client-side global transaction id emulation. Implicit commit happens, for example, when autocommit has been turned off, a query is executed and autocommit is enabled again. In that case, the statement will be committed by the server and SQL to maintain is injected before the autocommit is re-enabled.	Since 1.2.0.
transient_error_retries	How often an operation has been retried when a transient error was detected. See also, transient_error plugin configuration file setting.	Since 1.6.0.
fabric_sharding_lookup_servers_success	Number of successful sharding.lookup_servers remote procedure calls to MySQL Fabric. A call is considered successful if the plugin could reach MySQL Fabric and got any reply. The reply itself may or may not be understood by the plugin. Success refers to the network transport only. If the reply was not understood or indicates a valid error condition, fabric_sharding_lookup_servers_xml_failure gets incremented.	Since 1.6.0.
fabric_sharding_lookup_servers_failure	Number of failed sharding.lookup_servers remote procedure calls to MySQL Fabric. A remote procedure call is considered failed if there was a network error in connecting to, writing to or reading from MySQL Fabric.	Since 1.6.0.
fabric_sharding_lookup_servers_time	Time spent connecting to, writing to and reading from MySQL Fabrich during the sharding.lookup_servers remote procedure call. The value is aggregated for all calls. Time is measured in microseconds.	Since 1.6.0.
fabric_sharding_lookup_servers_bytes_received	Total number of bytes received from MySQL Fabric in reply to sharding.lookup_servers calls.	Since 1.6.0.
fabric_sharding_lookup_servers_xml_failure	How often a reply from MySQL Fabric to sharding.lookup_servers calls was not understood. Please note, the current experimental implementation does not distinguish between valid errors returned and malformed replies.	Since 1.6.0.
xa_begin	How many XA/distributed transactions have been started using mysqlnd_ms_xa_begin.	Since 1.6.0.
xa_commit_success	How many XA/distributed transactions have been successfully committed using mysqlnd_ms_xa_commit.	Since 1.6.0.
xa_commit_failure	How many XA/distributed transactions failed to commit during mysqlnd_ms_xa_commit.	Since 1.6.0.
xa_rollback_success	How many XA/distributed transactions have been successfully rolled back using mysqlnd_ms_xa_rollback. The figure does not include implict rollbacks performed as a result of mysqlnd_ms_xa_commit failure.	Since 1.6.0.
xa_rollback_failure	How many XA/distributed transactions could not be rolled back. This includes failures of mysqlnd_ms_xa_rollback but also failured during rollback when closing a connection, if rollback_on_close is set. Please, see also xa_rollback_on_close below.	Since 1.6.0.
xa_participants	Total number of participants in any XA transaction started with mysqlnd_ms_xa_begin.	Since 1.6.0.
xa_rollback_on_close	How many XA transactions have been rolled back implicitly when a connection was close and rollback_on_close is set. Depending on your coding policies, this may hint a flaw in your code as you may prefer to explicitly clean up resources.	Since 1.6.0.

Statistic	Description	Version
pool_master_total	Number of master servers (connections) in the internal connection pool.	Since 1.6.0.
pool_slave_total	Number of slave servers (connections) in the internal connection pool.	Since 1.6.0.
pool_master_active	Number of master servers (connections) from the internal connection pool which are currently used for picking a connection.	Since 1.6.0.
pool_slave_active	Number of slave servers (connections) from the internal connection pool which are currently used for picking a connection.	Since 1.6.0.
pool_update	How often the active connection list has been replaced and a new set of master and slave servers had been installed.	Since 1.6.0.
pool_master_reactivated	How often a master connection has been reused after being flushed from the active list.	Since 1.6.0.
pool_slave_reactivated	How often a slave connection has been reused after being flushed from the active list.	Since 1.6.0.

Examples

Example 7.101 `mysqlnd_ms_get_stats` example

```
<?php
printf("mysqlnd_ms.enable = %d\n", ini_get("mysqlnd_ms.enable"));
printf("mysqlnd_ms.collect_statistics = %d\n", ini_get("mysqlnd_ms.collect_statistics"));
var_dump(mysqlnd_ms_get_stats());
?>
```

The above example will output:

```
mysqlnd_ms.enable = 1
mysqlnd_ms.collect_statistics = 1
array(26) {
  ["use_slave"]=>
  string(1) "0"
  ["use_master"]=>
  string(1) "0"
  ["use_slave_guess"]=>
  string(1) "0"
  ["use_master_guess"]=>
  string(1) "0"
  ["use_slave_sql_hint"]=>
  string(1) "0"
  ["use_master_sql_hint"]=>
  string(1) "0"
  ["use_last_used_sql_hint"]=>
  string(1) "0"
  ["use_slave_callback"]=>
  string(1) "0"
  ["use_master_callback"]=>
  string(1) "0"
  ["non_lazy_connections_slave_success"]=>
  string(1) "0"
  ["non_lazy_connections_slave_failure"]=>
  string(1) "0"
  ["non_lazy_connections_master_success"]=>
  string(1) "0"
  ["non_lazy_connections_master_failure"]=>
```

```
  string(1) "0"
  ["lazy_connections_slave_success"]=>
  string(1) "0"
  ["lazy_connections_slave_failure"]=>
  string(1) "0"
  ["lazy_connections_master_success"]=>
  string(1) "0"
  ["lazy_connections_master_failure"]=>
  string(1) "0"
  ["trx_autocommit_on"]=>
  string(1) "0"
  ["trx_autocommit_off"]=>
  string(1) "0"
  ["trx_master_forced"]=>
  string(1) "0"
  ["gtid_autocommit_injections_success"]=>
  string(1) "0"
  ["gtid_autocommit_injections_failure"]=>
  string(1) "0"
  ["gtid_commit_injections_success"]=>
  string(1) "0"
  ["gtid_commit_injections_failure"]=>
  string(1) "0"
  ["gtid_implicit_commit_injections_success"]=>
  string(1) "0"
  ["gtid_implicit_commit_injections_failure"]=>
  string(1) "0"
  ["transient_error_retries"]=>
  string(1) "0"
}
```

See Also

Runtime configuration
mysqlnd_ms.collect_statistics
mysqlnd_ms.enable
Monitoring

7.8.7 mysqlnd_ms_match_wild

- `mysqlnd_ms_match_wild`

  Finds whether a table name matches a wildcard pattern or not

Description

```
  bool mysqlnd_ms_match_wild(
    string table_name,
    string wildcard);
```

Finds whether a table name matches a wildcard pattern or not.

This function is not of much practical relevance with PECL mysqlnd_ms 1.1.0 because the plugin does not support MySQL replication table filtering yet.

Parameters

`table_name`	The table name to check if it is matched by the wildcard.

wildcard — The wildcard pattern to check against the table name. The wildcard pattern supports the same placeholders as MySQL replication filters do.

MySQL replication filters can be configured by using the MySQL Server configuration options `--replicate-wild-do-table` and `--replicate-wild-do-db`. Please, consult the MySQL Reference Manual to learn more about this MySQL Server feature.

The supported placeholders are:

- `%` - zero or more literals
- `_` - one literal

Placeholders can be escaped using `\`.

Return Values

Returns `TRUE` `table_name` is matched by `wildcard`. Otherwise, returns `FALSE`

Examples

Example 7.102 `mysqlnd_ms_match_wild` example

```
<?php
var_dump(mysqlnd_ms_match_wild("schema_name.table_name", "schema%"));
var_dump(mysqlnd_ms_match_wild("abc", "_"));
var_dump(mysqlnd_ms_match_wild("table1", "table_"));
var_dump(mysqlnd_ms_match_wild("asia_customers", "%customers"));
var_dump(mysqlnd_ms_match_wild("funny%table","funny\%table"));
var_dump(mysqlnd_ms_match_wild("funnytable", "funny%table"));
?>
```

The above example will output:

```
bool(true)
bool(false)
bool(true)
bool(true)
bool(true)
bool(true)
```

7.8.8 `mysqlnd_ms_query_is_select`

- `mysqlnd_ms_query_is_select`

  Find whether to send the query to the master, the slave or the last used MySQL server

Description

```
int mysqlnd_ms_query_is_select(
  string query);
```

Finds whether to send the query to the master, the slave or the last used MySQL server.

The plugins built-in read/write split mechanism will be used to analyze the query string to make a recommendation where to send the query. The built-in read/write split mechanism is very basic and simple. The plugin will recommend sending all queries to the MySQL replication master server but those which begin with `SELECT`, or begin with a SQL hint which enforces sending the query to a slave server. Due to the basic but fast algorithm the plugin may propose to run some read-only statements such as `SHOW TABLES` on the replication master.

Parameters

`query`	Query string to test.

Return Values

A return value of `MYSQLND_MS_QUERY_USE_MASTER` indicates that the query should be send to the MySQL replication master server. The function returns a value of `MYSQLND_MS_QUERY_USE_SLAVE` if the query can be run on a slave because it is considered read-only. A value of `MYSQLND_MS_QUERY_USE_LAST_USED` is returned to recommend running the query on the last used server. This can either be a MySQL replication master server or a MySQL replication slave server.

If read write splitting has been disabled by setting `mysqlnd_ms.disable_rw_split`, the function will always return `MYSQLND_MS_QUERY_USE_MASTER` or `MYSQLND_MS_QUERY_USE_LAST_USED`.

Examples

Example 7.103 `mysqlnd_ms_query_is_select` example

```
<?php
function is_select($query)
{
 switch (mysqlnd_ms_query_is_select($query))
 {
  case MYSQLND_MS_QUERY_USE_MASTER:
   printf("'%s' should be run on the master.\n", $query);
   break;
  case MYSQLND_MS_QUERY_USE_SLAVE:
   printf("'%s' should be run on a slave.\n", $query);
   break;
  case MYSQLND_MS_QUERY_USE_LAST_USED:
   printf("'%s' should be run on the server that has run the previous query\n", $query);
   break;
  default:
   printf("No suggestion where to run the '%s', fallback to master recommended\n", $query);
   break;
 }
}

is_select("INSERT INTO test(id) VALUES (1)");
is_select("SELECT 1 FROM DUAL");
is_select("/*" . MYSQLND_MS_LAST_USED_SWITCH . "*/SELECT 2 FROM DUAL");
?>
```

The above example will output:

```
INSERT INTO test(id) VALUES (1) should be run on the master.
```

```
SELECT 1 FROM DUAL should be run on a slave.
/*ms=last_used*/SELECT 2 FROM DUAL should be run on the server that has run the previous query
```

See Also

Predefined Constants
`user` filter

Runtime configuration
mysqlnd_ms.disable_rw_split
mysqlnd_ms.enable

7.8.9 `mysqlnd_ms_set_qos`

- `mysqlnd_ms_set_qos`

  Sets the quality of service needed from the cluster

Description

```
bool mysqlnd_ms_set_qos(
  mixed connection,
  int service_level,
  int service_level_option,
  mixed option_value);
```

Sets the quality of service needed from the cluster. A database cluster delivers a certain quality of service to the user depending on its architecture. A major aspect of the quality of service is the consistency level the cluster can offer. An asynchronous MySQL replication cluster defaults to eventual consistency for slave reads: a slave may serve stale data, current data, or it may have not the requested data at all, because it is not synchronous to the master. In a MySQL replication cluster, only master accesses can give strong consistency, which promises that all clients see each others changes.

PECL/mysqlnd_ms hides the complexity of choosing appropriate nodes to achieve a certain level of service from the cluster. The "Quality of Service" filter implements the necessary logic. The filter can either be configured in the plugins configuration file, or at runtime using `mysqlnd_ms_set_qos`.

Similar results can be achieved with PECL mysqlnd_ms < 1.2.0, if using SQL hints to force the use of a certain type of node or using the `master_on_write` plugin configuration option. The first requires more code and causes more work on the application side. The latter is less refined than using the quality of service filter. Settings made through the function call can be reversed, as shown in the example below. The example temporarily switches to a higher service level (session consistency, read your writes) and returns back to the clusters default after it has performed all operations that require the better service. This way, read load on the master can be minimized compared to using `master_on_write`, which would continue using the master after the first write.

Since 1.5.0 calls will fail when done in the middle of a transaction if transaction stickiness is enabled and transaction boundaries have been detected. properly.

Parameters

`connection`	A PECL/mysqlnd_ms connection handle to a MySQL server of the type PDO_MYSQL, mysqli or ext/mysql for which a service level is to be set. The connection handle is obtained when opening a connection with a

	host name that matches a mysqlnd_ms configuration file entry using any of the above three MySQL driver extensions.
service_level	The requested service level: MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL, MYSQLND_MS_QOS_CONSISTENCY_SESSION or MYSQLND_MS_QOS_CONSISTENCY_STRONG.
service_level_option	An option to parameterize the requested service level. The option can either be MYSQLND_MS_QOS_OPTION_GTID or MYSQLND_MS_QOS_OPTION_AGE. The option MYSQLND_MS_QOS_OPTION_GTID can be used to refine the service level MYSQLND_MS_QOS_CONSISTENCY_SESSION. It must be combined with a fourth function parameter, the *option_value*. The *option_value* shall be a global transaction ID obtained from mysqlnd_ms_get_last_gtid. If set, the plugin considers both master servers and asynchronous slaves for session consistency (read your writes). Otherwise, only masters are used to achieve session consistency. A slave is considered up-to-date and checked if it has already replicated the global transaction ID from *option_value*. Please note, searching appropriate slaves is an expensive and slow operation. Use the feature sparsely, if the master cannot handle the read load alone. The MYSQLND_MS_QOS_OPTION_AGE option can be combined with the MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL service level, to filter out asynchronous slaves that lag more seconds behind the master than *option_value*. If set, the plugin will only consider slaves for reading if SHOW SLAVE STATUS reports Slave_IO_Running=Yes, Slave_SQL_Running=Yes and Seconds_Behind_Master <= option_value. Please note, searching appropriate slaves is an expensive and slow operation. Use the feature sparsely in version 1.2.0. Future versions may improve the algorithm used to identify candidates. Please, see the MySQL reference manual about the precision, accuracy and limitations of the MySQL administrative command SHOW SLAVE STATUS.
option_value	Parameter value for the service level option. See also the *service_level_option* parameter.

Return Values

Returns TRUE if the connections service level has been switched to the requested. Otherwise, returns FALSE

Notes

> **Note**
>
> mysqlnd_ms_set_qos requires PHP >= 5.4.0 and PECL mysqlnd_ms >= 1.2.0. Internally, it is using a mysqlnd library C functionality not available with PHP 5.3.
>
> Please note, all MySQL 5.6 production versions do not provide clients with enough information to use GTIDs for enforcing session consistency. In the worst case, the plugin will choose the master only.

Examples

Example 7.104 `mysqlnd_ms_set_qos` example

```
<?php
/* Open mysqlnd_ms connection using mysqli, PDO_MySQL or mysql extension */
$mysqli = new mysqli("myapp", "username", "password", "database");
if (!$mysqli)
  /* Of course, your error handling is nicer... */
  die(sprintf("[%d] %s\n", mysqli_connect_errno(), mysqli_connect_error()));

/* Session consistency: read your writes */
$ret = mysqlnd_ms_set_qos($mysqli, MYSQLND_MS_QOS_CONSISTENCY_SESSION);
if (!$ret)
  die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));

/* Will use master and return fresh data, client can see his last write */
if (!$res = $mysqli->query("SELECT item, price FROM orders WHERE order_id = 1"))
  die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));

/* Back to default: use of all slaves and masters permitted, stale data can happen */
if (!mysqlnd_ms_set_qos($mysqli, MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL))
  die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));
?>
```

See Also

`mysqlnd_ms_get_last_gtid`
Service level and consistency concept
Filter concept

7.8.10 `mysqlnd_ms_set_user_pick_server`

- `mysqlnd_ms_set_user_pick_server`

  Sets a callback for user-defined read/write splitting

Description

```
  bool mysqlnd_ms_set_user_pick_server(
    string function);
```

Sets a callback for user-defined read/write splitting. The plugin will call the callback only if `pick[]=user` is the default rule for server picking in the relevant section of the plugins configuration file.

The plugins built-in read/write query split mechanism decisions can be overwritten in two ways. The easiest way is to prepend the query string with the SQL hints `MYSQLND_MS_MASTER_SWITCH`, `MYSQLND_MS_SLAVE_SWITCH` or `MYSQLND_MS_LAST_USED_SWITCH`. Using SQL hints one can control, for example, whether a query shall be send to the MySQL replication master server or one of the slave servers. By help of SQL hints it is not possible to pick a certain slave server for query execution.

Full control on server selection can be gained using a callback function. Use of a callback is recommended to expert users only because the callback has to cover all cases otherwise handled by the plugin.

The plugin will invoke the callback function for selecting a server from the lists of configured master and slave servers. The callback function inspects the query to run and picks a server for query execution by returning the hosts URI, as found in the master and slave list.

If the lazy connections are enabled and the callback chooses a slave server for which no connection has been established so far and establishing the connection to the slave fails, the plugin will return an error upon the next action on the failed connection, for example, when running a query. It is the responsibility of the application developer to handle the error. For example, the application can re-run the query to trigger a new server selection and callback invocation. If so, the callback must make sure to select a different slave, or check slave availability, before returning to the plugin to prevent an endless loop.

Parameters

function	The function to be called. Class methods may also be invoked statically using this function by passing `array($classname, $methodname)` to this parameter. Additionally class methods of an object instance may be called by passing `array($objectinstance, $methodname)` to this parameter.

Return Values

Host to run the query on. The host URI is to be taken from the master and slave connection lists passed to the callback function. If callback returns a value neither found in the master nor in the slave connection lists the plugin will fallback to the second pick method configured via the `pick[]` setting in the plugin configuration file. If not second pick method is given, the plugin falls back to the build-in default pick method for server selection.

Notes

> **Note**
>
> `mysqlnd_ms_set_user_pick_server` is available with PECL mysqlnd_ms < 1.1.0. It has been replaced by the `user` filter. Please, check the Change History for upgrade notes.

Examples

Example 7.105 `mysqlnd_ms_set_user_pick_server` example

```
[myapp]
master[] = localhost
slave[] = 192.168.2.27:3306
slave[] = 192.168.78.136:3306
pick[] = user
```

```
<?php

function pick_server($connected, $query, $master, $slaves, $last_used)
{
 static $slave_idx = 0;
 static $num_slaves = NULL;
 if (is_null($num_slaves))
  $num_slaves = count($slaves);

 /* default: fallback to the plugins build-in logic */
 $ret = NULL;

 printf("User has connected to '%s'...\n", $connected);
 printf("... deciding where to run '%s'\n", $query);
```

```
 $where = mysqlnd_ms_query_is_select($query);
 switch ($where)
 {
  case MYSQLND_MS_QUERY_USE_MASTER:
   printf("... using master\n");
   $ret = $master[0];
   break;
  case MYSQLND_MS_QUERY_USE_SLAVE:
   /* SELECT or SQL hint for using slave */
   if (stristr($query, "FROM table_on_slave_a_only"))
   {
    /* a table which is only on the first configured slave  */
    printf("... access to table available only on slave A detected\n");
    $ret = $slaves[0];
   }
   else
   {
    /* round robin */
    printf("... some read-only query for a slave\n");
    $ret = $slaves[$slave_idx++ % $num_slaves];
   }
   break;
  case MYSQLND_MS_QUERY_LAST_USED:
   printf("... using last used server\n");
   $ret = $last_used;
   break;
 }

 printf("... ret = '%s'\n", $ret);
 return $ret;
}

mysqlnd_ms_set_user_pick_server("pick_server");

$mysqli = new mysqli("myapp", "root", "root", "test");

if (!($res = $mysqli->query("SELECT 1 FROM DUAL")))
 printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
else
 $res->close();

if (!($res = $mysqli->query("SELECT 2 FROM DUAL")))
 printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
else
 $res->close();


if (!($res = $mysqli->query("SELECT * FROM table_on_slave_a_only")))
 printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
else
 $res->close();

$mysqli->close();
?>
```

The above example will output:

```


User has connected to 'myapp'...
... deciding where to run 'SELECT 1 FROM DUAL'
... some read-only query for a slave
... ret = 'tcp://192.168.2.27:3306'
User has connected to 'myapp'...
```

```
... deciding where to run 'SELECT 2 FROM DUAL'
... some read-only query for a slave
... ret = 'tcp://192.168.78.136:3306'
User has connected to 'myapp'...
... deciding where to run 'SELECT * FROM table_on_slave_a_only'
... access to table available only on slave A detected
... ret = 'tcp://192.168.2.27:3306'
```

See Also

mysqlnd_ms_query_is_select
Filter concept
user filter

7.8.11 mysqlnd_ms_xa_begin

- mysqlnd_ms_xa_begin

  Starts a distributed/XA transaction among MySQL servers

Description

```
int mysqlnd_ms_xa_begin(
  mixed connection,
  string gtrid,
  int timeout);
```

Starts a XA transaction among MySQL servers. PECL/mysqlnd_ms acts as a transaction coordinator the distributed transaction.

Once a global transaction has been started, the plugin injects appropriate XA BEGIN SQL statements on all MySQL servers used in the following. The global transaction is either ended by calling mysqlnd_ms_xa_commit, mysqlnd_ms_xa_rollback or by an implicit rollback in case of an error.

During a global transaction, the plugin tracks all server switches, for example, when switching from one MySQL shard to another MySQL shard. Immediately before a query is run on a server that has not been participating in the global transaction yet, XA BEGIN is executed on the server. From a users perspective the injection happens during a call to a query execution function such as mysqli_query. Should the injection fail an error is reported to the caller of the query execution function. The failing server does not become a participant in the global transaction. The user may retry executing a query on the server and hereby retry injecting XA BEGIN, abort the global transaction because not all required servers can participate, or ignore and continue the global without the failed server.

Reasons to fail executing XA BEGIN include but are not limited to a server being unreachable or the server having an open, concurrent XA transaction using the same xid.

Please note, global and local transactions are mutually exclusive. You cannot start a XA transaction when you have a local transaction open. The local transaction must be ended first. The plugin tries to detect this conflict as early as possible. It monitors API calls for controlling local transactions to learn about the current state. However, if using SQL statements for local transactions such as BEGIN, the plugin may not know the current state and the conflict is not detected before XA BEGIN is injected and executed.

The use of other XA resources but MySQL servers is not supported by the function. To carry out a global transaction among, for example, a MySQL server and another vendors database system, you should issue the systems SQL commands yourself.

Experimental

The feature is currently under development. There may be issues and/or feature limitations. Do not use in production environments.

Parameters

connection	A MySQL connection handle obtained from any of the connect functions of the mysqli, mysql or PDO_MYSQL extensions.
gtrid	Global transaction identifier (gtrid). The gtrid is a binary string up to 64 bytes long. Please note, depending on your character set settings, 64 characters may require more than 64 bytes to store. In accordance with the MySQL SQL syntax, XA transactions use identifiers made of three parts. An xid consists of a global transaction identifier (gtrid), a branch qualifier (bqual) and a format identifier (formatID). Only the global transaction identifier can and needs to be set. The branch qualifier and format identifier are set automatically. The details should be considered implementation dependent, which may change without prior notice. In version 1.6 the branch qualifier is consecutive number which is incremented whenever a participant joins the global transaction.
timeout	Timeout in seconds. The default value is 60 seconds. The timeout is a hint to the garbage collection. If a transaction is recorded to take longer than expected, the garbage collection begins checking the transactions status. Setting a low value may make the garbage collection check the progress too often. Please note, checking the status of a global transaction may involve connecting to all recorded participants and possibly issuing queries on the servers.

Return Values

Returns TRUE if there is no open local or global transaction and a new global transaction can be started. Otherwise, returns FALSE

See Also

Quickstart XA/Distributed transactions
Runtime configuration
mysqlnd_ms_get_stats

7.8.12 mysqlnd_ms_xa_commit

- mysqlnd_ms_xa_commit

  Commits a distributed/XA transaction among MySQL servers

Description

```
int mysqlnd_ms_xa_commit(
  mixed connection,
  string gtrid);
```

Commits a global transaction among MySQL servers started by mysqlnd_ms_xa_begin.

If any of the global transaction participants fails to commit an implicit rollback is performed. It may happen that not all cases can be handled during the rollback. For example, no attempts will be made to reconnect to a participant after the connection to the participant has been lost. Solving cases that cannot easily be rolled back is left to the garbage collection.

> **Experimental**
>
> The feature is currently under development. There may be issues and/or feature limitations. Do not use in production environments.

Parameters

connection	A MySQL connection handle obtained from any of the connect functions of the mysqli, mysql or PDO_MYSQL extensions.
gtrid	Global transaction identifier (gtrid).

Return Values

Returns TRUE if the global transaction has been committed. Otherwise, returns FALSE

See Also

Quickstart XA/Distributed transactions
Runtime configuration
mysqlnd_ms_get_stats

7.8.13 mysqlnd_ms_xa_gc

- mysqlnd_ms_xa_gc

  Garbage collects unfinished XA transactions after severe errors

Description

```
int mysqlnd_ms_xa_gc(
  mixed connection,
  string gtrid,
  boolean ignore_max_retries);
```

Garbage collects unfinished XA transactions.

The XA protocol is a blocking protocol. There exist cases when servers participating in a global transaction cannot make progress when the transaction coordinator crashes or disconnects. In such a case, the MySQL servers keep waiting for instructions to finish the XA transaction in question. Because transactions occupy resources, transactions should always be terminated properly.

Garbage collection requires configuring a state store to track global transactions. Should a PHP client crash in the middle of a transaction and a new PHP client be started, then the built-in garbage collection

can learn about the aborted global transaction and terminate it. If you do not configure a state store, the garbage collection cannot perform any cleanup tasks.

The state store should be crash-safe and be highly available to survive its own crash. Currently, only MySQL is supported as a state store.

Garbage collection can also be performed automatically in the background. See the plugin configuration directive `garbage_collection` for details.

> **Experimental**
>
> The feature is currently under development. There may be issues and/or feature limitations. Do not use in production environments.

Parameters

`connection`	A MySQL connection handle obtained from any of the connect functions of the mysqli, mysql or PDO_MYSQL extensions.
`gtrid`	Global transaction identifier (gtrid). If given, the garbage collection considers the transaction only. Otherwise, the state store is scanned for any unfinished transaction.
`ignore_max_retries`	Whether to ignore the plugin configuration `max_retries` setting. If garbage collection continuously fails and the `max_retries` limit is reached prior to finishing the failed global transaction, you can attempt further runs prior to investigating the cause and solving the issue manually by issuing appropriate SQL statements on the participants. Setting the parameter has the same effect as temporarily setting `max_retries = 0`.

Return Values

Returns `TRUE` if garbage collection was successful. Otherwise, returns `FALSE`

See Also

Quickstart XA/Distributed transactions
Runtime configuration
State store configuration
`mysqlnd_ms_get_stats`

7.8.14 `mysqlnd_ms_xa_rollback`

- `mysqlnd_ms_xa_rollback`

  Rolls back a distributed/XA transaction among MySQL servers

Description

```
int mysqlnd_ms_xa_rollback(
  mixed connection,
  string gtrid);
```

Rolls back a global transaction among MySQL servers started by `mysqlnd_ms_xa_begin`.

If any of the global transaction participants fails to rollback the situation is left to be solved by the garbage collection.

> **Experimental**
>
> The feature is currently under development. There may be issues and/or feature limitations. Do not use in production environments.

Parameters

connection	A MySQL connection handle obtained from any of the connect functions of the mysqli, mysql or PDO_MYSQL extensions.
gtrid	Global transaction identifier (gtrid).

Return Values

Returns TRUE if the global transaction has been rolled back. Otherwise, returns FALSE

See Also

Quickstart XA/Distributed transactions
Runtime configuration
mysqlnd_ms_get_stats

7.9 Change History

This change history is a high level summary of selected changes that may impact applications and/or break backwards compatibility.

See also the CHANGES file in the source distribution for a complete list of changes.

7.9.1 PECL/mysqlnd_ms 1.6 series

1.6.0-alpha

- Release date: TBD
- Motto/theme: Maintenance and initial MySQL Fabric support

> **Note**
>
> This is the current development series. All features are at an early stage. Changes may happen at any time without prior notice. Please, do not use this version in production environments.
>
> The documentation may not reflect all changes yet.

Bug fixes

- Won't fix: #66616 R/W split fails: QOS with mysqlnd_get_last_gtid with built-in MySQL GTID

This is not a bug in the plugins implementation but a server side feature limitation not considered and documented before. MySQL 5.6 built-in GTIDs cannot be used to ensure session consistency when reading from slaves in all cases. In the worst case the plugin will not consider using the slaves and fallback to using the master. There will be no wrong results but no benefit from doing GTID checks either.

- Fixed #66064 - Random once load balancer ignoring weights

  Due to a config parsing bug random load balancing has ignored node weights if, and only if, the sticky flag was set (random once).

- Fixed #65496 - Wrong check for slave delay

  The quality of service filter has erroneously ignored slaves that lag for zero (0) seconds if a any maximum lag had been set. Although a slave was not lagging behind, it was excluded from the load balancing list if a maximum age was set by the QoS filter. This was due to using the wrong comparison operator in the source of the filter.

- Fixed #65408 - Compile failure with -Werror=format-security

Feature changes

- Introduced an internal connection pool. When using Fabric and switching from shard group A to shard group B, we are replacing the entire list of masters and slaves. This troubles the connections state alignment logic and some filters. Some filters cache information on the master and slave lists. The new internal connection pool abstraction allows us to inform the filters of changes, hence they can update their caches.

  Later on, the pool can also be used to reduce connection overhead. Assume you are switching from a shard group to another and back again. Whenever the switch is done, the pool's active server (and connection) lists are replaced. However, no longer used connections are not necessarily closed immediately but can be kept in the pool for later reuse.

  Please note, the connection pool is internalat this point. There are some new statistics to monitor it. However, you cannot yet configure pool size of behaviour.

- Added a basic distributed transaction abstraction. XA transactions can are supported ever since using standard SQL calls. This is inconvenient as XA participants must be managed manually. PECL/mysqlnd_ms introduces API calls to control XA transaction among MySQL servers. When using the new functions, PECL/mysqlnd_ms acts as a transaction coordinator. After starting a distributed transaction, the plugin tracks all servers involved until the transaction is ended and issues appropriate SQL statements on the XA participants.

  This is useful, for example, when using Fabric and sharding. When using Fabric the actual shard servers involved in a business transaction may not be known in advance. Thus, manually controlling a transaction that spawns multiple shards becomes difficult. Please, be warned about current limitations.

- Introduced automatic retry loop for transient errors and corresponding statistic to count the number of implicit retries. Some distributed database clusters use transient errors to hint a client to retry its operation in a bit. Most often, the client is then supposed to halt execution (sleep) for a short moment before retrying the desired operation. Immediately failing over to another node is not necessary in response to the error. Instead, a retry loop can be performed. Common situation when using MySQL Cluster.

- Introduced automatic retry loop for transient errors and corresponding statistic to count the number of implicit retries. Some distributed database clusters use transient errors to hint a client to retry its

operation in a bit. Most often, the client is then supposed to halt execution (sleep) for a short moment before retrying the desired operation. Immediately failing over to another node is not necessary in response to the error. Instead, a retry loop can be performed. Common situation when using MySQL Cluster.

- Introduced most basic support for the MySQL Fabric High Availability and sharding framework.

  Please, consider this pre-alpha quality. Both the server side framework and the client side code is supposed to work flawless considering the MySQL Fabric quickstart examples only. However, testing has not been performed to the level of prior plugin alpha releases. Either sides are moving targets, API changes may happen at any time without prior warning.

  As this is work in progress, the manual may not yet reflect allow feature limitations and known bugs.

- New statistics to monitor the Fabric XML RPC call `sharding.lookup_servers`: `fabric_sharding_lookup_servers_success`, `fabric_sharding_lookup_servers_failure`, `fabric_sharding_lookup_servers_time_total`, `fabric_sharding_lookup_servers_bytes_total`, `fabric_sharding_lookup_servers_xml_failure`.

- New functions related to MySQL Fabric: `mysqlnd_ms_fabric_select_shard`, `mysqlnd_ms_fabric_select_global`, `mysqlnd_ms_dump_servers`.

7.9.2 PECL/mysqlnd_ms 1.5 series

1.5.1-stable

- Release date: 06/2013
- Motto/theme: Sharding support, improved transaction support

> **Note**
>
> This is the current stable series. Use this version in production environments.
>
> The documentation is complete.

1.5.0-alpha

- Release date: 03/2013
- Motto/theme: Sharding support, improved transaction support

Bug fixes

- Fixed #60605 PHP segmentation fault when mysqlnd_ms is enabled.
- Setting transaction stickiness disables all load balancing, including automatic failover, for the duration of a transaction. So far connection switches could have happened in the middle of a transaction in multi-master configurations and during automatic failover although transaction monitoring had detected transaction boundaries properly.
- BC break and bug fix. SQL hints enforcing the use of a specific kind of server (`MYSQLND_MS_MASTER_SWITCH`, `MYSQLND_MS_SLAVE_SWITCH`, `MYSQLND_MS_LAST_USED_SWITCH`) are ignored for the duration of a transaction of transaction stickiness is enabled and transaction boundaries have been detected properly.

This is a change in behaviour. However, it is also a bug fix and a step to align behaviour. If, in previous versions, transaction stickiness, one of the above listed SQL hints and the quality of service filtering was combined it could happened that the SQL hints got ignored. In some case the SQL hints did work, in other cases they did not. The new behaviour is more consistent. SQL hints will always be ignore for the duration of a transaction, if transaction stickiness is enabled.

Please note, transaction boundary detection continues to be based on API call monitoring. SQL commands controlling transactions are not monitored.

- BC break and bug fix. Calls to `mysqlnd_ms_set_qos` will fail when done in the middle of a transaction if transaction stickiness is enabled. Connection switches are not allowed for the duration of a transaction. Changing the quality of service likely results on a different set of servers qualifying for query execution, possibly making it necessary to switch connections. Thus, the call is not allowed in during an active transaction. The quality of server can, however, be changed in between transactions.

Feature changes

- Introduced the `node_group` filter. The filter lets you organize servers (master and slaves) into groups. Queries can be directed to a certain group of servers by prefixing the query statement with a SQL hint/comment that contains the groups configured name. Grouping can be used for partitioning and sharding, and also to optimize for local caching. In the case of sharding, a group name can be thought of like a shard key. All queries for a given shard key will be executed on the configured shard. Note: both the client and server must support sharding for sharding to function with mysqlnd_ms.

- Extended configuration file validation during PHP startup (RINIT). An `E_WARNING` level error will be thrown if the configuration file can not be read (permissions), is empty, or the file (JSON) could not be parsed. Warnings may appear in log files, which depending on how PHP is configured.

  Distributions that aim to provide a pre-configured setup, including a configuration file stub, are asked to put `{}` into the configuration file to prevent this warning about an invalid configuration file.

  Further configuration file validation is done when parsing sections upon opening a connection. Please, note that there may still be situations when an invalid plugin configuration file does not lead to proper error messages but a failure to connect.

- As of PHP 5.5.0, improved support for transaction boundaries detection was added for `mysqli`. The `mysqli` extension has been modified to use the new C API calls of the `mysqlnd` library to begin, commit, and rollback a transaction or savepoint. If trx_stickiness is used to enable transaction aware load balancing, the `mysqli_begin`, `mysqli_commit` and `mysqli_rollback` functions will now be monitered by the plugin, to go along with the `mysqli_autocommit` function that was already supported. All SQL features to control transactions are also available through the improved `mysqli` transaction control related functions. This means that it is not required to issue SQL statements instead of using API calls. Applications using the appropriate API calls can be load balanced by PECL/mysqlnd_ms in a completely transaction-aware way.

  Please note, `PDO_MySQL` has not been updated yet to utilize the new mysqlnd API calls. Thus, transaction boundary detection with `PDO_MySQL` continues to be limited to the monitoring by passing in `PDO::ATTR_AUTOCOMMIT` to `PDO::setAttribute`.

- Introduced `trx_stickiness=on`. This trx_stickiness option differs from `trx_stickiness=master` as it tries to execute a read-only transaction on a slave, if quality of service (consistency level) allows the use of a slave. Read-only transactions were introduced in MySQL 5.6, and they offer performance gains.

- Query cache support is considered beta if used with the `mysqli` API. It should work fine with primary copy based clusters. For all other APIs, this feature continues to be called experimental.

- The code examples in the mysqlnd_ms source were updated.

7.9.3 PECL/mysqlnd_ms 1.4 series

1.4.2-stable

- Release date: 08/2012
- Motto/theme: Tweaking based on user feedback

1.4.1-beta

- Release date: 08/2012
- Motto/theme: Tweaking based on user feedback

Bug fixes

- Fixed build with PHP 5.5

1.4.0-alpha

- Release date: 07/2012
- Motto/theme: Tweaking based on user feedback

Feature changes

- BC break: Renamed plugin configuration setting `ini_file` to `config_file`. In early versions the plugin configuration file used ini style. Back then the configuration setting was named accordingly. It has now been renamed to reflect the newer file format and to distinguish it from PHP's own ini file (configuration directives file).
- Introduced new default charset setting `server_charset` to allow proper escaping before a connection is opened. This is most useful when using lazy connections, which are a default.
- Introduced `wait_for_gtid_timeout` setting to throttle slave reads that need session consistency. If global transaction identifier are used and the service level is set to session consistency, the plugin tries to find up-to-date slaves. The slave status check is done by a SQL statement. If nothing else is set, the slave status is checked only one can the search for more up-to-date slaves continues immediately thereafter. Setting `wait_for_gtid_timeout` instructs the plugin to poll a slaves status for `wait_for_gtid_timeout` seconds if the first execution of the SQL statement has shown that the slave is not up-to-date yet. The poll will be done once per second. This way, the plugin will wait for slaves to catch up and throttle the client.
- New failover strategy `loop_before_master`. By default the plugin does no failover. It is possible to enable automatic failover if a connection attempt fails. Upto version 1.3 only `master` strategy existed to failover to a master if a slave connection fails. `loop_before_master` is similar but tries all other slaves before attempting to connect to the master if a slave connection fails.

  The number of attempts can be limited using the `max_retries` option. Failed hosts can be remembered and skipped in load balancing for the rest of the web request. `max_retries` and `remember_failed` are considered experimental although decent stability is given. Syntax and semantics may change in the future without prior notice.

7.9.4 PECL/mysqlnd_ms 1.3 series

1.3.2-stable

- Release date: 04/2012
- Motto/theme: see 1.3.0-alpha

Bug fixes

- Fixed problem with multi-master where although in a transaction the queries to the master weren't sticky and were spread all over the masters (RR). Still not sticky for Random. Random_once is not affected.

1.3.1-beta

- Release date: 04/2012
- Motto/theme: see 1.3.0-alpha

Bug fixes

- Fixed problem with building together with QC.

1.3.0-alpha

- Release date: 04/2012
- Motto/theme: Query caching through quality-of-service concept

The 1.3 series aims to improve the performance of applications and the overall load of an asynchronous MySQL cluster, for example, a MySQL cluster using MySQL Replication. This is done by transparently replacing a slave access with a local cache access, if the application allows it by setting an appropriate quality of service flag. When using MySQL replication a slave can serve stale data. An application using MySQL replication must continue to work correctly with stale data. Given that the application is know to work correctly with stale data, the slave access can transparently be replace with a local cache access.

PECL/mysqlnd_qc serves as a cache backend. PECL/mysqlnd_qc supports use of various storage locations, among others main memory, `APC` and `MEMCACHE`.

Feature changes

- Added cache option to quality-of-service (QoS) filter.
  - New configure option `enable-mysqlnd-ms-cache-support`
  - New constant `MYSQLND_MS_HAVE_CACHE_SUPPORT`.
  - New constant `MYSQLND_MS_QOS_OPTION_CACHE` to be used with `mysqlnd_ms_set_qos`.
- Support for built-in global transaction identifier feature of MySQL 5.6.5-m8 or newer.

7.9.5 PECL/mysqlnd_ms 1.2 series

1.2.1-beta

- Release date: 01/2012
- Motto/theme: see 1.2.0-alpha

Minor test changes.

1.2.0-alpha

- Release date: 11/2011
- Motto/theme: Global Transaction ID injection and quality-of-service concept

In version 1.2 the focus continues to be on supporting MySQL database clusters with asynchronous replication. The plugin tries to make using the cluster introducing a quality-of-service filter which applications can use to define what service quality they need from the cluster. Service levels provided are eventual consistency with optional maximum age/slave slag, session consistency and strong consistency.

Additionally the plugin can do client-side global transaction id injection to make manual master failover easier.

Feature changes

- Introduced quality-of-service (QoS) filter. Service levels provided by QoS filter:
  - eventual consistency, optional option slave lag
  - session consistency, optional option GTID
  - strong consistency
- Added the `mysqlnd_ms_set_qos` function to set the required connection quality at runtime. The new constants related to `mysqlnd_ms_set_qos` are:
  - `MYSQLND_MS_QOS_CONSISTENCY_STRONG`
  - `MYSQLND_MS_QOS_CONSISTENCY_SESSION`
  - `MYSQLND_MS_QOS_CONSISTENCY_EVENTUAL`
  - `MYSQLND_MS_QOS_OPTION_GTID`
  - `MYSQLND_MS_QOS_OPTION_AGE`
- Added client-side global transaction id injection (GTID).
- New statistics related to GTID:
  - `gtid_autocommit_injections_success`
  - `gtid_autocommit_injections_failure`
  - `gtid_commit_injections_success`
  - `gtid_commit_injections_failure`
  - `gtid_implicit_commit_injections_success`
  - `gtid_implicit_commit_injections_failure`

- Added `mysqlnd_ms_get_last_gtid` to fetch the last global transaction id.
- Enabled support for multi master without slaves.

7.9.6 PECL/mysqlnd_ms 1.1 series

1.1.0

- Release date: 09/2011
- Motto/theme: Cover replication basics with production quality

The 1.1 and 1.0 series expose a similar feature set. Internally, the 1.1 series has been refactored to plan for future feature additions. A new configuration file format has been introduced, and limitations have been lifted. And the code quality and quality assurance has been improved.

Feature changes

- Added the (chainable) filter concept:
  - BC break: `mysqlnd_ms_set_user_pick_server` has been removed. Thehttp://svn.php.net/viewvc/pecl/mysqlnd_ms/trunk/ `user` filter has been introduced to replace it. The filter offers similar functionality, but see below for an explanation of the differences.
- New powerful JSON based configuration syntax.
- Lazy connections improved: security relevant, and state changing commands are covered.
- Support for (native) prepared statements.
- New statistics: `use_master_guess`, `use_slave_guess`.
  - BC break: Semantics of statistics changed for `use_slave`, `use_master`. Future changes are likely. Please see, `mysqlnd_ms_get_stats`.
- List of broadcasted messages extended by `ssl_set`.
- Library calls now monitored to remember settings for lazy connections: `change_user`, `select_db`, `set_charset`, `set_autocommit`.
- Introduced `mysqlnd_ms.disable_rw_split`. The configuration setting allows using the load balancing and lazy connection functionality independently of read write splitting.

Bug fixes

- Fixed PECL #22724 - Server switching (mysqlnd_ms_query_is_select() case sensitive)
- Fixed PECL #22784 - Using mysql_connect and mysql_select_db did not work
- Fixed PECL #59982 - Unusable extension with --enable-mysqlnd-ms-table-filter. Use of the option is NOT supported. You must not used it. Added note to m4.
- Fixed Bug #60119 - host="localhost" lost in mysqlnd_ms_get_last_used_connection()

The `mysqlnd_ms_set_user_pick_server` function was removed, and replaced in favor of a new `user` filter. You can no longer set a callback function using `mysqlnd_ms_set_user_pick_server` at

runtime, but instead have to configure it in the plugins configuration file. The `user` filter will pass the same arguments to the callback as before. Therefore, you can continue to use the same procedural function as a callback.callback It is no longer possible to use static class methods, or class methods of an object instance, as a callback. Doing so will cause the function executing a statement handled by the plugin to emit an `E_RECOVERABLE_ERROR` level error, which might look like: "`(mysqlnd_ms) Specified callback (picker) is not a valid callback`." Note: this may halt your application.

7.9.7 PECL/mysqlnd_ms 1.0 series

1.0.1-alpha

- Release date: 04/2011
- Motto/theme: bug fix release

1.0.0-alpha

- Release date: 04/2011
- Motto/theme: Cover replication basics to test user feedback

The first release of practical use. It features basic automatic read-write splitting, SQL hints to overrule automatic redirection, load balancing of slave requests, lazy connections, and optional, automatic use of the master after the first write.

The public feature set is close to that of the 1.1 release.

1.0.0-pre-alpha

- Release date: 09/2010
- Motto/theme: Proof of concept

Initial check-in. Essentially a demo of the mysqlnd plugin API.

Chapter 8 Mysqlnd query result cache plugin

Table of Contents

The mysqlnd query result cache plugin adds easy to use client-side query caching to all PHP MySQL extensions using mysqlnd.

As of version PHP 5.3.3 the MySQL native driver for PHP (`mysqlnd`) features an internal plugin C API. C plugins, such as the query cache plugin, can extend the functionality of mysqlnd.

Mysqlnd plugins such as the query cache plugin operate transparent from a user perspective. The cache plugin supports all PHP applications and all PHP MySQL extensions (mysqli, mysql, PDO_MYSQL). It does not change existing APIs.

No significant application changes are required to cache a query. The cache has two operation modes. It will either cache all queries (not recommended) or only those queries marked with a certain SQL hint (recommended).

8.1 Key Features

- Transparent and therefore easy to use
  - supports all PHP MySQL extensions
  - no API changes
  - very little application changes required
- Flexible invalidation strategy
  - Time-to-Live (TTL)
  - user-defined
- Storage with different scope and life-span
  - Default (Hash, process memory)
  - APC
  - MEMCACHE
  - sqlite
  - user-defined
- Built-in slam defense to prevent cache stampeding.

8.2 Limitations

The current 1.0.1 release of PECL mysqlnd_qc does not support PHP 5.4. Version 1.1.0-alpha lifts this limitation.

Prepared statements and unbuffered queries are fully supported. Thus, the plugin is capable of caching all statements issued with `mysqli` or `PDO_MySQL`, which are the only two PHP MySQL APIs to offer prepared statement support.

8.3 On the name

The shortcut `mysqlnd_qc` stands for `mysqlnd query cache plugin`. The name was chosen for a quick-and-dirty proof-of-concept. In the beginning the developers did not expect to continue using the code base. Sometimes PECL/mysqlnd_qc has also been called `client-side query result set cache`.

8.4 Quickstart and Examples

The mysqlnd query cache plugin is easy to use. This quickstart will demo typical use-cases, and provide practical advice on getting started.

It is strongly recommended to read the reference sections in addition to the quickstart. It is safe to begin with the quickstart. However, before using the plugin in mission critical environments we urge you to read additionally the background information from the reference sections.

Most of the examples use the mysqli extension because it is the most feature complete PHP MySQL extension. However, the plugin can be used with any PHP MySQL extension that is using the mysqlnd library.

8.4.1 Architecture and Concepts

The query cache plugin is implemented as a PHP extension. It is written in C and operates under the hood of PHP. During the startup of the PHP interpreter, it gets registered as a mysqlnd plugin to replace selected mysqlnd C methods. Hereby, it can change the behaviour of any PHP MySQL extension (mysqli, PDO_MYSQL, mysql) compiled to use the mysqlnd library without changing the extensions API. This makes the plugin compatible with each and every PHP MySQL application. Because existing APIs are not changed, it is almost transparent to use. Please, see the mysqlnd plugin API description for a discussion of the advantages of the plugin architecture and a comparison with proxy based solutions.

Transparent to use

At PHP run time PECL/mysqlnd_qc can proxy queries send from PHP (mysqlnd) to the MySQL server. It then inspects the statement string to find whether it shall cache its results. If so, result set is cached using a storage handler and further executions of the statement are served from the cache for a user-defined period. The Time to Live (TTL) of the cache entry can either be set globally or on a per statement basis.

A statement is either cached if the plugin is instructed to cache all statements globally using a or, if the query string starts with the SQL hint (`/*qc=on*/`). The plugin is capable of caching any query issued by calling appropriate API calls of any of the existing PHP MySQL extensions.

Flexible storage: various storage handler

Various storage handler are supported to offer different scopes for cache entries. Different scopes allow for different degrees in sharing cache entries among clients.

- `default` (built-in): process memory, scope: process, one or more web requests depending on PHP deployment model used
- `APC`: shared memory, scope: single server, multiple web requests
- `SQLite`: memory or file, scope: single server, multiple web requests
- `MEMCACHE`: main memory, scope: single or multiple server, multiple web requests
- `user` (built-in): user-defined - any, scope: user-defined - any

Support for the `APC`, `SQLite` and `MEMCACHE` storage handler has to be enabled at compile time. The `default` and `user` handler are built-in. It is possible to switch between compiled-in storage handlers on a per query basis at run time. However, it is recommended to pick one storage handler and use it for all cache entries.

Built-in slam defense to avoid overloading

To avoid overload situations the cache plugin has a built-in slam defense mechanism. If a popular cache entries expires many clients using the cache entries will try to refresh the cache entry. For the duration of the refresh many clients may access the database server concurrently. In the worst case, the database server becomes overloaded and it takes more and more time to refresh the cache entry, which in turn lets more and more clients try to refresh the cache entry. To prevent this from happening the plugin has a slam

defense mechanism. If slam defense is enabled and the plugin detects an expired cache entry it extends the life time of the cache entry before it refreshes the cache entry. This way other concurrent accesses to the expired cache entry are still served from the cache for a certain time. The other concurrent accesses to not trigger a concurrent refresh. Ideally, the cache entry gets refreshed by the client which extended the cache entries lifespan before other clients try to refresh the cache and potentially cause an overload situation.

Unique approach to caching

PECL/mysqlnd_qc has a unique approach to caching result sets that is superior to application based cache solutions. Application based solutions first fetch a result set into PHP variables. Then, the PHP variables are serialized for storage in a persistent cache, and then unserialized when fetching. The mysqlnd query cache stores the raw wire protocol data sent from MySQL to PHP in its cache and replays it, if still valid, on a cache hit. This way, it saves an extra serialization step for a cache put that all application based solutions have to do. It can store the raw wire protocol data in the cache without having to serialize into a PHP variable first and deserializing the PHP variable for storing in the cache again.

8.4.2 Setup

The plugin is implemented as a PHP extension. See also the installation instructions to install the PECL/mysqlnd_qc extension.

Compile or configure the PHP MySQL extension (mysqli, PDO_MYSQL, mysql) that you plan to use with support for the mysqlnd library. PECL/mysqlnd_qc is a plugin for the mysqlnd library. To use the plugin with any of the existing PHP MySQL extensions (APIs), the extension has to use the mysqlnd library.

Then, load the extension into PHP and activate the plugin in the PHP configuration file using the PHP configuration directive named mysqlnd_qc.enable_qc.

Example 8.1 Enabling the plugin (php.ini)

```
mysqlnd_qc.enable_qc=1
```

8.4.3 Caching queries

There are four ways to trigger caching of a query.

- Use of SQL hints on a per query basis
- User supplied callbacks to decide on a per query basis, for example, using `mysqlnd_qc_is_select`
- `mysqlnd_set_cache_condition` for rule based automatic per query decisions
- `mysqlnd_qc.cache_by_default = 1` to cache all queries blindly

Use of SQL hints and `mysqlnd_qc.cache_by_default = 1` are explained below. Please, refer to the function reference on `mysqlnd_qc_is_select` for a description of using a callback and, `mysqlnd_qc_set_cache_condition` on how to set rules for automatic caching.

A SQL hint is a SQL standards compliant comment. As a SQL comment it is ignored by the database. A statement is considered eligible for caching if it either begins with the SQL hint enabling caching or it is a `SELECT` statement.

An individual query which shall be cached must begin with the SQL hint `/*qc=on*/`. It is recommended to use the PHP constant `MYSQLND_QC_ENABLE_SWITCH` instead of using the string value.

- not eligible for caching and not cached: `INSERT INTO test(id) VALUES (1)`
- not eligible for caching and not cached: `SHOW ENGINES`
- eligible for caching but uncached: `SELECT id FROM test`
- eligible for caching and cached: `/*qc=on*/SELECT id FROM test`

The examples `SELECT` statement string is prefixed with the `MYSQLND_QC_ENABLE_SWITCH` SQL hint to enable caching of the statement. The SQL hint must be given at the very beginning of the statement string to enable caching.

Example 8.2 Using the `MYSQLND_QC_ENABLE_SWITCH` SQL hint

```
mysqlnd_qc.enable_qc=1
```

```
<?php
/* Connect, create and populate test table */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2)");

/* Will be cached because of the SQL hint */
$start = microtime(true);
$res   = $mysqli->query("/*" . MYSQLND_QC_ENABLE_SWITCH . "*/" . "SELECT id FROM test WHERE id = 1");

var_dump($res->fetch_assoc());
$res->free();

printf("Total time uncached query: %.6fs\n", microtime(true) - $start);

/* Cache hit */
$start = microtime(true);
$res   = $mysqli->query("/*" . MYSQLND_QC_ENABLE_SWITCH . "*/" . "SELECT id FROM test WHERE id = 1");

var_dump($res->fetch_assoc());
$res->free();

printf("Total time cached query: %.6fs\n", microtime(true) - $start);
?>
```

The above examples will output something similar to:

```
array(1) {
  ["id"]=>
  string(1) "1"
}
Total time uncached query: 0.000740s
array(1) {
  ["id"]=>
  string(1) "1"
}
```

```
Total time cached query: 0.000098s
```

If nothing else is configured, as it is the case in the quickstart example, the plugin will use the built-in `default` storage handler. The `default` storage handler uses process memory to hold a cache entry. Depending on the PHP deployment model, a PHP process may serve one or more web requests. Please, consult the web server manual for details. Details make no difference for the examples given in the quickstart.

The query cache plugin will cache all queries regardless if the query string begins with the SQL hint which enables caching or not, if the PHP configuration directive `mysqlnd_qc.cache_by_default` is set to `1`. The setting `mysqlnd_qc.cache_by_default` is evaluated by the core of the query cache plugins. Neither the built-in nor user-defined storage handler can overrule the setting.

The SQL hint `/*qc=off*/` can be used to disable caching of individual queries if `mysqlnd_qc.cache_by_default = 1` It is recommended to use the PHP constant `MYSQLND_QC_DISABLE_SWITCH` instead of using the string value.

Example 8.3 Using the `MYSQLND_QC_DISABLE_SWITCH` SQL hint

```
mysqlnd_qc.enable_qc=1
mysqlnd_qc.cache_by_default=1
```

```
<?php
/* Connect, create and populate test table */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2)");

/* Will be cached although no SQL hint is present because of mysqlnd_qc.cache_by_default = 1*/
$res = $mysqli->query("SELECT id FROM test WHERE id = 1");
var_dump($res->fetch_assoc());
$res->free();

$mysqli->query("DELETE FROM test WHERE id = 1");

/* Cache hit - no automatic invalidation and still valid! */
$res = $mysqli->query("SELECT id FROM test WHERE id = 1");
var_dump($res->fetch_assoc());
$res->free();

/* Cache miss - query must not be cached because of the SQL hint */
$res = $mysqli->query("/*" . MYSQLND_QC_DISABLE_SWITCH . "*/SELECT id FROM test WHERE id = 1");
var_dump($res->fetch_assoc());
$res->free();
?>
```

The above examples will output:

```
array(1) {
  ["id"]=>
  string(1) "1"
}
```

```
array(1) {
  ["id"]=>
  string(1) "1"
}
NULL
```

PECL/mysqlnd_qc forbids caching of statements for which at least one column from the statements result set shows no table name in its meta data by default. This is usually the case for columns originating from SQL functions such as `NOW()` or `LAST_INSERT_ID()`. The policy aims to prevent pitfalls if caching by default is used.

Example 8.4 Example showing which type of statements are not cached

```
mysqlnd_qc.enable_qc=1
mysqlnd_qc.cache_by_default=1
```

```
<?php
/* Connect, create and populate test table */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1)");

for ($i = 0; $i < 3; $i++) {

    $start = microtime(true);

    /* Note: statement will not be cached because of NOW() use */
    $res = $mysqli->query("SELECT id, NOW() AS _time FROM test");
    $row = $res->fetch_assoc();

    /* dump results */
    var_dump($row);

    printf("Total time: %.6fs\n", microtime(true) - $start);

    /* pause one second */
    sleep(1);
}
?>
```

The above examples will output something similar to:

```
array(2) {
  ["id"]=>
  string(1) "1"
  ["_time"]=>
  string(19) "2012-01-11 15:43:10"
}
Total time: 0.000540s
array(2) {
  ["id"]=>
  string(1) "1"
  ["_time"]=>
```

```
  string(19) "2012-01-11 15:43:11"
}
Total time: 0.000555s
array(2) {
  ["id"]=>
  string(1) "1"
  ["_time"]=>
  string(19) "2012-01-11 15:43:12"
}
Total time: 0.000549s
```

It is possible to enable caching for all statements including those which has columns in their result set for which MySQL reports no table, such as the statement from the example. Set `mysqlnd_qc.cache_no_table = 1` to enable caching of such statements. Please, note the difference in the measured times for the above and below examples.

Example 8.5 Enabling caching for all statements using the `mysqlnd_qc.cache_no_table` ini setting

```
mysqlnd_qc.enable_qc=1
mysqlnd_qc.cache_by_default=1
mysqlnd_qc.cache_no_table=1
```

```
<?php
/* Connect, create and populate test table */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1)");

for ($i = 0; $i < 3; $i++) {

    $start = microtime(true);

    /* Note: statement will not be cached because of NOW() use */
    $res = $mysqli->query("SELECT id, NOW() AS _time FROM test");
    $row = $res->fetch_assoc();

    /* dump results */
    var_dump($row);

    printf("Total time: %.6fs\n", microtime(true) - $start);

    /* pause one second */
    sleep(1);
}
?>
```

The above examples will output something similar to:

```
array(2) {
  ["id"]=>
  string(1) "1"
  ["_time"]=>
```

```
  string(19) "2012-01-11 15:47:45"
}
Total time: 0.000546s
array(2) {
  ["id"]=>
  string(1) "1"
  ["_time"]=>
  string(19) "2012-01-11 15:47:45"
}
Total time: 0.000187s
array(2) {
  ["id"]=>
  string(1) "1"
  ["_time"]=>
  string(19) "2012-01-11 15:47:45"
}
Total time: 0.000167s
```

> **Note**
>
> Although `mysqlnd_qc.cache_no_table = 1` has been created for use with `mysqlnd_qc.cache_by_default = 1` it is bound it. The plugin will evaluate the `mysqlnd_qc.cache_no_table` whenever a query is to be cached, no matter whether caching has been enabled using a SQL hint or any other measure.

8.4.4 Setting the TTL

The default invalidation strategy of the query cache plugin is Time to Live (`TTL`). The built-in storage handlers will use the default `TTL` defined by the PHP configuration value `mysqlnd_qc.ttl` unless the query string contains a hint for setting a different `TTL`. The `TTL` is specified in seconds. By default cache entries expire after `30` seconds

The example sets `mysqlnd_qc.ttl=3` to cache statements for three seconds by default. Every second it updates a database table record to hold the current time and executes a `SELECT` statement to fetch the record from the database. The `SELECT` statement is cached for three seconds because it is prefixed with the SQL hint enabling caching. The output verifies that the query results are taken from the cache for the duration of three seconds before they are refreshed.

Example 8.6 Setting the TTL with the `mysqlnd_qc.ttl` ini setting

```
mysqlnd_qc.enable_qc=1
mysqlnd_qc.ttl=3
```

```
<?php
/* Connect, create and populate test table */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id VARCHAR(255))");

for ($i = 0; $i < 7; $i++) {

    /* update DB row  */
    if (!$mysqli->query("DELETE FROM test") ||
```

```
        !$mysqli->query("INSERT INTO test(id) VALUES (NOW())"))
      /* Of course, a real-life script should do better error handling */
      die(sprintf("[%d] %s\n", $mysqli->errno, $mysqli->error));

    /* select latest row but cache results */
    $query  = "/*" . MYSQLND_QC_ENABLE_SWITCH . "*/";
    $query .= "SELECT id AS _time FROM test";
    if (!($res = $mysqli->query($query)) ||
        !($row = $res->fetch_assoc()))
    {
      printf("[%d] %s\n", $mysqli->errno, $mysqli->error);
    }
    $res->free();
    printf("Wall time %s - DB row time %s\n", date("H:i:s"), $row['_time']);

    /* pause one second */
    sleep(1);
}
?>
```

The above examples will output something similar to:

```
Wall time 14:55:59 - DB row time 2012-01-11 14:55:59
Wall time 14:56:00 - DB row time 2012-01-11 14:55:59
Wall time 14:56:01 - DB row time 2012-01-11 14:55:59
Wall time 14:56:02 - DB row time 2012-01-11 14:56:02
Wall time 14:56:03 - DB row time 2012-01-11 14:56:02
Wall time 14:56:04 - DB row time 2012-01-11 14:56:02
Wall time 14:56:05 - DB row time 2012-01-11 14:56:05
```

As can be seen from the example, any `TTL` based cache can serve stale data. Cache entries are not automatically invalidated, if underlying data changes. Applications using the default `TTL` invalidation strategy must be able to work correctly with stale data.

A user-defined cache storage handler can implement any invalidation strategy to work around this limitation.

The default `TTL` can be overruled using the SQL hint `/*qc_tt=seconds*/`. The SQL hint must be appear immediately after the SQL hint which enables caching. It is recommended to use the PHP constant `MYSQLND_QC_TTL_SWITCH` instead of using the string value.

Example 8.7 Setting TTL with SQL hints

```
<?php
$start = microtime(true);

/* Connect, create and populate test table */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2)");

printf("Default TTL\t: %d seconds\n", ini_get("mysqlnd_qc.ttl"));

/* Will be cached for 2 seconds */
$sql = sprintf("/*%s*//*%s%d*/SELECT id FROM test WHERE id = 1", MYSQLND_QC_ENABLE_SWITCH, MYSQLND_QC_TTL_SWIT
$res = $mysqli->query($sql);
```

```
var_dump($res->fetch_assoc());
$res->free();

$mysqli->query("DELETE FROM test WHERE id = 1");
sleep(1);

/* Cache hit - no automatic invalidation and still valid! */
$res = $mysqli->query($sql);
var_dump($res->fetch_assoc());
$res->free();

sleep(2);

/* Cache miss - cache entry has expired */
$res = $mysqli->query($sql);
var_dump($res->fetch_assoc());
$res->free();

printf("Script runtime\t: %d seconds\n", microtime(true) - $start);
?>
```

The above examples will output something similar to:

```
Default TTL     : 30 seconds
array(1) {
  ["id"]=>
  string(1) "1"
}
array(1) {
  ["id"]=>
  string(1) "1"
}
NULL
Script runtime  : 3 seconds
```

8.4.5 Pattern based caching

An application has three options for telling PECL/mysqlnd_qc whether a particular statement shall be used. The most basic approach is to cache all statements by setting `mysqlnd_qc.cache_by_default = 1`. This approach is often of little practical value. But it enables users to make a quick estimation about the maximum performance gains from caching. An application designed to use a cache may be able to prefix selected statements with the appropriate SQL hints. However, altering an applications source code may not always be possible or desired, for example, to avoid problems with software updates. Therefore, PECL/mysqlnd_qc allows setting a callback which decides if a query is to be cached.

The callback is installed with the `mysqlnd_qc_set_is_select` function. The callback is given the statement string of every statement inspected by the plugin. Then, the callback can decide whether to cache the function. The callback is supposed to return `FALSE` if the statement shall not be cached. A return value of `TRUE` makes the plugin try to add the statement into the cache. The cache entry will be given the default TTL (`mysqlnd_qc.ttl`). If the callback returns a numerical value it is used as the TTL instead of the global default.

Example 8.8 Setting a callback with `mysqlnd_qc_set_is_select`

```
mysqlnd_qc.enable_qc=1
mysqlnd_qc.collect_statistics=1
```

```
<?php
/* callback which decides if query is cached */
function is_select($query) {
    static $patterns = array(
      /* true - use default from mysqlnd_qc.ttl */
      "@SELECT\s+.*\s+FROM\s+test@ismU" => true,
      /* 3 - use TTL = 3 seconds */
      "@SELECT\s+.*\s+FROM\s+news@ismU" => 3
    );

    /* check if query does match pattern */
    foreach ($patterns as $pattern => $ttl) {
        if (preg_match($pattern, $query)) {
            printf("is_select(%45s): cache\n", $query);
            return $ttl;
        }
    }
    printf("is_select(%45s): do not cache\n", $query);
    return false;
}
/* install callback */
mysqlnd_qc_set_is_select("is_select");

/* Connect, create and populate test table */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)");

/* cache put */
$mysqli->query("SELECT id FROM test WHERE id = 1");
/* cache hit */
$mysqli->query("SELECT id FROM test WHERE id = 1");
/* cache put */
$mysqli->query("SELECT * FROM test");

$stats = mysqlnd_qc_get_core_stats();
printf("Cache put: %d\n", $stats['cache_put']);
printf("Cache hit: %d\n", $stats['cache_hit']);
?>
```

The above examples will output something similar to:

```
is_select(                   DROP TABLE IF EXISTS test): do not cache
is_select(                   CREATE TABLE test(id INT)): do not cache
is_select(    INSERT INTO test(id) VALUES (1), (2), (3)): do not cache
is_select(             SELECT id FROM test WHERE id = 1): cache
is_select(             SELECT id FROM test WHERE id = 1): cache
is_select(                          SELECT * FROM test): cache
Cache put: 2
Cache hit: 1
```

The examples callback tests if a statement string matches a pattern. If this is the case, it either returns TRUE to cache the statement using the global default TTL or an alternative TTL.

To minimize application changes the callback can put into and registered in an auto prepend file.

8.4.6 Slam defense

A badly designed cache can do more harm than good. In the worst case a cache can increase database server load instead of minimizing it. An overload situation can occur if a highly shared cache entry expires (cache stampeding).

Cache entries are shared and reused to a different degree depending on the storage used. The default storage handler stores cache entries in process memory. Thus, a cache entry can be reused for the life-span of a process. Other PHP processes cannot access it. If Memcache is used, a cache entry can be shared among multiple PHP processes and even among multiple machines, depending on the set up being used.

If a highly shared cache entry stored, for example, in Memcache expires, many clients gets a cache miss. Many client requests can no longer be served from the cache but try to run the underlying query on the database server. Until the cache entry is refreshed, more and more clients contact the database server. In the worst case, a total lost of service is the result.

The overload can be avoided using a storage handler which limits the reuse of cache entries to few clients. Then, at the average, its likely that only a limited number of clients will try to refresh a cache entry concurrently.

Additionally, the built-in slam defense mechanism can and should be used. If slam defense is activated an expired cache entry is given an extended life time. The first client getting a cache miss for the expired cache entry tries to refresh the cache entry within the extended life time. All other clients requesting the cache entry are temporarily served from the cache although the original `TTL` of the cache entry has expired. The other clients will not experience a cache miss before the extended life time is over.

Example 8.9 Enabling the slam defense mechanism

```
mysqlnd_qc.slam_defense=1
mysqlnd_qc.slam_defense_ttl=1
```

The slam defense mechanism is enabled with the PHP configuration directive `mysqlnd_qc.slam_defense`. The extended life time of a cache entry is set with `mysqlnd_qc.slam_defense_ttl`.

The function `mysqlnd_qc_get_core_stats` returns an array of statistics. The statistics `slam_stale_refresh` and `slam_stale_hit` are incremented if slam defense takes place.

It is not possible to give a one-fits-all recommendation on the slam defense configuration. Users are advised to monitor and test their setup and derive settings accordingly.

8.4.7 Finding cache candidates

A statement should be considered for caching if it is executed often and has a long run time. Cache candidates are found by creating a list of statements sorted by the product of the number of executions

multiplied by the statements run time. The function `mysqlnd_qc_get_query_trace_log` returns a query log which help with the task.

Collecting a query trace is a slow operation. Thus, it is disabled by default. The PHP configuration directive `mysqlnd_qc.collect_query_trace` is used to enable it. The functions trace contains one entry for every query issued before the function is called.

Example 8.10 Collecting a query trace

```
mysqlnd_qc.enable_qc=1
mysqlnd_qc.collect_query_trace=1
```

```
<?php
/* connect to MySQL */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");

/* dummy queries to fill the query trace */
for ($i = 0; $i < 2; $i++) {
    $res = $mysqli->query("SELECT 1 AS _one FROM DUAL");
    $res->free();
}

/* dump trace */
var_dump(mysqlnd_qc_get_query_trace_log());
?>
```

The above examples will output:

```
array(2) {
  [0]=>
  array(8) {
    ["query"]=>
    string(26) "SELECT 1 AS _one FROM DUAL"
    ["origin"]=>
    string(102) "#0 qc.php(7): mysqli->query('SELECT 1 AS _on...')
#1 {main}"
    ["run_time"]=>
    int(0)
    ["store_time"]=>
    int(25)
    ["eligible_for_caching"]=>
    bool(false)
    ["no_table"]=>
    bool(false)
    ["was_added"]=>
    bool(false)
    ["was_already_in_cache"]=>
    bool(false)
  }
  [1]=>
  array(8) {
    ["query"]=>
    string(26) "SELECT 1 AS _one FROM DUAL"
    ["origin"]=>
    string(102) "#0 qc.php(7): mysqli->query('SELECT 1 AS _on...')
#1 {main}"
```

```
    ["run_time"]=>
    int(0)
    ["store_time"]=>
    int(8)
    ["eligible_for_caching"]=>
    bool(false)
    ["no_table"]=>
    bool(false)
    ["was_added"]=>
    bool(false)
    ["was_already_in_cache"]=>
    bool(false)
  }
}
```

Assorted information is given in the trace. Among them timings and the origin of the query call. The origin property holds a code backtrace to identify the source of the query. The depth of the backtrace can be limited with the PHP configuration directive `mysqlnd_qc.query_trace_bt_depth`. The default depth is `3`.

Example 8.11 Setting the backtrace depth with the `mysqlnd_qc.query_trace_bt_depth` ini setting

```
mysqlnd_qc.enable_qc=1
mysqlnd_qc.collect_query_trace=1
```

```
<?php
/* connect to MySQL */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)");

/* dummy queries to fill the query trace */
for ($i = 0; $i < 3; $i++) {
    $res = $mysqli->query("SELECT id FROM test WHERE id = " . $mysqli->real_escape_string($i));
    $res->free();
}

$trace = mysqlnd_qc_get_query_trace_log();
$summary = array();
foreach ($trace as $entry) {
    if (!isset($summary[$entry['query']])) {
        $summary[$entry['query']] = array(
            "executions" => 1,
            "time"       => $entry['run_time'] + $entry['store_time'],
        );
    } else {
        $summary[$entry['query']]['executions']++;
        $summary[$entry['query']]['time'] += $entry['run_time'] + $entry['store_time'];
    }
}

foreach ($summary as $query => $details) {
    printf("%45s: %5dms (%dx)\n",
    $query, $details['time'], $details['executions']);
}
?>
```

```
```

The above examples will output something similar to:

```
                 DROP TABLE IF EXISTS test:     0ms (1x)
                 CREATE TABLE test(id INT):     0ms (1x)
   INSERT INTO test(id) VALUES (1), (2), (3):     0ms (1x)
            SELECT id FROM test WHERE id = 0:    25ms (1x)
            SELECT id FROM test WHERE id = 1:    10ms (1x)
            SELECT id FROM test WHERE id = 2:     9ms (1x)
```

8.4.8 Measuring cache efficiency

PECL/mysqlnd_qc offers three ways to measure the cache efficiency. The function `mysqlnd_qc_get_normalized_query_trace_log` returns statistics aggregated by the normalized query string, `mysqlnd_qc_get_cache_info` gives storage handler specific information which includes a list of all cached items, depending on the storage handler. Additionally, the core of PECL/mysqlnd_qc collects high-level summary statistics aggregated per PHP process. The high-level statistics are returned by `mysqlnd_qc_get_core_stats`.

The functions `mysqlnd_qc_get_normalized_query_trace_log` and `mysqlnd_qc_get_core_stats` will not collect data unless data collection has been enabled through their corresponding PHP configuration directives. Data collection is disabled by default for performance considerations. It is configurable with the mysqlnd_qc.time_statistics option, which determines if timing information should be collected. Collection of time statistics is enabled by default but only performed if data collection as such has been enabled. Recording time statistics causes extra system calls. In most cases, the benefit of the monitoring outweighs any potential performance penalty of the additional system calls.

Example 8.12 Collecting statistics data with the `mysqlnd_qc.time_statistics` ini setting

```
mysqlnd_qc.enable_qc=1
mysqlnd_qc.collect_statistics=1
```

```
<?php
/* connect to MySQL */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)");

/* dummy queries */
for ($i = 1; $i <= 4; $i++) {
    $query = sprintf("/*%s*/SELECT id FROM test WHERE id = %d", MYSQLND_QC_ENABLE_SWITCH, $i % 2);
    $res   = $mysqli->query($query);

    $res->free();
}

var_dump(mysqlnd_qc_get_core_stats());
?>
```

The above examples will output something similar to:

```
array(26) {
  ["cache_hit"]=>
  string(1) "2"
  ["cache_miss"]=>
  string(1) "2"
  ["cache_put"]=>
  string(1) "2"
  ["query_should_cache"]=>
  string(1) "4"
  ["query_should_not_cache"]=>
  string(1) "3"
  ["query_not_cached"]=>
  string(1) "3"
  ["query_could_cache"]=>
  string(1) "4"
  ["query_found_in_cache"]=>
  string(1) "2"
  ["query_uncached_other"]=>
  string(1) "0"
  ["query_uncached_no_table"]=>
  string(1) "0"
  ["query_uncached_no_result"]=>
  string(1) "0"
  ["query_uncached_use_result"]=>
  string(1) "0"
  ["query_aggr_run_time_cache_hit"]=>
  string(2) "28"
  ["query_aggr_run_time_cache_put"]=>
  string(3) "900"
  ["query_aggr_run_time_total"]=>
  string(3) "928"
  ["query_aggr_store_time_cache_hit"]=>
  string(2) "14"
  ["query_aggr_store_time_cache_put"]=>
  string(2) "40"
  ["query_aggr_store_time_total"]=>
  string(2) "54"
  ["receive_bytes_recorded"]=>
  string(3) "136"
  ["receive_bytes_replayed"]=>
  string(3) "136"
  ["send_bytes_recorded"]=>
  string(2) "84"
  ["send_bytes_replayed"]=>
  string(2) "84"
  ["slam_stale_refresh"]=>
  string(1) "0"
  ["slam_stale_hit"]=>
  string(1) "0"
  ["request_counter"]=>
  int(1)
  ["process_hash"]=>
  int(1929695233)
}
```

For a quick overview, call `mysqlnd_qc_get_core_stats`. It delivers cache usage, cache timing and traffic related statistics. Values are aggregated on a per process basis for all queries issued by any PHP MySQL API call.

Some storage handler, such as the default handler, can report cache entries, statistics related to the entries and meta data for the underlying query through the `mysqlnd_qc_get_cache_info` function. Please note, that the information returned depends on the storage handler. Values are aggregated on a per process basis.

Example 8.13 Example `mysqlnd_qc_get_cache_info` usage

```
mysqlnd_qc.enable_qc=1
```

```
<?php
/* connect to MySQL */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)");

/* dummy queries to fill the query trace */
for ($i = 1; $i <= 4; $i++) {
    $query = sprintf("/*%s*/SELECT id FROM test WHERE id = %d", MYSQLND_QC_ENABLE_SWITCH, $i % 2);
    $res   = $mysqli->query($query);

    $res->free();
}

var_dump(mysqlnd_qc_get_cache_info());
?>
```

The above examples will output something similar to:

```
array(4) {
  ["num_entries"]=>
  int(2)
  ["handler"]=>
  string(7) "default"
  ["handler_version"]=>
  string(5) "1.0.0"
  ["data"]=>
  array(2) {
    ["Localhost via UNIX socket
3306
root
test|/*qc=on*/SELECT id FROM test WHERE id = 1"]=>
    array(2) {
      ["statistics"]=>
      array(11) {
        ["rows"]=>
        int(1)
        ["stored_size"]=>
        int(71)
        ["cache_hits"]=>
        int(1)
        ["run_time"]=>
        int(391)
        ["store_time"]=>
        int(27)
        ["min_run_time"]=>
```

```
        int(16)
        ["max_run_time"]=>
        int(16)
        ["min_store_time"]=>
        int(8)
        ["max_store_time"]=>
        int(8)
        ["avg_run_time"]=>
        int(8)
        ["avg_store_time"]=>
        int(4)
      }
      ["metadata"]=>
      array(1) {
        [0]=>
        array(8) {
          ["name"]=>
          string(2) "id"
          ["orig_name"]=>
          string(2) "id"
          ["table"]=>
          string(4) "test"
          ["orig_table"]=>
          string(4) "test"
          ["db"]=>
          string(4) "test"
          ["max_length"]=>
          int(1)
          ["length"]=>
          int(11)
          ["type"]=>
          int(3)
        }
      }
    }
    ["Localhost via UNIX socket
3306
root
test|/*qc=on*/SELECT id FROM test WHERE id = 0"]=>
    array(2) {
      ["statistics"]=>
      array(11) {
        ["rows"]=>
        int(0)
        ["stored_size"]=>
        int(65)
        ["cache_hits"]=>
        int(1)
        ["run_time"]=>
        int(299)
        ["store_time"]=>
        int(13)
        ["min_run_time"]=>
        int(11)
        ["max_run_time"]=>
        int(11)
        ["min_store_time"]=>
        int(6)
        ["max_store_time"]=>
        int(6)
        ["avg_run_time"]=>
        int(5)
        ["avg_store_time"]=>
        int(3)
      }
      ["metadata"]=>
      array(1) {
```

```
        [0]=>
        array(8) {
          ["name"]=>
          string(2) "id"
          ["orig_name"]=>
          string(2) "id"
          ["table"]=>
          string(4) "test"
          ["orig_table"]=>
          string(4) "test"
          ["db"]=>
          string(4) "test"
          ["max_length"]=>
          int(0)
          ["length"]=>
          int(11)
          ["type"]=>
          int(3)
        }
      }
    }
  }
}
```

It is possible to further break down the granularity of statistics to the level of the normalized statement string. The normalized statement string is the statements string with all parameters replaced with question marks. For example, the two statements `SELECT id FROM test WHERE id = 0` and `SELECT id FROM test WHERE id = 1` are normalized into `SELECT id FROM test WHERE id = ?`. Their both statistics are aggregated into one entry for `SELECT id FROM test WHERE id = ?`.

Example 8.14 Example `mysqlnd_qc_get_normalized_query_trace_log` usage

```
mysqlnd_qc.enable_qc=1
mysqlnd_qc.collect_normalized_query_trace=1
```

```
<?php
/* connect to MySQL */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)");

/* dummy queries to fill the query trace */
for ($i = 1; $i <= 4; $i++) {
    $query = sprintf("/*%s*/SELECT id FROM test WHERE id = %d", MYSQLND_QC_ENABLE_SWITCH, $i % 2);
    $res   = $mysqli->query($query);

    $res->free();
}

var_dump(mysqlnd_qc_get_normalized_query_trace_log());
?>
```

The above examples will output something similar to:

```
array(4) {
  [0]=>
  array(9) {
    ["query"]=>
    string(25) "DROP TABLE IF EXISTS test"
    ["occurences"]=>
    int(0)
    ["eligible_for_caching"]=>
    bool(false)
    ["avg_run_time"]=>
    int(0)
    ["min_run_time"]=>
    int(0)
    ["max_run_time"]=>
    int(0)
    ["avg_store_time"]=>
    int(0)
    ["min_store_time"]=>
    int(0)
    ["max_store_time"]=>
    int(0)
  }
  [1]=>
  array(9) {
    ["query"]=>
    string(27) "CREATE TABLE test (id INT )"
    ["occurences"]=>
    int(0)
    ["eligible_for_caching"]=>
    bool(false)
    ["avg_run_time"]=>
    int(0)
    ["min_run_time"]=>
    int(0)
    ["max_run_time"]=>
    int(0)
    ["avg_store_time"]=>
    int(0)
    ["min_store_time"]=>
    int(0)
    ["max_store_time"]=>
    int(0)
  }
  [2]=>
  array(9) {
    ["query"]=>
    string(46) "INSERT INTO test (id ) VALUES (? ), (? ), (? )"
    ["occurences"]=>
    int(0)
    ["eligible_for_caching"]=>
    bool(false)
    ["avg_run_time"]=>
    int(0)
    ["min_run_time"]=>
    int(0)
    ["max_run_time"]=>
    int(0)
    ["avg_store_time"]=>
    int(0)
    ["min_store_time"]=>
    int(0)
    ["max_store_time"]=>
    int(0)
  }
  [3]=>
  array(9) {
```

```
    ["query"]=>
    string(31) "SELECT id FROM test WHERE id =?"
    ["occurences"]=>
    int(4)
    ["eligible_for_caching"]=>
    bool(true)
    ["avg_run_time"]=>
    int(179)
    ["min_run_time"]=>
    int(11)
    ["max_run_time"]=>
    int(393)
    ["avg_store_time"]=>
    int(12)
    ["min_store_time"]=>
    int(7)
    ["max_store_time"]=>
    int(25)
  }
}
```

The source distribution of PECL/mysqlnd_qc contains a directory `web/` in which web based monitoring scripts can be found which give an example how to write a cache monitor. Please, follow the instructions given in the source.

Since PECL/mysqlnd_qc 1.1.0 it is possible to write statistics into a log file. Please, see `mysqlnd_qc.collect_statistics_log_file`.

8.4.9 Beyond TTL: user-defined storage

The query cache plugin supports the use of user-defined storage handler. User-defined storage handler can use arbitrarily complex invalidation algorithms and support arbitrary storage media.

All user-defined storage handlers have to provide a certain interface. The functions of the user-defined storage handler will be called by the core of the cache plugin. The necessary interface consists of seven public functions. Both procedural and object oriented user-defined storage handler must implement the same set of functions.

Example 8.15 Using a user-defined storage handler

```
<?php
/* Enable default caching of all statements */
ini_set("mysqlnd_qc.cache_by_default", 1);

/* Procedural user defined storage handler functions */

$__cache = array();

function get_hash($host_info, $port, $user, $db, $query) {
    global $__cache;
    printf("\t%s(%d)\n", __FUNCTION__, func_num_args());

    return md5(sprintf("%s%s%s%s%s", $host_info, $port, $user, $db, $query));
}

function find_query_in_cache($key) {
    global $__cache;
    printf("\t%s(%d)\n", __FUNCTION__, func_num_args());
```

```
    if (isset($__cache[$key])) {
        $tmp = $__cache[$key];
        if ($tmp["valid_until"] < time()) {
            unset($__cache[$key]);
            $ret = NULL;
        } else {
            $ret = $__cache[$key]["data"];
        }
    } else {
        $ret = NULL;
    }

    return $ret;
}

function return_to_cache($key) {
    /*
     Called on cache hit after cached data has been processed,
     may be used for reference counting
    */
    printf("\t%s(%d)\n", __FUNCTION__, func_num_args());
}

function add_query_to_cache_if_not_exists($key, $data, $ttl, $run_time, $store_time, $row_count) {
    global $__cache;
    printf("\t%s(%d)\n", __FUNCTION__, func_num_args());

    $__cache[$key] = array(
        "data"               => $data,
        "row_count"          => $row_count,
        "valid_until"        => time() + $ttl,
        "hits"               => 0,
        "run_time"           => $run_time,
        "store_time"         => $store_time,
        "cached_run_times"   => array(),
        "cached_store_times" => array(),
    );

    return TRUE;
}

function query_is_select($query) {
    printf("\t%s('%s'): ", __FUNCTION__, $query);

    $ret = FALSE;
    if (stristr($query, "SELECT") !== FALSE) {
        /* cache for 5 seconds */
        $ret = 5;
    }

    printf("%s\n", (FALSE === $ret) ? "FALSE" : $ret);
    return $ret;
}

function update_query_run_time_stats($key, $run_time, $store_time) {
    global $__cache;
    printf("\t%s(%d)\n", __FUNCTION__, func_num_args());

    if (isset($__cache[$key])) {
        $__cache[$key]['hits']++;
        $__cache[$key]["cached_run_times"][] = $run_time;
        $__cache[$key]["cached_store_times"][] = $store_time;
    }
}

function get_stats($key = NULL) {
```

```
    global $__cache;
    printf("\t%s(%d)\n", __FUNCTION__, func_num_args());

    if ($key && isset($__cache[$key])) {
        $stats = $__cache[$key];
    } else {
        $stats = array();
        foreach ($__cache as $key => $details) {
            $stats[$key] = array(
               'hits'              => $details['hits'],
               'bytes'             => strlen($details['data']),
               'uncached_run_time' => $details['run_time'],
               'cached_run_time'   => (count($details['cached_run_times']))
                                      ? array_sum($details['cached_run_times']) / count($details['cached_run_t
                                      : 0,
            );
        }
    }

    return $stats;
}

function clear_cache() {
    global $__cache;
    printf("\t%s(%d)\n", __FUNCTION__, func_num_args());

    $__cache = array();
    return TRUE;
}

/* Install procedural user-defined storage handler */
if (!mysqlnd_qc_set_user_handlers("get_hash", "find_query_in_cache",
    "return_to_cache", "add_query_to_cache_if_not_exists",
    "query_is_select", "update_query_run_time_stats", "get_stats", "clear_cache")) {

        printf("Failed to install user-defined storage handler\n");
}


/* Connect, create and populate test table */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2)");

printf("\nCache put/cache miss\n");

$res = $mysqli->query("SELECT id FROM test WHERE id = 1");
var_dump($res->fetch_assoc());
$res->free();

/* Delete record to verify we get our data from the cache */
$mysqli->query("DELETE FROM test WHERE id = 1");

printf("\nCache hit\n");

$res = $mysqli->query("SELECT id FROM test WHERE id = 1");
var_dump($res->fetch_assoc());
$res->free();

printf("\nDisplay cache statistics\n");
var_dump(mysqlnd_qc_get_cache_info());

printf("\nFlushing cache, cache put/cache miss");
var_dump(mysqlnd_qc_clear_cache());

$res = $mysqli->query("SELECT id FROM test WHERE id = 1");
```

```
var_dump($res->fetch_assoc());
$res->free();
?>
```

The above examples will output something similar to:

```
        query_is_select('DROP TABLE IF EXISTS test'): FALSE
        query_is_select('CREATE TABLE test(id INT)'): FALSE
        query_is_select('INSERT INTO test(id) VALUES (1), (2)'): FALSE

Cache put/cache miss
        query_is_select('SELECT id FROM test WHERE id = 1'): 5
        get_hash(5)
        find_query_in_cache(1)
        add_query_to_cache_if_not_exists(6)
array(1) {
  ["id"]=>
  string(1) "1"
}
        query_is_select('DELETE FROM test WHERE id = 1'): FALSE

Cache hit
        query_is_select('SELECT id FROM test WHERE id = 1'): 5
        get_hash(5)
        find_query_in_cache(1)
        return_to_cache(1)
        update_query_run_time_stats(3)
array(1) {
  ["id"]=>
  string(1) "1"
}

Display cache statistics
        get_stats(0)
array(4) {
  ["num_entries"]=>
  int(1)
  ["handler"]=>
  string(4) "user"
  ["handler_version"]=>
  string(5) "1.0.0"
  ["data"]=>
  array(1) {
    ["18683c177dc89bb352b29965d112fdaa"]=>
    array(4) {
      ["hits"]=>
      int(1)
      ["bytes"]=>
      int(71)
      ["uncached_run_time"]=>
      int(398)
      ["cached_run_time"]=>
      int(4)
    }
  }
}

Flushing cache, cache put/cache miss    clear_cache(0)
bool(true)
        query_is_select('SELECT id FROM test WHERE id = 1'): 5
        get_hash(5)
        find_query_in_cache(1)
        add_query_to_cache_if_not_exists(6)
```

```
NULL
```

8.5 Installing/Configuring

8.5.1 Requirements

`PHP 5.3.3` or a newer version of `PHP`.

PECL/mysqlnd_qc is a mysqlnd plugin. It plugs into the mysqlnd library. To use you this plugin with a PHP MySQL extension, the extension (mysqli, mysql, or PDO_MYSQL) must enable the mysqlnd library.

For using the APC storage handler with PECL/mysqlnd_qc 1.0 `APC 3.1.3p1-beta` or newer. PECL/mysqlnd_qc 1.2 has been tested with `APC 3.1.13-beta`. The APC storage handler cannot be used with a shared build. You cannot use the PHP configuration directive `extension` to load the APC and PECL/mysqlnd_qc extensions if PECL/mysqlnd_qc will use APC as a storage handler. For using the APC storage handler, you have to statically compile PHP with APC and PECL/mysqlnd_qc support into PHP.

For using `MEMCACHE` storage handler: Use `libmemcache 0.38` or newer. PECL/mysqlnd_qc 1.2 has been tested with `libmemcache 1.4.0`.

For using `sqlite` storage handler: Use the sqlite3 extension that bundled with PHP.

8.5.2 Installation

This PECL extension is not bundled with PHP.

Information for installing this PECL extension may be found in the manual chapter titled Installation of PECL extensions. Additional information such as new releases, downloads, source files, maintainer information, and a CHANGELOG, can be located here: http://pecl.php.net/package/mysqlnd_qc

A DLL for this PECL extension is currently unavailable. See also the building on Windows section.

8.5.3 Runtime Configuration

The behaviour of these functions is affected by settings in `php.ini`.

Table 8.1 mysqlnd_qc Configure Options

Name	Default	Changeable	Changelog
mysqlnd_qc.enable_qc	1	PHP_INI_SYSTEM	
mysqlnd_qc.ttl	30	PHP_INI_ALL	
mysqlnd_qc.cache_by_default	0	PHP_INI_ALL	
mysqlnd_qc.cache_no_table	0	PHP_INI_ALL	

Name	Default	Changeable	Changelog
mysqlnd_qc.use_request_time	0	PHP_INI_ALL	
mysqlnd_qc.time_statistics	1	PHP_INI_ALL	
mysqlnd_qc.collect_statistics	0	PHP_INI_ALL	
mysqlnd_qc.collect_statistics_log_file	/tmp/mysqlnd_qc.stats	PHP_INI_SYSTEM	
mysqlnd_qc.collect_query_trace	0	PHP_INI_SYSTEM	
mysqlnd_qc.query_trace_bt_depth	3	PHP_INI_SYSTEM	
mysqlnd_qc.collect_normalized_query_trace	0	PHP_INI_SYSTEM	
mysqlnd_qc.ignore_sql_comments	1	PHP_INI_ALL	
mysqlnd_qc.slam_defense	0	PHP_INI_SYSTEM	
mysqlnd_qc.slam_defense_ttl	0	PHP_INI_SYSTEM	
mysqlnd_qc.std_data_copy	0	PHP_INI_SYSTEM	
mysqlnd_qc.apc_prefix	qc_	PHP_INI_ALL	
mysqlnd_qc.memc_server	127.0.0.1	PHP_INI_ALL	
mysqlnd_qc.memc_port	11211	PHP_INI_ALL	
mysqlnd_qc.sqlite_data_file	:memory:	PHP_INI_ALL	

Here's a short explanation of the configuration directives.

mysqlnd_qc.enable_qc integer
Enables or disables the plugin. If disabled the extension will not plug into mysqlnd to proxy internal mysqlnd C API calls.

mysqlnd_qc.ttl integer
Default Time-to-Live (TTL) for cache entries in seconds.

mysqlnd_qc.cache_by_default integer
Cache all queries regardless if they begin with the SQL hint that enables caching of a query or not. Storage handler cannot overrule the setting. It is evaluated by the core of the plugin.

mysqlnd_qc.cache_no_table integer
Whether to cache queries with no table name in any of columns meta data of their result set, for example, `SELECT SLEEP(1)`, `SELECT NOW()`, `SELECT SUBSTRING()`.

mysqlnd_qc.use_request_time integer
Use PHP global request time to avoid `gettimeofday()` system calls? If using `APC` storage handler it should be set to the value of `apc.use_request_time`, if not warnings will be generated.

mysqlnd_qc.time_statistics integer
Collect run time and store time statistics using `gettimeofday()` system call? Data will be collected only if you also set `mysqlnd_qc.collect_statistics = 1`,

mysqlnd_qc.collect_statistics integer
Collect statistics for `mysqlnd_qc_get_core_stats`? Does not influence storage handler statistics! Handler statistics can be an integral part of the handler internal storage format. Therefore, collection of some handler statistics cannot be disabled.

mysqlnd_qc.collect_statistics_log_file integer
If `mysqlnd_qc.collect_statistics` and `mysqlnd_qc.collect_statistics_log_file` are set, the plugin will dump statistics into the specified log file at every 10th web request during PHP request shutdown. The log file needs to be writable by the web server user.

Since 1.1.0.

mysqlnd_qc.collect_query_trace
integer
: Collect query back traces?

mysqlnd_qc.query_trace_bt_depth
integer
: Maximum depth/level of a query code backtrace.

mysqlnd_qc.ignore_sql_comments
integer
: Whether to remove SQL comments from a query string before hashing it to generate a cache key. Disable if you do not want two statemts such as `SELECT /*my_source_ip=123*/ id FROM test` and `SELECT /*my_source_ip=456*/ id FROM test` to refer to the same cache entry.

  Since 1.1.0.

mysqlnd_qc.slam_defense
integer
: Activates handler based slam defense (cache stampeding protection) if available. Supported by `Default` and `APC` storage handler

mysqlnd_qc.slam_defense_ttl
integer
: `TTL` for stale cache entries which are served while another client updates the entries. Supported by `APC` storage handler.

mysqlnd_qc.collect_normalized_query_trace
integer
: Collect aggregated normalized query traces? The setting has no effect by default. You compile the extension using the define `NORM_QUERY_TRACE_LOG` to make use of the setting.

mysqlnd_qc.std_data_copy
integer
: Default storage handler: copy cached wire data? EXPERIMENTAL – use default setting!

mysqlnd_qc.apc_prefix
string
: The `APC` storage handler stores data in the `APC` user cache. The setting sets a prefix to be used for cache entries.

mysqlnd_qc.memc_server
string
: `MEMCACHE` storage handler: memcache server host.

mysqlnd_qc.memc_port
integer
: `MEMCACHE` storage handler: memcached server port.

mysqlnd_qc.sqlite_data_file
string
: `sqlite` storage handler: data file. Any setting but `:memory:` may be of little practical value.

8.6 Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

SQL hint related

Example 8.16 Using SQL hint constants

The query cache is controlled by SQL hints. SQL hints are used to enable and disable caching. SQL hints can be used to set the `TTL` of a query.

The SQL hints recognized by the query cache can be manually changed at compile time. This makes it possible to use `mysqlnd_qc` in environments in which the default SQL hints are already taken and

interpreted by other systems. Therefore it is recommended to use the SQL hint string constants instead of manually adding the default SQL hints to the query string.

```
<?php
/* Use constants for maximum portability */
$query = "/*" . MYSQLND_QC_ENABLE_SWITCH . "*/SELECT id FROM test";

/* Valid but less portable: default TTL */
$query = "/*qc=on*/SELECT id FROM test";

/* Valid but less portable: per statement TTL */
$query = "/*qc=on*//*qc_ttl=5*/SELECT id FROM test";

printf("MYSQLND_QC_ENABLE_SWITCH: %s\n", MYSQLND_QC_ENABLE_SWITCH);
printf("MYSQLND_QC_DISABLE_SWITCH: %s\n", MYSQLND_QC_DISABLE_SWITCH);
printf("MYSQLND_QC_TTL_SWITCH: %s\n", MYSQLND_QC_TTL_SWITCH);
?>
```

The above examples will output:

```
MYSQLND_QC_ENABLE_SWITCH: qc=on
MYSQLND_QC_DISABLE_SWITCH: qc=off
MYSQLND_QC_TTL_SWITCH: qc_ttl=
```

`MYSQLND_QC_ENABLE_SWITCH` (string)
: SQL hint used to enable caching of a query.

`MYSQLND_QC_DISABLE_SWITCH` (string)
: SQL hint used to disable caching of a query if `mysqlnd_qc.cache_by_default = 1`.

`MYSQLND_QC_TTL_SWITCH` (string)
: SQL hint used to set the TTL of a result set.

`MYSQLND_QC_SERVER_ID_SWITCH` (string)
: This SQL hint should not be used in general.

  It is needed by PECL/mysqlnd_ms to group cache entries for one statement but originating from different physical connections. If the hint is used connection settings such as user, hostname and charset are not considered for generating a cache key of a query. Instead the given value and the query string are used as input to the hashing function that generates the key.

  PECL/mysqlnd_ms may, if instructed, cache results from MySQL Replication slaves. Because it can hold many connections to the slave the cache key shall not be formed from the user, hostname or other settings that may vary for the various slave connections. Instead, PECL/mysqlnd_ms provides an identifier which refers to the group of slave connections that shall be enabled to share cache entries no matter which physical slave connection was to generate the cache entry.

  Use of this feature outside of PECL/mysqlnd_ms is not recommended.

`mysqlnd_qc_set_cache_condition` related

Example 8.17 Example `mysqlnd_qc_set_cache_condition` usage

The function `mysqlnd_qc_set_cache_condition` allows setting conditions for automatic caching of statements which don't begin with the SQL hints necessary to manually enable caching.

```
<?php
/* Cache all accesses to tables with the name "new%" in schema/database "db_example" for 1 second */
if (!mysqlnd_qc_set_cache_condition(MYSQLND_QC_CONDITION_META_SCHEMA_PATTERN, "db_example.new%", 1)) {
  die("Failed to set cache condition!");
}

$mysqli = new mysqli("host", "user", "password", "db_example", "port");
/* cached although no SQL hint given  */
$mysqli->query("SELECT id, title FROM news");

$pdo_mysql = new PDO("mysql:host=host;dbname=db_example;port=port", "user", "password");
/* not cached: no SQL hint, no pattern match */
$pdo_mysql->query("SELECT id, title FROM latest_news");
/* cached: TTL 1 second, pattern match */
$pdo_mysql->query("SELECT id, title FROM news");
?>
```

`MYSQLND_QC_CONDITION_META_SCHEMA_PATTERN` (int) — Used as a parameter of `mysqlnd_qc_set_cache_condition` to set conditions for schema based automatic caching.

Other

The plugin version number can be obtained using either `MYSQLND_QC_VERSION`, which is the string representation of the numerical version number, or `MYSQLND_QC_VERSION_ID`, which is an integer such as 10000. Developers can calculate the version number as follows.

Version (part)	Example
Major*10000	1*10000 = 10000
Minor*100	0*100 = 0
Patch	0 = 0
MYSQLND_QC_VERSION_ID	10000

`MYSQLND_QC_VERSION` (string) — Plugin version string, for example, “1.0.0-prototype”.

`MYSQLND_QC_VERSION_ID` (int) — Plugin version number, for example, 10000.

8.7 mysqlnd_qc Functions

8.7.1 `mysqlnd_qc_clear_cache`

- `mysqlnd_qc_clear_cache`

  Flush all cache contents

Description

```
bool mysqlnd_qc_clear_cache();
```

Flush all cache contents.

Flushing the cache is a storage handler responsibility. All built-in storage handler but the `memcache` storage handler support flushing the cache. The `memcache` storage handler cannot flush its cache contents.

User-defined storage handler may or may not support the operation.

Parameters

This function has no parameters.

Return Values

Returns `TRUE` on success or `FALSE` on failure.

A return value of `FALSE` indicates that flushing all cache contents has failed or the operation is not supported by the active storage handler. Applications must not expect that calling the function will always flush the cache.

8.7.2 `mysqlnd_qc_get_available_handlers`

- `mysqlnd_qc_get_available_handlers`

  Returns a list of available storage handler

Description

```
array mysqlnd_qc_get_available_handlers();
```

Which storage are available depends on the compile time configuration of the query cache plugin. The `default` storage handler is always available. All other storage handler must be enabled explicitly when building the extension.

Parameters

This function has no parameters.

Return Values

Returns an array of available built-in storage handler. For each storage handler the version number and version string is given.

Examples

Example 8.18 `mysqlnd_qc_get_available_handlers` example

```
<?php
var_dump(mysqlnd_qc_get_available_handlers());
?>
```

The above examples will output:

```
array(5) {
  ["default"]=>
  array(2) {
    ["version"]=>
    string(5) "1.0.0"
    ["version_number"]=>
    int(100000)
  }
  ["user"]=>
  array(2) {
    ["version"]=>
    string(5) "1.0.0"
    ["version_number"]=>
    int(100000)
  }
  ["APC"]=>
  array(2) {
    ["version"]=>
    string(5) "1.0.0"
    ["version_number"]=>
    int(100000)
  }
  ["MEMCACHE"]=>
  array(2) {
    ["version"]=>
    string(5) "1.0.0"
    ["version_number"]=>
    int(100000)
  }
  ["sqlite"]=>
  array(2) {
    ["version"]=>
    string(5) "1.0.0"
    ["version_number"]=>
    int(100000)
  }
}
```

See Also

Installation
mysqlnd_qc_set_storage_handler

8.7.3 mysqlnd_qc_get_cache_info

- mysqlnd_qc_get_cache_info

  Returns information on the current handler, the number of cache entries and cache entries, if available

Description

```
  array mysqlnd_qc_get_cache_info();
```

Parameters

This function has no parameters.

Return Values

Returns information on the current handler, the number of cache entries and cache entries, if available. If and what data will be returned for the cache entries is subject to the active storage handler. Storage handler are free to return any data. Storage handler are recommended to return at least the data provided by the default handler, if technically possible.

The scope of the information is the PHP process. Depending on the PHP deployment model a process may serve one or more web requests.

Values are aggregated for all cache activities on a per storage handler basis. It is not possible to tell how much queries originating from `mysqli`, `PDO_MySQL` or `mysql`.API calls have contributed to the aggregated data values. Use `mysqlnd_qc_get_core_stats` to get timing data aggregated for all storage handlers.

Array of cache information

`handler` string — The active storage handler.

All storage handler. Since 1.0.0.

`handler_version` string — The version of the active storage handler.

All storage handler. Since 1.0.0.

`num_entries` int — The number of cache entries. The value depends on the storage handler in use.

The default, APC and SQLite storage handler provide the actual number of cache entries.

The MEMCACHE storage handler always returns `0`. MEMCACHE does not support counting the number of cache entries.

If a user defined handler is used, the number of entries of the `data` property is reported.

Since 1.0.0.

`data` array — The version of the active storage handler.

Additional storage handler dependent data on the cache entries. Storage handler are requested to provide similar and comparable information. A user defined storage handler is free to return any data.

Since 1.0.0.

The following information is provided by the default storage handler for the `data` property.

The `data` property holds a hash. The hash is indexed by the internal cache entry identifier of the storage handler. The cache entry identifier is human-readable and contains the query string leading to the cache entry. Please, see also the example below. The following data is given for every cache entry.

`statistics` array — Statistics of the cache entry.

Since 1.0.0.

Property	Description	Version
rows	Number of rows of the cached result set.	Since 1.0.0.
stored_size	The size of the cached result set in bytes. This is the size of the payload. The value is not suited for calculating the total memory consumption of all cache entries including the administrative overhead of the cache entries.	Since 1.0.0.
cache_hits	How often the cached entry has been returned.	Since 1.0.0.
run_time	Run time of the statement to which the cache entry belongs. This is the run time of the uncached statement. It is the time between sending the statement to MySQL receiving a reply from MySQL. Run time saved by using the query cache plugin can be calculated like this: `cache_hits * ((run_time - avg_run_time) + (store_time - avg_store_time))`.	Since 1.0.0.
store_time	Store time of the statements result set to which the cache entry belongs. This is the time it took to fetch and store the results of the uncached statement.	Since 1.0.0.
min_run_time	Minimum run time of the cached statement. How long it took to find the statement in the cache.	Since 1.0.0.
min_store_time	Minimum store time of the cached statement. The time taken for fetching the cached result set from the storage medium and decoding	Since 1.0.0.

Property	Description	Version
avg_run_time	Average run time of the cached statement.	Since 1.0.0.
avg_store_time	Average store time of the cached statement.	Since 1.0.0.
max_run_time	Average run time of the cached statement.	Since 1.0.0.
max_store_time	Average store time of the cached statement.	Since 1.0.0.
valid_until	Timestamp when the cache entry expires.	Since 1.1.0.

metadata array

Metadata of the cache entry. This is the metadata provided by MySQL together with the result set of the statement in question. Different versions of the MySQL server may return different metadata. Unlike with some of the PHP MySQL extensions no attempt is made to hide MySQL server version dependencies and version details from the caller. Please, refer to the MySQL C API documentation that belongs to the MySQL server in use for further details.

The metadata list contains one entry for every column.

Since 1.0.0.

Property	Description	Version
name	The field name. Depending on the MySQL version this may be the fields alias name.	Since 1.0.0.
org_name	The field name.	Since 1.0.0.
table	The table name. If an alias name was used for the table, this usually holds the alias name.	Since 1.0.0.
org_table	The table name.	Since 1.0.0.
db	The database/schema name.	Since 1.0.0.
max_length	The maximum width of the field. Details may	Since 1.0.0.

Property	Description	Version
	vary by MySQL server version.	
length	The width of the field. Details may vary by MySQL server version.	Since 1.0.0.
type	The data type of the field. Details may vary by the MySQL server in use. This is the MySQL C API type constants value. It is recommended to use type constants provided by the mysqli extension to test for its meaning. You should not test for certain type values by comparing with certain numbers.	Since 1.0.0.

The APC storage handler returns the same information for the data property but no metadata. The metadata of a cache entry is set to NULL.

The MEMCACHE storage handler does not fill the data property. Statistics are not available on a per cache entry basis with the MEMCACHE storage handler.

A user defined storage handler is free to provide any data.

Examples

Example 8.19 mysqlnd_qc_get_cache_info example

The example shows the output from the built-in default storage handler. Other storage handler may report different data.

```
<?php
/* Populate the cache, e.g. using mysqli */
$mysqli = new mysqli("host", "user", "password", "schema");
$mysqli->query("/*" . MYSQLND_QC_ENABLE_SWITCH . "*/SELECT id FROM test");

/* Display cache information */
var_dump(mysqlnd_qc_get_cache_info());
?>
```

The above examples will output:

```
array(4) {
```

```
["num_entries"]=>
int(1)
["handler"]=>
string(7) "default"
["handler_version"]=>
string(5) "1.0.0"
["data"]=>
array(1) {
  ["Localhost via UNIX socket 3306 user schema|/*qc=on*/SELECT id FROM test"]=>
  array(2) {
    ["statistics"]=>
    array(11) {
      ["rows"]=>
      int(6)
      ["stored_size"]=>
      int(101)
      ["cache_hits"]=>
      int(0)
      ["run_time"]=>
      int(471)
      ["store_time"]=>
      int(27)
      ["min_run_time"]=>
      int(0)
      ["max_run_time"]=>
      int(0)
      ["min_store_time"]=>
      int(0)
      ["max_store_time"]=>
      int(0)
      ["avg_run_time"]=>
      int(0)
      ["avg_store_time"]=>
      int(0)
    }
    ["metadata"]=>
    array(1) {
      [0]=>
      array(8) {
        ["name"]=>
        string(2) "id"
        ["orig_name"]=>
        string(2) "id"
        ["table"]=>
        string(4) "test"
        ["orig_table"]=>
        string(4) "test"
        ["db"]=>
        string(4) "schema"
        ["max_length"]=>
        int(1)
        ["length"]=>
        int(11)
        ["type"]=>
        int(3)
      }
    }
  }
}
}
```

See Also

mysqlnd_qc_get_core_stats

8.7.4 mysqlnd_qc_get_core_stats

- mysqlnd_qc_get_core_stats

  Statistics collected by the core of the query cache

Description

```
array mysqlnd_qc_get_core_stats();
```

Returns an array of statistics collected by the core of the cache plugin. The same data fields will be reported for any storage handler because the data is collected by the core.

The PHP configuration setting mysqlnd_qc.collect_statistics controls the collection of statistics. The collection of statistics is disabled by default for performance reasons. Disabling the collection of statistics will also disable the collection of time related statistics.

The PHP configuration setting mysqlnd_qc.collect_time_statistics controls the collection of time related statistics.

The scope of the core statistics is the PHP process. Depending on your deployment model a PHP process may handle one or multiple requests.

Statistics are aggregated for all cache entries and all storage handler. It is not possible to tell how much queries originating from mysqli, PDO_MySQL or mysql API calls have contributed to the aggregated data values.

Parameters

This function has no parameters.

Return Values

Array of core statistics

Statistic	Description	Version
cache_hit	Statement is considered cacheable and cached data has been reused. Statement is considered cacheable and a cache miss happened but the statement got cached by someone else while we process it and thus we can fetch the result from the refreshed cache.	Since 1.0.0.
cache_miss	Statement is considered cacheable... • ... and has been added to the cache • ... but the PHP configuration directive setting of mysqlnd_qc.cache_no_table = 1 has prevented caching.	Since 1.0.0.

Statistic	Description	Version
	• ... but an unbuffered result set is requested. • ... but a buffered result set was empty.	
cache_put	Statement is considered cacheable and has been added to the cache. Take care when calculating derived statistics. Storage handler with a storage life time beyond process scope may report cache_put = 0 together with cache_hit > 0, if another process has filled the cache. You may want to use num_entries from mysqlnd_qc_get_cache_info if the handler supports it (default, APC).	Since 1.0.0.
query_should_cache	Statement is considered cacheable based on query string analysis. The statement may or may not be added to the cache. See also cache_put.	Since 1.0.0.
query_should_not_cache	Statement is considered not cacheable based on query string analysis.	Since 1.0.0.
query_not_cached	Statement is considered not cacheable or it is considered cachable but the storage handler has not returned a hash key for it.	Since 1.0.0.
query_could_cache	Statement is considered cacheable... • ... and statement has been run without errors • ... and meta data shows at least one column in the result set The statement may or may not be in the cache already. It may or may not be added to the cache later on.	Since 1.0.0.
query_found_in_cache	Statement is considered cacheable and we have found it in the cache but we have not replayed the cached data yet and we have not send the result set to the client yet. This is not considered a cache hit because	Since 1.0.0.

Statistic	Description	Version
	the client might not fetch the result or the cached data may be faulty.	
query_uncached_other	Statement is considered cacheable and it may or may not be in the cache already but either replaying cached data has failed, no result set is available or some other error has happened.	
query_uncached_no_table	Statement has not been cached because the result set has at least one column which has no table name in its meta data. An example of such a query is SELECT SLEEP(1). To cache those statements you have to change default value of the PHP configuration directive mysqlnd_qc.cache_no_table and set mysqlnd_qc.cache_no_table = 1. Often, it is not desired to cache such statements.	Since 1.0.0.
query_uncached_use_result	Statement would have been cached if a buffered result set had been used. The situation is also considered as a cache miss and cache_miss will be incremented as well.	Since 1.0.0.
query_aggr_run_time_cache_hit	Aggregated run time (ms) of all cached queries. Cached queries are those which have incremented cache_hit.	Since 1.0.0.
query_aggr_run_time_cache_put	Aggregated run time (ms) of all uncached queries that have been put into the cache. See also cache_put.	Since 1.0.0.
query_aggr_run_time_total	Aggregated run time (ms) of all uncached and cached queries that have been inspected and executed by the query cache.	Since 1.0.0.
query_aggr_store_time_cache_hit	Aggregated store time (ms) of all cached queries. Cached queries are those which have incremented cache_hit.	Since 1.0.0.
query_aggr_store_time_cache_put	Aggregated store time (ms) of all uncached queries that have been put into the cache. See also cache_put.	Since 1.0.0.

Statistic	Description	Version
query_aggr_store_time_total	Aggregated store time (ms) of all uncached and cached queries that have been inspected and executed by the query cache.	Since 1.0.0.
receive_bytes_recorded	Recorded incoming network traffic (bytes) send from MySQL to PHP. The traffic may or may not have been added to the cache. The traffic is the total for all queries regardless if cached or not.	Since 1.0.0.
receive_bytes_replayed	Network traffic replayed during cache. This is the total amount of incoming traffic saved because of the usage of the query cache plugin.	Since 1.0.0.
send_bytes_recorded	Recorded outgoing network traffic (bytes) send from MySQL to PHP. The traffic may or may not have been added to the cache. The traffic is the total for all queries regardless if cached or not.	Since 1.0.0.
send_bytes_replayed	Network traffic replayed during cache. This is the total amount of outgoing traffic saved because of the usage of the query cache plugin.	Since 1.0.0.
slam_stale_refresh	Number of cache misses which triggered serving stale data until the client causing the cache miss has refreshed the cache entry.	Since 1.0.0.
slam_stale_hit	Number of cache hits while a stale cache entry gets refreshed.	Since 1.0.0.

Examples

Example 8.20 `mysqlnd_qc_get_core_stats` example

```
<?php
/* Enable collection of statistics - default: disabled */
ini_set("mysqlnd_qc.collect_statistics", 1);

/* Enable collection of all timing related statistics -
default: enabled but overruled by mysqlnd_qc.collect_statistics = 0 */
ini_set("mysqlnd_qc.collect_time_statistics", 1);

/* Populate the cache, e.g. using mysqli */
$mysqli = new mysqli('host', 'user', 'password', 'schema');

/* Cache miss and cache put */
$mysqli->query("/*qc=on*/SELECT id FROM test");
```

```
/* Cache hit */
$mysqli->query("/*qc=on*/SELECT id FROM test");

/* Display core statistics */
var_dump(mysqlnd_qc_get_core_stats());
?>
```

The above examples will output:

```
array(26) {
  ["cache_hit"]=>
  string(1) "1"
  ["cache_miss"]=>
  string(1) "1"
  ["cache_put"]=>
  string(1) "1"
  ["query_should_cache"]=>
  string(1) "2"
  ["query_should_not_cache"]=>
  string(1) "0"
  ["query_not_cached"]=>
  string(1) "0"
  ["query_could_cache"]=>
  string(1) "2"
  ["query_found_in_cache"]=>
  string(1) "1"
  ["query_uncached_other"]=>
  string(1) "0"
  ["query_uncached_no_table"]=>
  string(1) "0"
  ["query_uncached_no_result"]=>
  string(1) "0"
  ["query_uncached_use_result"]=>
  string(1) "0"
  ["query_aggr_run_time_cache_hit"]=>
  string(1) "4"
  ["query_aggr_run_time_cache_put"]=>
  string(3) "395"
  ["query_aggr_run_time_total"]=>
  string(3) "399"
  ["query_aggr_store_time_cache_hit"]=>
  string(1) "2"
  ["query_aggr_store_time_cache_put"]=>
  string(1) "8"
  ["query_aggr_store_time_total"]=>
  string(2) "10"
  ["receive_bytes_recorded"]=>
  string(2) "65"
  ["receive_bytes_replayed"]=>
  string(2) "65"
  ["send_bytes_recorded"]=>
  string(2) "29"
  ["send_bytes_replayed"]=>
  string(2) "29"
  ["slam_stale_refresh"]=>
  string(1) "0"
  ["slam_stale_hit"]=>
  string(1) "0"
  ["request_counter"]=>
  int(1)
  ["process_hash"]=>
  int(3547549858)
}
```

```
```

See Also

Runtime configuration
mysqlnd_qc.collect_statistics
mysqlnd_qc.time_statistics
mysqlnd_qc_get_cache_info

8.7.5 mysqlnd_qc_get_normalized_query_trace_log

- mysqlnd_qc_get_normalized_query_trace_log

  Returns a normalized query trace log for each query inspected by the query cache

Description

```
array mysqlnd_qc_get_normalized_query_trace_log();
```

Returns a normalized query trace log for each query inspected by the query cache. The collection of the trace log is disabled by default. To collect the trace log you have to set the PHP configuration directive mysqlnd_qc.collect_normalized_query_trace to 1

Entries in the trace log are grouped by the normalized query statement. The normalized query statement is the query statement with all statement parameter values being replaced with a question mark. For example, the two statements SELECT id FROM test WHERE id = 1 and SELECT id FROM test WHERE id = 2 are normalized as SELECT id FROM test WHERE id = ?. Whenever a statement is inspected by the query cache which matches the normalized statement pattern, its statistics are grouped by the normalized statement string.

Parameters

This function has no parameters.

Return Values

An array of query log. Every list entry contains the normalized query stringand further detail information.

Key	Description
query	Normalized statement string.
occurences	How many statements have matched the normalized statement string in addition to the one which has created the log entry. The value is zero if a statement has been normalized, its normalized representation has been added to the log but no further queries inspected by PECL/mysqlnd_qc have the same normalized statement string.
eligible_for_caching	Whether the statement could be cached. An statement eligible for caching has not necessarily been cached. It not possible to tell for sure if or how many cached statement have contributed to the aggregated normalized statement log entry. However, comparing the minimum and average run time one can make an educated guess.
avg_run_time	The average run time of all queries contributing to the query log entry. The run time is the time between sending the query statement to MySQL and receiving an answer from MySQL.
avg_store_time	The average store time of all queries contributing to the query log entry. The store time is the time needed to fetch a statements result set from the server to the client and, storing it on the client.

Key	Description
min_run_time	The minimum run time of all queries contributing to the query log entry.
min_store_time	The minimum store time of all queries contributing to the query log entry.
max_run_time	The maximum run time of all queries contributing to the query log entry.
max_store_time	The maximum store time of all queries contributing to the query log entry.

Examples

Example 8.21 `mysqlnd_qc_get_normalized_query_trace_log` example

```
mysqlnd_qc.collect_normalized_query_trace=1
```

```
<?php
/* Connect, create and populate test table */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2)");

/* not cached */
$res = $mysqli->query("SELECT id FROM test WHERE id = 1");
var_dump($res->fetch_assoc());
$res->free();

/* cache put */
$res = $mysqli->query("/*" . MYSQLND_QC_ENABLE_SWITCH . "*/" . "SELECT id FROM test WHERE id = 2");
var_dump($res->fetch_assoc());
$res->free();

/* cache hit */
$res = $mysqli->query("/*" . MYSQLND_QC_ENABLE_SWITCH . "*/" . "SELECT id FROM test WHERE id = 2");
var_dump($res->fetch_assoc());
$res->free();

var_dump(mysqlnd_qc_get_normalized_query_trace_log());
?>
```

The above examples will output:

```
array(1) {
  ["id"]=>
  string(1) "1"
}
array(1) {
  ["id"]=>
  string(1) "2"
}
array(1) {
  ["id"]=>
  string(1) "2"
}
array(4) {
  [0]=>
  array(9) {
```

```
    ["query"]=>
    string(25) "DROP TABLE IF EXISTS test"
    ["occurences"]=>
    int(0)
    ["eligible_for_caching"]=>
    bool(false)
    ["avg_run_time"]=>
    int(0)
    ["min_run_time"]=>
    int(0)
    ["max_run_time"]=>
    int(0)
    ["avg_store_time"]=>
    int(0)
    ["min_store_time"]=>
    int(0)
    ["max_store_time"]=>
    int(0)
  }
  [1]=>
  array(9) {
    ["query"]=>
    string(27) "CREATE TABLE test (id INT )"
    ["occurences"]=>
    int(0)
    ["eligible_for_caching"]=>
    bool(false)
    ["avg_run_time"]=>
    int(0)
    ["min_run_time"]=>
    int(0)
    ["max_run_time"]=>
    int(0)
    ["avg_store_time"]=>
    int(0)
    ["min_store_time"]=>
    int(0)
    ["max_store_time"]=>
    int(0)
  }
  [2]=>
  array(9) {
    ["query"]=>
    string(40) "INSERT INTO test (id ) VALUES (? ), (? )"
    ["occurences"]=>
    int(0)
    ["eligible_for_caching"]=>
    bool(false)
    ["avg_run_time"]=>
    int(0)
    ["min_run_time"]=>
    int(0)
    ["max_run_time"]=>
    int(0)
    ["avg_store_time"]=>
    int(0)
    ["min_store_time"]=>
    int(0)
    ["max_store_time"]=>
    int(0)
  }
  [3]=>
  array(9) {
    ["query"]=>
    string(31) "SELECT id FROM test WHERE id =?"
    ["occurences"]=>
    int(2)
```

```
    ["eligible_for_caching"]=>
    bool(true)
    ["avg_run_time"]=>
    int(159)
    ["min_run_time"]=>
    int(12)
    ["max_run_time"]=>
    int(307)
    ["avg_store_time"]=>
    int(10)
    ["min_store_time"]=>
    int(8)
    ["max_store_time"]=>
    int(13)
  }
}
```

See Also

Runtime configuration
mysqlnd_qc.collect_normalized_query_trace
mysqlnd_qc.time_statistics
mysqlnd_qc_get_query_trace_log

8.7.6 mysqlnd_qc_get_query_trace_log

- mysqlnd_qc_get_query_trace_log

  Returns a backtrace for each query inspected by the query cache

Description

```
  array mysqlnd_qc_get_query_trace_log();
```

Returns a backtrace for each query inspected by the query cache. The collection of the backtrace is disabled by default. To collect the backtrace you have to set the PHP configuration directive mysqlnd_qc.collect_query_trace to 1

The maximum depth of the backtrace is limited to the depth set with the PHP configuration directive mysqlnd_qc.query_trace_bt_depth.

Parameters

This function has no parameters.

Return Values

An array of query backtrace. Every list entry contains the query string, a backtrace and further detail information.

Key	Description
query	Query string.
origin	Code backtrace.
run_time	Query run time in milliseconds. The collection of all times and the necessary gettimeofday system calls can be disabled by setting the PHP configuration directive mysqlnd_qc.time_statistics to 0

Key	Description
store_time	Query result set store time in milliseconds. The collection of all times and the necessary gettimeofday system calls can be disabled by setting the PHP configuration directive mysqlnd_qc.time_statistics to 0
eligible_for_caching	TRUE if query is cacheable otherwise FALSE.
no_table	TRUE if the query has generated a result set and at least one column from the result set has no table name set in its metadata. This is usually the case with queries which one probably do not want to cache such as SELECT SLEEP(1). By default any such query will not be added to the cache. See also PHP configuration directive mysqlnd_qc.cache_no_table.
was_added	TRUE if the query result has been put into the cache, otherwise FALSE.
was_already_in_cache	TRUE if the query result would have been added to the cache if it was not already in the cache (cache hit). Otherwise FALSE.

Examples

Example 8.22 mysqlnd_qc_get_query_trace_log example

```
mysqlnd_qc.collect_query_trace=1
```

```
<?php
/* Connect, create and populate test table */
$mysqli = new mysqli("host", "user", "password", "schema", "port", "socket");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2)");

/* not cached */
$res = $mysqli->query("SELECT id FROM test WHERE id = 1");
var_dump($res->fetch_assoc());
$res->free();

/* cache put */
$res = $mysqli->query("/*" . MYSQLND_QC_ENABLE_SWITCH . "*/" . "SELECT id FROM test WHERE id = 2");
var_dump($res->fetch_assoc());
$res->free();

/* cache hit */
$res = $mysqli->query("/*" . MYSQLND_QC_ENABLE_SWITCH . "*/" . "SELECT id FROM test WHERE id = 2");
var_dump($res->fetch_assoc());
$res->free();

var_dump(mysqlnd_qc_get_query_trace_log());
?>
```

The above examples will output:

```
array(1) {
  ["id"]=>
  string(1) "1"
}
array(1) {
  ["id"]=>
```

```
  string(1) "2"
}
array(1) {
  ["id"]=>
  string(1) "2"
}
array(6) {
  [0]=>
  array(8) {
    ["query"]=>
    string(25) "DROP TABLE IF EXISTS test"
    ["origin"]=>
    string(102) "#0 qc.php(4): mysqli->query('DROP TABLE IF E...')
#1 {main}"
    ["run_time"]=>
    int(0)
    ["store_time"]=>
    int(0)
    ["eligible_for_caching"]=>
    bool(false)
    ["no_table"]=>
    bool(false)
    ["was_added"]=>
    bool(false)
    ["was_already_in_cache"]=>
    bool(false)
  }
  [1]=>
  array(8) {
    ["query"]=>
    string(25) "CREATE TABLE test(id INT)"
    ["origin"]=>
    string(102) "#0 qc.php(5): mysqli->query('CREATE TABLE te...')
#1 {main}"
    ["run_time"]=>
    int(0)
    ["store_time"]=>
    int(0)
    ["eligible_for_caching"]=>
    bool(false)
    ["no_table"]=>
    bool(false)
    ["was_added"]=>
    bool(false)
    ["was_already_in_cache"]=>
    bool(false)
  }
  [2]=>
  array(8) {
    ["query"]=>
    string(36) "INSERT INTO test(id) VALUES (1), (2)"
    ["origin"]=>
    string(102) "#0 qc.php(6): mysqli->query('INSERT INTO tes...')
#1 {main}"
    ["run_time"]=>
    int(0)
    ["store_time"]=>
    int(0)
    ["eligible_for_caching"]=>
    bool(false)
    ["no_table"]=>
    bool(false)
    ["was_added"]=>
    bool(false)
    ["was_already_in_cache"]=>
    bool(false)
  }
```

```
  [3]=>
  array(8) {
    ["query"]=>
    string(32) "SELECT id FROM test WHERE id = 1"
    ["origin"]=>
    string(102) "#0 qc.php(9): mysqli->query('SELECT id FROM ...')
#1 {main}"
    ["run_time"]=>
    int(0)
    ["store_time"]=>
    int(25)
    ["eligible_for_caching"]=>
    bool(false)
    ["no_table"]=>
    bool(false)
    ["was_added"]=>
    bool(false)
    ["was_already_in_cache"]=>
    bool(false)
  }
  [4]=>
  array(8) {
    ["query"]=>
    string(41) "/*qc=on*/SELECT id FROM test WHERE id = 2"
    ["origin"]=>
    string(103) "#0 qc.php(14): mysqli->query('/*qc=on*/SELECT...')
#1 {main}"
    ["run_time"]=>
    int(311)
    ["store_time"]=>
    int(13)
    ["eligible_for_caching"]=>
    bool(true)
    ["no_table"]=>
    bool(false)
    ["was_added"]=>
    bool(true)
    ["was_already_in_cache"]=>
    bool(false)
  }
  [5]=>
  array(8) {
    ["query"]=>
    string(41) "/*qc=on*/SELECT id FROM test WHERE id = 2"
    ["origin"]=>
    string(103) "#0 qc.php(19): mysqli->query('/*qc=on*/SELECT...')
#1 {main}"
    ["run_time"]=>
    int(13)
    ["store_time"]=>
    int(8)
    ["eligible_for_caching"]=>
    bool(true)
    ["no_table"]=>
    bool(false)
    ["was_added"]=>
    bool(false)
    ["was_already_in_cache"]=>
    bool(true)
  }
}
```

See Also

Runtime configuration

8.7.7 mysqlnd_qc_set_cache_condition

- mysqlnd_qc_set_cache_condition

  Set conditions for automatic caching

Description

```
bool mysqlnd_qc_set_cache_condition(
  int condition_type,
  mixed condition,
  mixed condition_option);
```

Sets a condition for automatic caching of statements which do not contain the necessary SQL hints to enable caching of them.

Parameters

condition_type	Type of the condition. The only allowed value is MYSQLND_QC_CONDITION_META_SCHEMA_PATTERN.
condition	Parameter for the condition set with condition_type. Parameter type and structure depend on condition_type If condition_type equals MYSQLND_QC_CONDITION_META_SCHEMA_PATTERN condition must be a string. The string sets a pattern. Statements are cached if table and database meta data entry of their result sets match the pattern. The pattern is checked for a match with the db and org_table meta data entries provided by the underlying MySQL client server library. Please, check the MySQL Reference manual for details about the two entries. The db and org_table values are concatenated with a dot (.) before matched against condition. Pattern matching supports the wildcards % and _. The wildcard % will match one or many arbitrary characters. _ will match one arbitrary character. The escape symbol is backslash.
condition_option	Option for condition. Type and structure depend on condition_type. If condition_type equals MYSQLND_QC_CONDITION_META_SCHEMA_PATTERN condition_options is the TTL to be used.

Examples

Example 8.23 mysqlnd_qc_set_cache_condition example

```
<?php
/* Cache all accesses to tables with the name "new%" in schema/database "db_example" for 1 second */
if (!mysqlnd_qc_set_cache_condition(MYSQLND_QC_CONDITION_META_SCHEMA_PATTERN, "db_example.new%", 1)) {
  die("Failed to set cache condition!");
}

$mysqli = new mysqli("host", "user", "password", "db_example", "port");
/* cached although no SQL hint given  */
$mysqli->query("SELECT id, title FROM news");

$pdo_mysql = new PDO("mysql:host=host;dbname=db_example;port=port", "user", "password");
/* not cached: no SQL hint, no pattern match */
$pdo_mysql->query("SELECT id, title FROM latest_news");
/* cached: TTL 1 second, pattern match */
$pdo_mysql->query("SELECT id, title FROM news");
?>
```

Return Values

Returns TRUE on success or FALSE on FAILURE.

See Also

Quickstart: pattern based caching

8.7.8 mysqlnd_qc_set_is_select

- `mysqlnd_qc_set_is_select`

  Installs a callback which decides whether a statement is cached

Description

```
  mixed mysqlnd_qc_set_is_select(
    string callback);
```

Installs a callback which decides whether a statement is cached.

There are several ways of hinting PELC/mysqlnd_qc to cache a query. By default, PECL/mysqlnd_qc attempts to cache a if caching of all statements is enabled or the query string begins with a certain SQL hint. The plugin internally calls a function named `is_select()` to find out. This internal function can be replaced with a user-defined callback. Then, the user-defined callback is responsible to decide whether the plugin attempts to cache a statement. Because the internal function is replaced with the callback, the callback gains full control. The callback is free to ignore the configuration setting `mysqlnd_qc.cache_by_default` and SQL hints.

The callback is invoked for every statement inspected by the plugin. It is given the statements string as a parameter. The callback returns `FALSE` if the statement shall not be cached. It returns `TRUE` to make the plugin attempt to cache the statements result set, if any. A so-created cache entry is given the default TTL set with the PHP configuration directive `mysqlnd_qc.ttl`. If a different TTL shall be used, the callback returns a numeric value to be used as the TTL.

The internal `is_select` function is part of the internal cache storage handler interface. Thus, a user-defined storage handler offers the same capabilities.

Parameters

This function has no parameters.

Return Values

Returns TRUE on success or FALSE on failure.

Examples

Example 8.24 mysqlnd_qc_set_is_select example

```
<?php
/* callback which decides if query is cached */
function is_select($query) {
  static $patterns = array(
   /* true - use default from mysqlnd_qc.ttl */
   "@SELECT\s+.*\s+FROM\s+test@ismU" => true,
   /* 3 - use TTL = 3 seconds */
   "@SELECT\s+.*\s+FROM\s+news@ismU" => 3
  );
  /* check if query does match pattern */
  foreach ($patterns as $pattern => $ttl) {
    if (preg_match($pattern, $query)) {
      printf("is_select(%45s): cache\n", $query);
      return $ttl;
    }
  }
  printf("is_select(%45s): do not cache\n", $query);
  return false;
}
mysqlnd_qc_set_is_select("is_select");

/* Connect, create and populate test table */
$mysqli = new mysqli("host", "user", "password", "schema");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)");

/* cache put */
$mysqli->query("SELECT id FROM test WHERE id = 1");
/* cache hit */
$mysqli->query("SELECT id FROM test WHERE id = 1");
/* cache put */
$mysqli->query("SELECT * FROM test");
?>
```

The above examples will output:

```
is_select(                    DROP TABLE IF EXISTS test): do not cache
is_select(                    CREATE TABLE test(id INT)): do not cache
is_select(    INSERT INTO test(id) VALUES (1), (2), (3)): do not cache
is_select(              SELECT id FROM test WHERE id = 1): cache
is_select(              SELECT id FROM test WHERE id = 1): cache
is_select(                           SELECT * FROM test): cache
```

See Also

Runtime configuration

mysqlnd_qc.ttl
mysqlnd_qc.cache_by_default
mysqlnd_qc_set_user_handlers

8.7.9 mysqlnd_qc_set_storage_handler

- mysqlnd_qc_set_storage_handler

  Change current storage handler

Description

```
bool mysqlnd_qc_set_storage_handler(
  string handler);
```

Sets the storage handler used by the query cache. A list of available storage handler can be obtained from mysqlnd_qc_get_available_handlers. Which storage are available depends on the compile time configuration of the query cache plugin. The default storage handler is always available. All other storage handler must be enabled explicitly when building the extension.

Parameters

handler	Handler can be of type string representing the name of a built-in storage handler or an object of type mysqlnd_qc_handler_default. The names of the built-in storage handler are default, APC, MEMCACHE, sqlite.

Return Values

Returns TRUE on success or FALSE on failure.

If changing the storage handler fails a catchable fatal error will be thrown. The query cache cannot operate if the previous storage handler has been shutdown but no new storage handler has been installed.

Examples

Example 8.25 mysqlnd_qc_set_storage_handler example

The example shows the output from the built-in default storage handler. Other storage handler may report different data.

```
<?php
var_dump(mysqlnd_qc_set_storage_handler("memcache"));

if (true === mysqlnd_qc_set_storage_handler("default"))
  printf("Default storage handler activated");

/* Catchable fatal error */
var_dump(mysqlnd_qc_set_storage_handler("unknown"));
?>
```

The above examples will output:

```
bool(true)
Default storage handler activated
Catchable fatal error: mysqlnd_qc_set_storage_handler(): Unknown handler 'unknown' in (file) on line (line)
```

See Also

Installation
mysqlnd_qc_get_available_handlers

8.7.10 mysqlnd_qc_set_user_handlers

- mysqlnd_qc_set_user_handlers

  Sets the callback functions for a user-defined procedural storage handler

Description

```
bool mysqlnd_qc_set_user_handlers(
  string get_hash,
  string find_query_in_cache,
  string return_to_cache,
  string add_query_to_cache_if_not_exists,
  string query_is_select,
  string update_query_run_time_stats,
  string get_stats,
  string clear_cache);
```

Sets the callback functions for a user-defined procedural storage handler.

Parameters

get_hash	Name of the user function implementing the storage handler get_hash functionality.
find_query_in_cache	Name of the user function implementing the storage handler find_in_cache functionality.
return_to_cache	Name of the user function implementing the storage handler return_to_cache functionality.
add_query_to_cache_if_not_exists	Name of the user function implementing the storage handler add_query_to_cache_if_not_exists functionality.
query_is_select	Name of the user function implementing the storage handler query_is_select functionality.
update_query_run_time_stats	Name of the user function implementing the storage handler update_query_run_time_stats functionality.
get_stats	Name of the user function implementing the storage handler get_stats functionality.
clear_cache	Name of the user function implementing the storage handler clear_cache functionality.

Return Values

Returns TRUE on success or FALSE on FAILURE.

See Also

Procedural user-defined storage handler example

8.8 Change History

This change history is a high level summary of selected changes that may impact applications and/or break backwards compatibility.

See also the `CHANGES` file in the source distribution for a complete list of changes.

8.8.1 PECL/mysqlnd_qc 1.2 series

1.2.0 - alpha

- Release date: 03/2013
- Motto/theme: PHP 5.5 compatibility

Feature changes

- Update build for PHP 5.5 (Credits: Remi Collet)
- APC storage handler update
  - Fix build for APC 3.1.13-beta and trunk
- Introduced `MYSQLND_QC_VERSION` and `MYSQLND_QC_VERSION_ID`.

8.8.2 PECL/mysqlnd_qc 1.1 series

1.1.0 - stable

- Release date: 04/2012
- Motto/theme: PHP 5.4 compatibility, schema pattern based caching and mysqlnd_ms support

1.1.0 - beta

- Release date: 04/2012
- Motto/theme: PHP 5.4 compatibility, schema pattern based caching and mysqlnd_ms support

1.1.0 - alpha

- Release date: 04/2012

- Motto/theme: PHP 5.4 compatibility, schema pattern based caching and mysqlnd_ms support

Feature changes

- APC storage handler update
  - Fix build for APC 3.1.9+
  - Note: Use of the APC storage handler is currently not recommended due to stability issues of APC itself.
- New PHP configuration directives
  - `mysqlnd_qc.collect_statistics_log_file`. Aggregated cache statistics log file written after every 10th request served by the PHP process.
  - `mysqlnd_qc.ignore_sql_comments`. Control whether SQL comments are ignored for cache key hash generation.
- New constants and SQL hints
  - `MYSQLND_QC_SERVER_ID_SWITCH` allows grouping of cache entries from different physical connections. This is needed by PECL/mysqlnd_ms.
  - `MYSQLND_QC_CONDITION_META_SCHEMA_PATTERN` to be used with `mysqlnd_qc_set_cache_condition`.
- New function `mysqlnd_qc_set_cache_condition` for built-in schema pattern based caching. Likely to support a wider range of conditions in the future.
- Report `valid_until` timestamp for cache entries of the default handler through `mysqlnd_qc_get_cache_info`.
- Include charset number for cache entry hashing. This should prevent serving result sets which have the wrong charset.

  API change: get_hash_key expects new "charsetnr" (int) parameter after "port".
- API change: changing is_select() signature from bool is_select() to mixed is_select(). Mixed can be either boolean or array(long ttl, string server_id). This is needed by PECL/mysqlnd_ms.

Other

- Support acting as a cache backend for PECL/mysqlnd_ms 1.3.0-beta or later to transparently replace MySQL Replication slave reads with cache accesses, if the user explicitly allows.

Bug fixes

- Fixed Bug #59959 (config.m4, wrong library - 64bit memcached handler builds) (Credits: Remi Collet)

8.8.3 PECL/mysqlnd_qc 1.0 series

1.0.1-stable

- Release date: 12/2010

- Motto/theme: Prepared statement support

Added support for Prepared statements and unbuffered queries.

1.0.0-beta

- Release date: 07/2010
- Motto/theme: TTL-based cache with various storage options (Memcache, APC, SQLite, user-defined)

Initial public release of the transparent TTL-based query result cache. Flexible storage of cached results. Various storage media supported.

Chapter 9 Mysqlnd user handler plugin

Table of Contents

The mysqlnd user handler plugin (mysqlnd_uh) allows users to set hooks for most internal calls of the MySQL native driver for PHP (mysqlnd). Mysqlnd and its plugins, including PECL/mysqlnd_uh, operate on a layer beneath the PHP MySQL extensions. A mysqlnd plugin can be considered as a proxy between the PHP MySQL extensions and the MySQL server as part of the PHP executable on the client-side. Because the plugins operates on their own layer below the PHP MySQL extensions, they can monitor and change application actions without requiring application changes. If the PHP MySQL extensions (mysqli, mysql, PDO_MYSQL) are compiled to use mysqlnd this can be used for:

- Monitoring
  - Queries executed by any of the PHP MySQL extensions
  - Prepared statements executing by any of the PHP MySQL extensions
- Auditing
  - Detection of database usage
  - SQL injection protection using black and white lists
- Assorted
  - Load Balancing connections

The MySQL native driver for PHP (mysqlnd) features an internal plugin C API. C plugins, such as the mysqlnd user handler plugin, can extend the functionality of mysqlnd. PECL/mysqlnd_uh makes parts of the internal plugin C API available to the PHP user for plugin development with PHP.

> **Status**
>
> The mysqlnd user handler plugin is in alpha status. Take appropriate care before using it in production environments.

9.1 Security considerations

PECL/mysqlnd_uh gives users access to MySQL user names, MySQL password used by any of the PHP MySQL extensions to connect to MySQL. It allows monitoring of all queries and prepared statements exposing the statement string to the user. Therefore, the extension should be installed with care. The `PHP_INI_SYSTEM` configuration setting `mysqlnd_uh.enable` can be used to prevent users from hooking mysqlnd calls.

Code obfuscators and similar technologies are not suitable to prevent monitoring of mysqlnd library activities if PECL/mysqlnd_uh is made available and the user can install a proxy, for example, using auto_prepend_file.

9.2 Documentation note

Many of the mysqlnd_uh functions are briefly described because the mysqli extension is a thin abstraction layer on top of the MySQL C API that the `mysqlnd` library provides. Therefore, the corresponding mysqli documentation (along with the MySQL reference manual) can be consulted to receive more information about a particular function.

9.3 On the name

The shortcut `mysqlnd_uh` stands for `mysqlnd user handler`, and has been the name since early development.

9.4 Quickstart and Examples

The mysqlnd user handler plugin can be understood as a client-side proxy for all PHP MySQL extensions (mysqli, mysql, PDO_MYSQL), if they are compiled to use the mysqlnd library. The extensions use the `mysqlnd` library internally, at the C level, to communicate with the MySQL server. PECL/mysqlnd_uh allows it to hook many `mysqlnd` calls. Therefore, most activities of the PHP MySQL extensions can be monitored.

Because monitoring happens at the level of the library, at a layer below the application, it is possible to monitor applications without changing them.

On the C level, the `mysqlnd` library is structured in modules or classes. The extension hooks almost all methods of the `mysqlnd` internal `connection` class and exposes them through the user space class `MysqlndUhConnection`. Some few methods of the mysqlnd internal `statement` class are made available to the PHP user with the class `MysqlndUhPreparedStatement`. By subclassing the classes

`MysqlndUhConnection` and `MysqlndUhPreparedStatement` users get access to `mysqlnd` internal function calls.

> **Note**
>
> The internal `mysqlnd` function calls are not designed to be exposed to the PHP user. Manipulating their activities may cause PHP to crash or leak memory. Often, this is not considered a bug. Please, keep in mind that you are accessing C library functions through PHP which are expected to take certain actions, which you may not be able to emulate in user space. Therefore, it is strongly recommended to always call the parent method implementation when subclassing `MysqlndUhConnection` or `MysqlndUhPreparedStatement`. To prevent the worst case, the extension performs some sanity checks. Please, see also the Mysqlnd_uh Configure Options.

9.4.1 Setup

The plugin is implemented as a PHP extension. See the installation instructions to install the PECL/mysqlnd_uh extension. Then, load the extension into PHP and activate the plugin in the PHP configuration file using the PHP configuration directive named mysqlnd_uh.enable. The below example shows the default settings of the extension.

Example 9.1 Enabling the plugin (php.ini)

```
mysqlnd_uh.enable=1
mysqlnd_uh.report_wrong_types=1
```

9.4.2 How it works

This describes the background and inner workings of the mysqlnd_uh extension.

Two classes are provided by the extension: `MysqlndUhConnection` and `MysqlndUhPreparedStatement`. `MysqlndUhConnection` lets you access almost all methods of the `mysqlnd` internal `connection` class. The latter exposes some selected methods of the `mysqlnd` internal `statement` class. For example, `MysqlndUhConnection::connect` maps to the `mysqlnd` library C function `mysqlnd_conn__connect`.

As a mysqlnd plugin, the PECL/mysqlnd_uh extension replaces `mysqlnd` library C functions with its own functions. Whenever a PHP MySQL extension compiled to use `mysqlnd` calls a mysqlnd function, the functions installed by the plugin are executed instead of the original `mysqlnd` ones. For example, `mysqli_connect` invokes `mysqlnd_conn__connect`, so the connect function installed by PECL/mysqlnd_uh will be called. The functions installed by PECL/mysqlnd_uh are the methods of the built-in classes.

The built-in PHP classes and their methods do nothing but call their `mysqlnd` C library counterparts, to behave exactly like the original `mysqlnd` function they replace. The code below illustrates in pseudo-code what the extension does.

Example 9.2 Pseudo-code: what a built-in class does

```
class MysqlndUhConnection {
  public function connect(($conn, $host, $user, $passwd, $db, $port, $socket, $mysql_flags) {
    MYSQLND* c_mysqlnd_connection = convert_from_php_to_c($conn);
    ...
    return call_c_function(mysqlnd_conn__connect(c_mysqlnd_connection, ...));
  }
}
```

The build-in classes behave like a transparent proxy. It is possible for you to replace the proxy with your own. This is done by subclassing `MysqlndUhConnection` or `MysqlndUhPreparedStatement` to extend the functionality of the proxy, followed by registering a new proxy object. Proxy objects are installed by `mysqlnd_uh_set_connection_proxy` and `mysqlnd_uh_set_statement_proxy`.

Example 9.3 Installing a proxy

```
<?php
class proxy extends MysqlndUhConnection {
 public function connect($res, $host, $user, $passwd, $db, $port, $socket, $mysql_flags) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::connect($res, $host, $user, $passwd, $db, $port, $socket, $mysql_flags);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
?>
```

The above example will output:

```
proxy::connect(array (
  0 => NULL,
  1 => 'localhost',
  2 => 'root',
  3 => '',
  4 => 'test',
  5 => 3306,
  6 => NULL,
  7 => 131072,
))
proxy::connect returns true
```

9.4.3 Installing a proxy

The extension provides two built-in classes: `MysqlndUhConnection` and `MysqlndUhPreparedStatement`. The classes are used for hooking `mysqlnd` library calls. Their methods correspond to `mysqlnd` internal functions. By default they act like a transparent proxy and do nothing but call their `mysqlnd` counterparts. By subclassing the classes you can install your own proxy to monitor `mysqlnd`.

See also the How it works guide to learn about the inner workings of this extension.

Connection proxies are objects of the type `MysqlndUhConnection`. Connection proxy objects are installed by `mysqlnd_uh_set_connection_proxy`. If you install the built-in class `MysqlndUhConnection` as a proxy, nothing happens. It behaves like a transparent proxy.

Example 9.4 Proxy registration, mysqlnd_uh.enable=1

```
<?php
mysqlnd_uh_set_connection_proxy(new MysqlndUhConnection());
$mysqli = new mysqli("localhost", "root", "", "test");
?>
```

The `PHP_INI_SYSTEM` configuration setting `mysqlnd_uh.enable` controls whether a proxy may be set. If disabled, the extension will throw errors of type `E_WARNING`

Example 9.5 Proxy installation disabled

```
mysqlnd_uh.enable=0
```

```
<?php
mysqlnd_uh_set_connection_proxy(new MysqlndUhConnection());
$mysqli = new mysqli("localhost", "root", "", "test");
?>
```

The above example will output:

```
PHP Warning:  MysqlndUhConnection::__construct(): (Mysqlnd User Handler) The plugin has been disabled by setti
PHP Warning:  mysqlnd_uh_set_connection_proxy(): (Mysqlnd User Handler) The plugin has been disabled by settin
```

To monitor `mysqlnd`, you have to write your own proxy object subclassing `MysqlndUhConnection`. Please, see the function reference for a the list of methods that can be subclassed. Alternatively, you can use reflection to inspect the built-in `MysqlndUhConnection`.

Create a new class `proxy`. Derive it from the built-in class `MysqlndUhConnection`. Replace the `MysqlndUhConnection::connect`. method. Print out the host parameter value passed to the method. Make sure that you call the parent implementation of the `connect` method. Failing to do so may give unexpected and undesired results, including memory leaks and crashes.

Register your proxy and open three connections using the PHP MySQL extensions mysqli, mysql, PDO_MYSQL. If the extensions have been compiled to use the `mysqlnd` library, the `proxy::connect` method will be called three times, once for each connection opened.

Example 9.6 Connection proxy

```
<?php
class proxy extends MysqlndUhConnection {
  public function connect($res, $host, $user, $passwd, $db, $port, $socket, $mysql_flags) {
   printf("Connection opened to '%s'\n", $host);
   /* Always call the parent implementation! */
   return parent::connect($res, $host, $user, $passwd, $db, $port, $socket, $mysql_flags);
  }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysql = mysql_connect("localhost", "root", "");
$pdo = new PDO("mysql:host=localhost;dbname=test", "root", "");
?>
```

The above example will output:

```
Connection opened to 'localhost'
Connection opened to 'localhost'
Connection opened to 'localhost'
```

The use of prepared statement proxies follows the same pattern: create a proxy object of the type `MysqlndUhPreparedStatement` and install the proxy using `mysqlnd_uh_set_statement_proxy`.

Example 9.7 Prepared statement proxy

```
<?php
class stmt_proxy extends MysqlndUhPreparedStatement {
 public function prepare($res, $query) {
  printf("%s(%s)\n", __METHOD__, $query);
  return parent::prepare($res, $query);
 }
}
mysqlnd_uh_set_statement_proxy(new stmt_proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$stmt = $mysqli->prepare("SELECT 'mysqlnd hacking made easy' AS _msg FROM DUAL");
?>
```

The above example will output:

```
stmt_proxy::prepare(SELECT 'mysqlnd hacking made easy' AS _msg FROM DUAL)
```

9.4.4 Basic query monitoring

Basic monitoring of a query statement is easy with PECL/mysqlnd_uh. Combined with `debug_print_backtrace` it can become a powerful tool, for example, to find the origin of certain statement. This may be desired when searching for slow queries but also after database refactoring to

find code still accessing deprecated databases or tables. The latter may be a complicated matter to do otherwise, especially if the application uses auto-generated queries.

Example 9.8 Basic Monitoring

```
<?php
class conn_proxy extends MysqlndUhConnection {
 public function query($res, $query) {
  debug_print_backtrace();
  return parent::query($res, $query);
 }
}
class stmt_proxy extends MysqlndUhPreparedStatement {
 public function prepare($res, $query) {
  debug_print_backtrace();
  return parent::prepare($res, $query);
 }
}
mysqlnd_uh_set_connection_proxy(new conn_proxy());
mysqlnd_uh_set_statement_proxy(new stmt_proxy());

printf("Proxies installed...\n");
$pdo = new PDO("mysql:host=localhost;dbname=test", "root", "");
var_dump($pdo->query("SELECT 1 AS _one FROM DUAL")->fetchAll(PDO::FETCH_ASSOC));

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->prepare("SELECT 1 AS _two FROM DUAL");
?>
```

The above example will output:

```
#0  conn_proxy->query(Resource id #19, SELECT 1 AS _one FROM DUAL)
#1  PDO->query(SELECT 1 AS _one FROM DUAL) called at [example.php:19]
array(1) {
  [0]=>
  array(1) {
    ["_one"]=>
    string(1) "1"
  }
}
#0  stmt_proxy->prepare(Resource id #753, SELECT 1 AS _two FROM DUAL)
#1  mysqli->prepare(SELECT 1 AS _two FROM DUAL) called at [example.php:22]
```

For basic query monitoring you should install a connection and a prepared statement proxy. The connection proxy should subclass `MysqlndUhConnection::query`. All database queries not using native prepared statements will call this method. In the example the `query` function is invoked by a PDO call. By default, `PDO_MySQL` is using prepared statement emulation.

All native prepared statements are prepared with the `prepare` method of `mysqlnd` exported through `MysqlndUhPreparedStatement::prepare`. Subclass `MysqlndUhPreparedStatement` and overwrite `prepare` for native prepared statement monitoring.

9.5 Installing/Configuring

9.5.1 Requirements

PHP 5.3.3 or later. It is recommended to use PHP 5.4.0 or later to get access to the latest mysqlnd features.

The mysqlnd_uh user handler plugin supports all PHP applications and all available PHP MySQL extensions (mysqli, mysql, PDO_MYSQL). The PHP MySQL extension must be configured to use mysqlnd in order to be able to use the mysqlnd_uh plugin for mysqlnd.

The alpha versions makes use of some mysqli features. You must enable mysqli to compile the plugin. This requirement may be removed in the future. Note, that this requirement does not restrict you to use the plugin only with mysqli. You can use the plugin to monitor mysql, mysqli and PDO_MYSQL.

9.5.2 Installation

Information for installing this PECL extension may be found in the manual chapter titled Installation of PECL extensions. Additional information such as new releases, downloads, source files, maintainer information, and a CHANGELOG, can be located here: http://pecl.php.net/package/mysqlnd-uh

PECL/mysqlnd_uh is currently not available on Windows. The source code of the extension makes use of C99 constructs not allowed with PHP Windows builds.

9.5.3 Runtime Configuration

The behaviour of these functions is affected by settings in php.ini.

Table 9.1 Mysqlnd_uh Configure Options

Name	Default	Changeable	Changelog
mysqlnd_uh.enable	1	PHP_INI_SYSTEM	
mysqlnd_uh.report_wrong_types	1	PHP_INI_ALL	

Here's a short explanation of the configuration directives.

mysqlnd_uh.enable integer
Enables or disables the plugin. If set to disabled, the extension will not allow users to plug into mysqlnd to hook mysqlnd calls.

mysqlnd_uh.report_wrong_types integer
Whether to report wrong return value types of user hooks as E_WARNING level errors. This is recommended for detecting errors.

9.5.4 Resource Types

This extension has no resource types defined.

9.6 Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

Most of the constants refer to details of the MySQL Client Server Protocol. Please, refer to the MySQL reference manual to learn about their meaning. To avoid content duplication, only short descriptions are given.

`MysqlndUhConnection::simpleCommand` related

The following constants can be used to detect what command is to be send through `MysqlndUhConnection::simpleCommand`.

`MYSQLND_UH_MYSQLND_COM_SLEEP` MySQL Client Server protocol command: COM_SLEEP. (integer)

`MYSQLND_UH_MYSQLND_COM_QUIT` MySQL Client Server protocol command: COM_QUIT. (integer)

`MYSQLND_UH_MYSQLND_COM_INIT_DB` MySQL Client Server protocol command: COM_INIT_DB. (integer)

`MYSQLND_UH_MYSQLND_COM_QUERY` MySQL Client Server protocol command: COM_QUERY. (integer)

`MYSQLND_UH_MYSQLND_COM_FIELD_LIST` MySQL Client Server protocol command: COM_FIELD_LIST. (integer)

`MYSQLND_UH_MYSQLND_COM_CREATE_DB` MySQL Client Server protocol command: COM_CREATE_DB. (integer)

`MYSQLND_UH_MYSQLND_COM_DROP_DB` MySQL Client Server protocol command: COM_DROP_DB. (integer)

`MYSQLND_UH_MYSQLND_COM_REFRESH` MySQL Client Server protocol command: COM_REFRESH. (integer)

`MYSQLND_UH_MYSQLND_COM_SHUTDOWN` MySQL Client Server protocol command: COM_SHUTDOWN. (integer)

`MYSQLND_UH_MYSQLND_COM_STATISTICS` MySQL Client Server protocol command: COM_STATISTICS. (integer)

`MYSQLND_UH_MYSQLND_COM_PROCESS_INFO` MySQL Client Server protocol command: COM_PROCESS_INFO. (integer)

`MYSQLND_UH_MYSQLND_COM_CONNECT` MySQL Client Server protocol command: COM_CONNECT. (integer)

`MYSQLND_UH_MYSQLND_COM_PROCESS_KILL` MySQL Client Server protocol command: COM_PROCESS_KILL. (integer)

`MYSQLND_UH_MYSQLND_COM_DEBUG` MySQL Client Server protocol command: COM_DEBUG. (integer)

`MYSQLND_UH_MYSQLND_COM_PING` MySQL Client Server protocol command: COM_PING. (integer)

`MYSQLND_UH_MYSQLND_COM_TIME` MySQL Client Server protocol command: COM_TIME. (integer)

MYSQLND_UH_MYSQLND_COM_DELAYED_INSERT (integer) — MySQL Client Server protocol command: COM_DELAYED_INSERT.

MYSQLND_UH_MYSQLND_COM_CHANGE_USER (integer) — MySQL Client Server protocol command: COM_CHANGE_USER.

MYSQLND_UH_MYSQLND_COM_BINLOG_DUMP (integer) — MySQL Client Server protocol command: COM_BINLOG_DUMP.

MYSQLND_UH_MYSQLND_COM_TABLE_DUMP (integer) — MySQL Client Server protocol command: COM_TABLE_DUMP.

MYSQLND_UH_MYSQLND_COM_CONNECT_OUT (integer) — MySQL Client Server protocol command: COM_CONNECT_OUT.

MYSQLND_UH_MYSQLND_COM_REGISTER_SLAVE (integer) — MySQL Client Server protocol command: COM_REGISTER_SLAVED.

MYSQLND_UH_MYSQLND_COM_STMT_PREPARE (integer) — MySQL Client Server protocol command: COM_STMT_PREPARE.

MYSQLND_UH_MYSQLND_COM_STMT_EXECUTE (integer) — MySQL Client Server protocol command: COM_STMT_EXECUTE.

MYSQLND_UH_MYSQLND_COM_STMT_SEND_LONG_DATA (integer) — MySQL Client Server protocol command: COM_STMT_SEND_LONG_DATA.

MYSQLND_UH_MYSQLND_COM_STMT_CLOSE (integer) — MySQL Client Server protocol command: COM_STMT_CLOSE.

MYSQLND_UH_MYSQLND_COM_STMT_RESET (integer) — MySQL Client Server protocol command: COM_STMT_RESET.

MYSQLND_UH_MYSQLND_COM_SET_OPTION (integer) — MySQL Client Server protocol command: COM_SET_OPTION.

MYSQLND_UH_MYSQLND_COM_STMT_FETCH (integer) — MySQL Client Server protocol command: COM_STMT_FETCH.

MYSQLND_UH_MYSQLND_COM_DAEMON (integer) — MySQL Client Server protocol command: COM_DAEMON.

MYSQLND_UH_MYSQLND_COM_END (integer) — MySQL Client Server protocol command: COM_END.

The following constants can be used to analyze the `ok_packet` argument of `MysqlndUhConnection::simpleCommand`.

MYSQLND_UH_MYSQLND_PROT_GREET_PACKET (integer) — MySQL Client Server protocol packet: greeting.

MYSQLND_UH_MYSQLND_PROT_AUTH_PACKET (integer) — MySQL Client Server protocol packet: authentication.

MYSQLND_UH_MYSQLND_PROT_OK_PACKET (integer) — MySQL Client Server protocol packet: OK.

MYSQLND_UH_MYSQLND_PROT_EOF_PACKET (integer) — MySQL Client Server protocol packet: EOF.

MYSQLND_UH_MYSQLND_PROT_CMD_PACKET MySQL Client Server protocol packet: command.
(integer)

MYSQLND_UH_MYSQLND_PROT_RSET_HEADER_PACKET MySQL Client Server protocol packet: result set header.
(integer)

MYSQLND_UH_MYSQLND_PROT_RSET_FLD_PACKET MySQL Client Server protocol packet: resultset field.
(integer)

MYSQLND_UH_MYSQLND_PROT_ROW_PACKET MySQL Client Server protocol packet: row.
(integer)

MYSQLND_UH_MYSQLND_PROT_STATS_PACKET MySQL Client Server protocol packet: stats.
(integer)

MYSQLND_UH_MYSQLND_PREPARE_RESP_PACKET MySQL Client Server protocol packet: prepare response.
(integer)

MYSQLND_UH_MYSQLND_CHG_USER_RESP_PACKET MySQL Client Server protocol packet: change user response.
(integer)

MYSQLND_UH_MYSQLND_PROT_LAST No practical meaning. Last entry marker of internal C data structure list.
(integer)

`MysqlndUhConnection::close` related

The following constants can be used to detect why a connection has been closed through `MysqlndUhConnection::close()`.

MYSQLND_UH_MYSQLND_CLOSE_EXPLICIT User has called mysqlnd to close the connection.
(integer)

MYSQLND_UH_MYSQLND_CLOSE_IMPLICIT Implicitly closed, for example, during garbage connection.
(integer)

MYSQLND_UH_MYSQLND_CLOSE_DISCONNECTED Connection error.
(integer)

MYSQLND_UH_MYSQLND_CLOSE_LAST No practical meaning. Last entry marker of internal C data structure list.
(integer)

MysqlndUhConnection::setServerOption() related

The following constants can be used to detect which option is set through `MysqlndUhConnection::setServerOption()`.

MYSQLND_UH_SERVER_OPTION_MULTI_STATEMENTS_ON Option enables multi statement support.
(integer)

MYSQLND_UH_SERVER_OPTION_MULTI_STATEMENTS_OFF Option disables multi statement support.
(integer)

`MysqlndUhConnection::setClientOption` related

The following constants can be used to detect which option is set through `MysqlndUhConnection::setClientOption`.

MYSQLND_UH_MYSQLND_OPTION_CONNECT_TIMEOUT Option connection timeout.
(integer)

`MYSQLND_UH_MYSQLND_OPTION_OPT_COMPRESS` Option: whether the MySQL compressed protocol is to be used.
(integer)

`MYSQLND_UH_MYSQLND_OPTION_OPT_NAMED_PIPE` Option: named pipe to use for connection (Windows).
(integer)

`MYSQLND_UH_MYSQLND_OPTION_INIT_COMMAND` Option: init command to execute upon connect.
(integer)

`MYSQLND_UH_MYSQLND_READ_DEFAULT_FILE` Option: MySQL server default file to read upon connect.
(integer)

`MYSQLND_UH_MYSQLND_READ_DEFAULT_GROUP` Option: MySQL server default file group to read upon connect.
(integer)

`MYSQLND_UH_MYSQLND_SET_CHARSET_DIR` Option: charset description files directory.
(integer)

`MYSQLND_UH_MYSQLND_SET_CHARSET_NAME` Option: charset name.
(integer)

`MYSQLND_UH_MYSQLND_OPT_LOCAL_INFILE` Option: Whether to allow `LOAD DATA LOCAL INFILE` use.
(integer)

`MYSQLND_UH_MYSQLND_OPT_PROTOCOL` Option: supported protocol version.
(integer)

`MYSQLND_UH_MYSQLND_SHARED_MEMORY_BASE_NAME` Option: shared memory base name for shared memory connections.
(integer)

`MYSQLND_UH_MYSQLND_OPT_READ_TIMEOUT` Option: connection read timeout.
(integer)

`MYSQLND_UH_MYSQLND_OPT_WRITE_TIMEOUT` Option: connection write timeout.
(integer)

`MYSQLND_UH_MYSQLND_OPT_USE_RESULT` Option: unbuffered result sets.
(integer)

`MYSQLND_UH_MYSQLND_OPT_USE_EMBEDDED_CONNECTION` Embedded server related.
(integer)

`MYSQLND_UH_MYSQLND_OPT_GUESS_CONNECTION` Embedded server related.
(integer)

`MYSQLND_UH_MYSQLND_SET_CLIENT_IP` TODO
(integer)

`MYSQLND_UH_MYSQLND_SECURE_AUTH` TODO
(integer)

`MYSQLND_UH_MYSQLND_REPORT_DATA_TRUNCATION` Option: Whether to report data truncation.
(integer)

`MYSQLND_UH_MYSQLND_OPT_RECONNECT` Option: Whether to reconnect automatically.
(integer)

`MYSQLND_UH_MYSQLND_OPT_SSL_VERIFY_SERVER_CERT` (integer) — Option: TODO

`MYSQLND_UH_MYSQLND_OPT_NET_CMD_BUFFER_SIZE` (integer) — Option: mysqlnd network buffer size for commands.

`MYSQLND_UH_MYSQLND_OPT_NET_READ_BUFFER_SIZE` (integer) — Option: mysqlnd network buffer size for reading from the server.

`MYSQLND_UH_MYSQLND_OPT_SSL_KEY` (integer) — Option: SSL key.

`MYSQLND_UH_MYSQLND_OPT_SSL_CERT` (integer) — Option: SSL certificate.

`MYSQLND_UH_MYSQLND_OPT_SSL_CA` (integer) — Option: SSL CA.

`MYSQLND_UH_MYSQLND_OPT_SSL_CAPATH` (integer) — Option: Path to SSL CA.

`MYSQLND_UH_MYSQLND_OPT_SSL_CIPHER` (integer) — Option: SSL cipher.

`MYSQLND_UH_MYSQLND_OPT_SSL_PASSPHRASE` (integer) — Option: SSL passphrase.

`MYSQLND_UH_SERVER_OPTION_PLUGIN_DIR` (integer) — Option: server plugin directory.

`MYSQLND_UH_SERVER_OPTION_DEFAULT_AUTH` (integer) — Option: default authentication method.

`MYSQLND_UH_SERVER_OPTION_SET_CLIENT_IP` (integer) — TODO

`MYSQLND_UH_MYSQLND_OPT_MAX_ALLOWED_PACKET` (integer) — Option: maximum allowed packet size. Available as of `PHP 5.4.0`.

`MYSQLND_UH_MYSQLND_OPT_AUTH_PROTOCOL` (integer) — Option: TODO. Available as of `PHP 5.4.0`.

`MYSQLND_UH_MYSQLND_OPT_INT_AND_FLOAT_NATIVE` (integer) — Option: make mysqlnd return integer and float columns as long even when using the MySQL Client Server text protocol. Only available with a custom build of mysqlnd.

Other

The plugins version number can be obtained using `MYSQLND_UH_VERSION` or `MYSQLND_UH_VERSION_ID`. `MYSQLND_UH_VERSION` is the string representation of the numerical version number `MYSQLND_UH_VERSION_ID`, which is an integer such as 10000. Developers can calculate the version number as follows.

Version (part)	Example
Major*10000	1*10000 = 10000
Minor*100	0*100 = 0
Patch	0 = 0

Version (part)	Example
MYSQLND_UH_VERSION_ID	10000

MYSQLND_UH_VERSION (string) Plugin version string, for example, “1.0.0-alpha”.

MYSQLND_UH_VERSION_ID (integer) Plugin version number, for example, 10000.

9.7 The MysqlndUhConnection class


```
MysqlndUhConnection {
MysqlndUhConnection

     Methods

  public bool MysqlndUhConnection::changeUser(
    mysqlnd_connection connection,
    string user,
    string password,
    string database,
    bool silent,
    int passwd_len);

  public string MysqlndUhConnection::charsetName(
    mysqlnd_connection connection);

  public bool MysqlndUhConnection::close(
    mysqlnd_connection connection,
    int close_type);

  public bool MysqlndUhConnection::connect(
    mysqlnd_connection connection,
    string host,
    string use",
    string password,
    string database,
    int port,
    string socket,
    int mysql_flags);

  public MysqlndUhConnection::__construct();

  public bool MysqlndUhConnection::endPSession(
    mysqlnd_connection connection);

  public string MysqlndUhConnection::escapeString(
    mysqlnd_connection connection,
    string escape_string);

  public int MysqlndUhConnection::getAffectedRows(
    mysqlnd_connection connection);

  public int MysqlndUhConnection::getErrorNumber(
    mysqlnd_connection connection);

  public string MysqlndUhConnection::getErrorString(
    mysqlnd_connection connection);

  public int MysqlndUhConnection::getFieldCount(
    mysqlnd_connection connection);
```

```
public string MysqlndUhConnection::getHostInformation(
  mysqlnd_connection connection);

public int MysqlndUhConnection::getLastInsertId(
  mysqlnd_connection connection);

public void MysqlndUhConnection::getLastMessage(
  mysqlnd_connection connection);

public string MysqlndUhConnection::getProtocolInformation(
  mysqlnd_connection connection);

public string MysqlndUhConnection::getServerInformation(
  mysqlnd_connection connection);

public string MysqlndUhConnection::getServerStatistics(
  mysqlnd_connection connection);

public int MysqlndUhConnection::getServerVersion(
  mysqlnd_connection connection);

public string MysqlndUhConnection::getSqlstate(
  mysqlnd_connection connection);

public array MysqlndUhConnection::getStatistics(
  mysqlnd_connection connection);

public int MysqlndUhConnection::getThreadId(
  mysqlnd_connection connection);

public int MysqlndUhConnection::getWarningCount(
  mysqlnd_connection connection);

public bool MysqlndUhConnection::init(
  mysqlnd_connection connection);

public bool MysqlndUhConnection::killConnection(
  mysqlnd_connection connection,
  int pid);

public array MysqlndUhConnection::listFields(
  mysqlnd_connection connection,
  string table,
  string achtung_wild);

public void MysqlndUhConnection::listMethod(
  mysqlnd_connection connection,
  string query,
  string achtung_wild,
  string par1);

public bool MysqlndUhConnection::moreResults(
  mysqlnd_connection connection);

public bool MysqlndUhConnection::nextResult(
  mysqlnd_connection connection);

public bool MysqlndUhConnection::ping(
  mysqlnd_connection connection);

public bool MysqlndUhConnection::query(
  mysqlnd_connection connection,
  string query);

public bool MysqlndUhConnection::queryReadResultsetHeader(
  mysqlnd_connection connection,
  mysqlnd_statement mysqlnd_stmt);
```

```
public bool MysqlndUhConnection::reapQuery(
  mysqlnd_connection connection);

public bool MysqlndUhConnection::refreshServer(
  mysqlnd_connection connection,
  int options);

public bool MysqlndUhConnection::restartPSession(
  mysqlnd_connection connection);

public bool MysqlndUhConnection::selectDb(
  mysqlnd_connection connection,
  string database);

public bool MysqlndUhConnection::sendClose(
  mysqlnd_connection connection);

public bool MysqlndUhConnection::sendQuery(
  mysqlnd_connection connection,
  string query);

public bool MysqlndUhConnection::serverDumpDebugInformation(
  mysqlnd_connection connection);

public bool MysqlndUhConnection::setAutocommit(
  mysqlnd_connection connection,
  int mode);

public bool MysqlndUhConnection::setCharset(
  mysqlnd_connection connection,
  string charset);

public bool MysqlndUhConnection::setClientOption(
  mysqlnd_connection connection,
  int option,
  int value);

public void MysqlndUhConnection::setServerOption(
  mysqlnd_connection connection,
  int option);

public void MysqlndUhConnection::shutdownServer(
  string MYSQLND_UH_RES_MYSQLND_NAME,
  string "level");

public bool MysqlndUhConnection::simpleCommand(
  mysqlnd_connection connection,
  int command,
  string arg,
  int ok_packet,
  bool silent,
  bool ignore_upsert_status);

public bool MysqlndUhConnection::simpleCommandHandleResponse(
  mysqlnd_connection connection,
  int ok_packet,
  bool silent,
  int command,
  bool ignore_upsert_status);

public bool MysqlndUhConnection::sslSet(
  mysqlnd_connection connection,
  string key,
  string cert,
  string ca,
  string capath,
  string cipher);
```

```
  public resource MysqlndUhConnection::stmtInit(
    mysqlnd_connection connection);

  public resource MysqlndUhConnection::storeResult(
    mysqlnd_connection connection);

  public bool MysqlndUhConnection::txCommit(
    mysqlnd_connection connection);

  public bool MysqlndUhConnection::txRollback(
    mysqlnd_connection connection);

  public resource MysqlndUhConnection::useResult(
    mysqlnd_connection connection);

}
```

9.7.1 MysqlndUhConnection::changeUser

- MysqlndUhConnection::changeUser

  Changes the user of the specified mysqlnd database connection

Description

```
  public bool MysqlndUhConnection::changeUser(
    mysqlnd_connection connection,
    string user,
    string password,
    string database,
    bool silent,
    int passwd_len);
```

Changes the user of the specified mysqlnd database connection

Parameters

connection	Mysqlnd connection handle. Do not modify!
user	The MySQL user name.
password	The MySQL password.
database	The MySQL database to change to.
silent	Controls if mysqlnd is allowed to emit errors or not.
passwd_len	Length of the MySQL password.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.9 MysqlndUhConnection::changeUser example

```
<?php
class proxy extends MysqlndUhConnection {
 /* Hook mysqlnd's connection::change_user call */
```

```
 public function changeUser($res, $user, $passwd, $db, $silent, $passwd_len) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::changeUser($res, $user, $passwd, $db, $silent, $passwd_len);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
/* Install proxy/hooks to be used with all future mysqlnd connection */
mysqlnd_uh_set_connection_proxy(new proxy());

/* Create mysqli connection which is using the mysqlnd library */
$mysqli = new mysqli("localhost", "root", "", "test");

/* Example of a user API call which triggers the hooked mysqlnd call */
var_dump($mysqli->change_user("root", "bar", "test"));
?>
```

The above example will output:

```
proxy::changeUser(array (
  0 => NULL,
  1 => 'root',
  2 => 'bar',
  3 => 'test',
  4 => false,
  5 => 3,
))
proxy::changeUser returns false
bool(false)
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_change_user

9.7.2 MysqlndUhConnection::charsetName

- MysqlndUhConnection::charsetName

  Returns the default character set for the database connection

Description

```
 public string MysqlndUhConnection::charsetName(
   mysqlnd_connection connection);
```

Returns the default character set for the database connection.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

The default character set.

Examples

Example 9.10 `MysqlndUhConnection::charsetName` example

```
<?php
class proxy extends MysqlndUhConnection {
  public function charsetName($res) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::charsetName($res);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
var_dump(mysqli_character_set_name($mysqli));
?>
```

The above example will output:

```
proxy::charsetName(array (
  0 => NULL,
))
proxy::charsetName returns 'latin1'
string(6) "latin1"
```

See Also

`mysqlnd_uh_set_connection_proxy`
`mysqli_character_set_name`

9.7.3 `MysqlndUhConnection::close`

- `MysqlndUhConnection::close`

  Closes a previously opened database connection

Description

```
  public bool MysqlndUhConnection::close(
    mysqlnd_connection connection,
    int close_type);
```

Closes a previously opened database connection.

> **Note**
>
> Failing to call the parent implementation may cause memory leaks or crash PHP. This is not considered a bug. Please, keep in mind that the `mysqlnd` library functions have never been designed to be exposed to the user space.

Parameters

connection	The connection to be closed. Do not modify!
close_type	Why the connection is to be closed. The value of *close_type* is one of MYSQLND_UH_MYSQLND_CLOSE_EXPLICIT, MYSQLND_UH_MYSQLND_CLOSE_IMPLICIT, MYSQLND_UH_MYSQLND_CLOSE_DISCONNECTED or MYSQLND_UH_MYSQLND_CLOSE_LAST. The latter should never be seen, unless the default behaviour of the mysqlnd library has been changed by a plugin.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.11 MysqlndUhConnection::close example

```
<?php
function close_type_to_string($close_type) {
 $mapping = array(
  MYSQLND_UH_MYSQLND_CLOSE_DISCONNECTED => "MYSQLND_UH_MYSQLND_CLOSE_DISCONNECTED",
  MYSQLND_UH_MYSQLND_CLOSE_EXPLICIT => "MYSQLND_UH_MYSQLND_CLOSE_EXPLICIT",
  MYSQLND_UH_MYSQLND_CLOSE_IMPLICIT => "MYSQLND_UH_MYSQLND_CLOSE_IMPLICIT",
  MYSQLND_UH_MYSQLND_CLOSE_LAST => "MYSQLND_UH_MYSQLND_CLOSE_IMPLICIT"
 );
 return (isset($mapping[$close_type])) ? $mapping[$close_type] : 'unknown';
}

class proxy extends MysqlndUhConnection {
  public function close($res, $close_type) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   printf("close_type = %s\n", close_type_to_string($close_type));
   /* WARNING: you must call the parent */
   $ret = parent::close($res, $close_type);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->close();
?>
```

The above example will output:

```
proxy::close(array (
  0 => NULL,
  1 => 0,
))
close_type = MYSQLND_UH_MYSQLND_CLOSE_EXPLICIT
proxy::close returns true
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_close
mysql_close

9.7.4 MysqlndUhConnection::connect

- MysqlndUhConnection::connect

  Open a new connection to the MySQL server

Description

```
public bool MysqlndUhConnection::connect(
  mysqlnd_connection connection,
  string host,
  string use",
  string password,
  string database,
  int port,
  string socket,
  int mysql_flags);
```

Open a new connection to the MySQL server.

Parameters

connection	Mysqlnd connection handle. Do not modify!
host	Can be either a host name or an IP address. Passing the NULL value or the string "localhost" to this parameter, the local host is assumed. When possible, pipes will be used instead of the TCP/IP protocol.
user	The MySQL user name.
password	If not provided or NULL, the MySQL server will attempt to authenticate the user against those user records which have no password only. This allows one username to be used with different permissions (depending on if a password as provided or not).
database	If provided will specify the default database to be used when performing queries.
port	Specifies the port number to attempt to connect to the MySQL server.
socket	Specifies the socket or named pipe that should be used. If NULL, mysqlnd will default to /tmp/mysql.sock.
mysql_flags	Connection options.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.12 MysqlndUhConnection::connect example

```
<?php
class proxy extends MysqlndUhConnection {
 public function connect($res, $host, $user, $passwd, $db, $port, $socket, $mysql_flags) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::connect($res, $host, $user, $passwd, $db, $port, $socket, $mysql_flags);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
?>
```

The above example will output:

```
proxy::connect(array (
  0 => NULL,
  1 => 'localhost',
  2 => 'root',
  3 => '',
  4 => 'test',
  5 => 3306,
  6 => NULL,
  7 => 131072,
))
proxy::connect returns true
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_connect
mysql_connect

9.7.5 MysqlndUhConnection::__construct

- MysqlndUhConnection::__construct

  The __construct purpose

Description

```
public MysqlndUhConnection::__construct();
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

This function has no parameters.

Return Values

9.7.6 MysqlndUhConnection::endPSession

- MysqlndUhConnection::endPSession

  End a persistent connection

Description

```
public bool MysqlndUhConnection::endPSession(
  mysqlnd_connection connection);
```

End a persistent connection

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.13 MysqlndUhConnection::endPSession example

```
<?php
class proxy extends MysqlndUhConnection {
 public function endPSession($conn) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::endPSession($conn);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("p:localhost", "root", "", "test");
$mysqli->close();
?>
```

The above example will output:

```
proxy::endPSession(array (
  0 => NULL,
))
proxy::endPSession returns true
```

See Also

mysqlnd_uh_set_connection_proxy

9.7.7 MysqlndUhConnection::escapeString

- MysqlndUhConnection::escapeString

  Escapes special characters in a string for use in an SQL statement, taking into account the current charset of the connection

Description

```
public string MysqlndUhConnection::escapeString(
  mysqlnd_connection connection,
  string escape_string);
```

Escapes special characters in a string for use in an SQL statement, taking into account the current charset of the connection.

Parameters

MYSQLND_UH_RES_MYSQLND_NAME	Mysqlnd connection handle. Do not modify!
escape_string	The string to be escaped.

Return Values

The escaped string.

Examples

Example 9.14 MysqlndUhConnection::escapeString example

```
<?php
class proxy extends MysqlndUhConnection {
 public function escapeString($res, $string) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::escapeString($res, $string);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->set_charset("latin1");
$mysqli->real_escape_string("test0'test");
?>
```

The above example will output:

```
proxy::escapeString(array (
  0 => NULL,
  1 => 'test0\'test',
))
proxy::escapeString returns 'test0\\\'test'
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_real_escape_string
mysql_real_escape_string

9.7.8 MysqlndUhConnection::getAffectedRows

- MysqlndUhConnection::getAffectedRows

  Gets the number of affected rows in a previous MySQL operation

Description

```
public int MysqlndUhConnection::getAffectedRows(
  mysqlnd_connection connection);
```

Gets the number of affected rows in a previous MySQL operation.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Number of affected rows.

Examples

Example 9.15 MysqlndUhConnection::getAffectedRows example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getAffectedRows($res) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::getAffectedRows($res);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT)");
$mysqli->query("INSERT INTO test(id) VALUES (1)");
var_dump($mysqli->affected_rows);
?>
```

The above example will output:

```
proxy::getAffectedRows(array (
  0 => NULL,
))
proxy::getAffectedRows returns 1
```

```
int(1)
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_affected_rows
mysql_affected_rows

9.7.9 MysqlndUhConnection::getErrorNumber

- MysqlndUhConnection::getErrorNumber

  Returns the error code for the most recent function call

Description

```
  public int MysqlndUhConnection::getErrorNumber(
    mysqlnd_connection connection);
```

Returns the error code for the most recent function call.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Error code for the most recent function call.

Examples

MysqlndUhConnection::getErrorNumber is not only executed after the invocation of a user space API call which maps directly to it but also called internally.

Example 9.16 MysqlndUhConnection::getErrorNumber example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getErrorNumber($res) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::getErrorNumber($res);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

printf("connect...\n");
$mysqli = new mysqli("localhost", "root", "", "test");
printf("query...\n");
$mysqli->query("PLEASE_LET_THIS_BE_INVALID_SQL");
printf("errno...\n");
var_dump($mysqli->errno);
printf("close...\n");
$mysqli->close();
?>
```

```
```

The above example will output:

```
connect...
proxy::getErrorNumber(array (
  0 => NULL,
))
proxy::getErrorNumber returns 0
query...
errno...
proxy::getErrorNumber(array (
  0 => NULL,
))
proxy::getErrorNumber returns 1064
int(1064)
close...
```

See Also

mysqlnd_uh_set_connection_proxy
MysqlndUhConnection::getErrorString
mysqli_errno
mysql_errno

9.7.10 MysqlndUhConnection::getErrorString

- MysqlndUhConnection::getErrorString

  Returns a string description of the last error

Description

```
  public string MysqlndUhConnection::getErrorString(
    mysqlnd_connection connection);
```

Returns a string description of the last error.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Error string for the most recent function call.

Examples

MysqlndUhConnection::getErrorString is not only executed after the invocation of a user space API call which maps directly to it but also called internally.

Example 9.17 MysqlndUhConnection::getErrorString example

```
<?php
class proxy extends MysqlndUhConnection {
```

```
 public function getErrorString($res) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::getErrorString($res);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

printf("connect...\n");
$mysqli = new mysqli("localhost", "root", "", "test");
printf("query...\n");
$mysqli->query("WILL_I_EVER_LEARN_SQL?");
printf("errno...\n");
var_dump($mysqli->error);
printf("close...\n");
$mysqli->close();
?>
```

The above example will output:

```
connect...
proxy::getErrorString(array (
  0 => NULL,
))
proxy::getErrorString returns ''
query...
errno...
proxy::getErrorString(array (
  0 => NULL,
))
proxy::getErrorString returns 'You have an error in your SQL syntax; check the manual that corresponds to y
string(168) "You have an error in your SQL syntax; check the manual that corresponds to your MySQL server v
close...
```

See Also

mysqlnd_uh_set_connection_proxy
MysqlndUhConnection::getErrorNumber
mysqli_error
mysql_error

9.7.11 MysqlndUhConnection::getFieldCount

- MysqlndUhConnection::getFieldCount

  Returns the number of columns for the most recent query

Description

```
 public int MysqlndUhConnection::getFieldCount(
   mysqlnd_connection connection);
```

Returns the number of columns for the most recent query.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Number of columns.

Examples

`MysqlndUhConnection::getFieldCount` is not only executed after the invocation of a user space API call which maps directly to it but also called internally.

Example 9.18 `MysqlndUhConnection::getFieldCount` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getFieldCount($res) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::getFieldCount($res);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->query("WILL_I_EVER_LEARN_SQL?");
var_dump($mysqli->field_count);
$mysqli->query("SELECT 1, 2, 3 FROM DUAL");
var_dump($mysqli->field_count);
?>
```

The above example will output:

```
proxy::getFieldCount(array (
  0 => NULL,
))
proxy::getFieldCount returns 0
int(0)
proxy::getFieldCount(array (
  0 => NULL,
))
proxy::getFieldCount returns 3
proxy::getFieldCount(array (
  0 => NULL,
))
proxy::getFieldCount returns 3
int(3)
```

See Also

`mysqlnd_uh_set_connection_proxy`
`mysqli_field_count`

9.7.12 `MysqlndUhConnection::getHostInformation`

- `MysqlndUhConnection::getHostInformation`

  Returns a string representing the type of connection used

Description

```
public string MysqlndUhConnection::getHostInformation(
  mysqlnd_connection connection);
```

Returns a string representing the type of connection used.

Parameters

`connection`	Mysqlnd connection handle. Do not modify!

Return Values

Connection description.

Examples

Example 9.19 `MysqlndUhConnection::getHostInformation` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getHostInformation($res) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::getHostInformation($res);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
var_dump($mysqli->host_info);
?>
```

The above example will output:

```
proxy::getHostInformation(array (
  0 => NULL,
))
proxy::getHostInformation returns 'Localhost via UNIX socket'
string(25) "Localhost via UNIX socket"
```

See Also

`mysqlnd_uh_set_connection_proxy`
`mysqli_get_host_info`
`mysql_get_host_info`

9.7.13 `MysqlndUhConnection::getLastInsertId`

- `MysqlndUhConnection::getLastInsertId`

  Returns the auto generated id used in the last query.

Description

```
  public int MysqlndUhConnection::getLastInsertId(
    mysqlnd_connection connection);
```

Returns the auto generated id used in the last query.

Parameters

`connection`	Mysqlnd connection handle. Do not modify!

Return Values

Last insert id.

Examples

Example 9.20 `MysqlndUhConnection::getLastInsertId` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getLastInsertId($res) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::getLastInsertId($res);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->query("DROP TABLE IF EXISTS test");
$mysqli->query("CREATE TABLE test(id INT AUTO_INCREMENT PRIMARY KEY, col VARCHAR(255))");
$mysqli->query("INSERT INTO test(col) VALUES ('a')");
var_dump($mysqli->insert_id);
?>
```

The above example will output:

```
proxy::getLastInsertId(array (
  0 => NULL,
))
proxy::getLastInsertId returns 1
int(1)
```

See Also

`mysqlnd_uh_set_connection_proxy`
`mysqli_insert_id`
`mysql_insert_id`

9.7.14 MysqlndUhConnection::getLastMessage

- MysqlndUhConnection::getLastMessage

  Retrieves information about the most recently executed query

Description

```
public void MysqlndUhConnection::getLastMessage(
  mysqlnd_connection connection);
```

Retrieves information about the most recently executed query.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Last message. Trying to return a string longer than 511 bytes will cause an error of the type E_WARNING and result in the string being truncated.

Examples

Example 9.21 MysqlndUhConnection::getLastMessage example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getLastMessage($res) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::getLastMessage($res);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
var_dump($mysqli->info);
$mysqli->query("DROP TABLE IF EXISTS test");
var_dump($mysqli->info);
?>
```

The above example will output:

```
proxy::getLastMessage(array (
  0 => NULL,
))
proxy::getLastMessage returns ''
string(0) ""
proxy::getLastMessage(array (
  0 => NULL,
))
proxy::getLastMessage returns ''
string(0) ""
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_info
mysql_info

9.7.15 MysqlndUhConnection::getProtocolInformation

- MysqlndUhConnection::getProtocolInformation

  Returns the version of the MySQL protocol used

Description

```
public string MysqlndUhConnection::getProtocolInformation(
  mysqlnd_connection connection);
```

Returns the version of the MySQL protocol used.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

The protocol version.

Examples

Example 9.22 MysqlndUhConnection::getProtocolInformation example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getProtocolInformation($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::getProtocolInformation($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
var_dump($mysqli->protocol_version);
?>
```

The above example will output:

```
proxy::getProtocolInformation(array (
  0 => NULL,
))
proxy::getProtocolInformation returns 10
int(10)
```

```
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_get_proto_info
mysql_get_proto_info

9.7.16 MysqlndUhConnection::getServerInformation

- MysqlndUhConnection::getServerInformation

  Returns the version of the MySQL server

Description

```
public string MysqlndUhConnection::getServerInformation(
  mysqlnd_connection connection);
```

Returns the version of the MySQL server.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

The server version.

Examples

Example 9.23 MysqlndUhConnection::getServerInformation example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getServerInformation($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::getServerInformation($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
var_dump($mysqli->server_info);
?>
```

The above example will output:

```
proxy::getServerInformation(array (
  0 => NULL,
))
proxy::getServerInformation returns '5.1.45-debug-log'
string(16) "5.1.45-debug-log"
```

```
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_get_server_info
mysql_get_server_info

9.7.17 MysqlndUhConnection::getServerStatistics

- MysqlndUhConnection::getServerStatistics

  Gets the current system status

Description

```
public string MysqlndUhConnection::getServerStatistics(
  mysqlnd_connection connection);
```

Gets the current system status.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

The system status message.

Examples

Example 9.24 MysqlndUhConnection::getServerStatistics example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getServerStatistics($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::getServerStatistics($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
var_dump(mysqli_stat($mysqli));
?>
```

The above example will output:

```
proxy::getServerStatistics(array (
  0 => NULL,
))
proxy::getServerStatistics returns 'Uptime: 2059995  Threads: 1  Questions: 126157  Slow queries: 0  Opens: 63
string(140) "Uptime: 2059995  Threads: 1  Questions: 126157  Slow queries: 0  Opens: 6377  Flush tables: 1  Op
```

```
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_stat
mysql_stat

9.7.18 MysqlndUhConnection::getServerVersion

- MysqlndUhConnection::getServerVersion

  Returns the version of the MySQL server as an integer

Description

```
public int MysqlndUhConnection::getServerVersion(
  mysqlnd_connection connection);
```

Returns the version of the MySQL server as an integer.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

The MySQL version.

Examples

Example 9.25 MysqlndUhConnection::getServerVersion example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getServerVersion($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::getServerVersion($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
var_dump($mysqli->server_version);
?>
```

The above example will output:

```
proxy::getServerVersion(array (
  0 => NULL,
))
proxy::getServerVersion returns 50145
int(50145)
```

```
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_get_server_version
mysql_get_server_version

9.7.19 MysqlndUhConnection::getSqlstate

- MysqlndUhConnection::getSqlstate

  Returns the SQLSTATE error from previous MySQL operation

Description

```
  public string MysqlndUhConnection::getSqlstate(
    mysqlnd_connection connection);
```

Returns the SQLSTATE error from previous MySQL operation.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

The SQLSTATE code.

Examples

Example 9.26 MysqlndUhConnection::getSqlstate example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getSqlstate($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::getSqlstate($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
var_dump($mysqli->sqlstate);
$mysqli->query("AN_INVALID_REQUEST_TO_PROVOKE_AN_ERROR");
var_dump($mysqli->sqlstate);
?>
```

The above example will output:

```
proxy::getSqlstate(array (
  0 => NULL,
))
```

```
proxy::getSqlstate returns '00000'
string(5) "00000"
proxy::getSqlstate(array (
  0 => NULL,
))
proxy::getSqlstate returns '42000'
string(5) "42000"
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_sql_state

9.7.20 MysqlndUhConnection::getStatistics

- MysqlndUhConnection::getStatistics

  Returns statistics about the client connection.

Description

```
  public array MysqlndUhConnection::getStatistics(
    mysqlnd_connection connection);
```

Returns statistics about the client connection.

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Connection statistics collected by mysqlnd.

Examples

Example 9.27 MysqlndUhConnection::getStatistics example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getStatistics($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::getStatistics($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
var_dump($mysqli->get_connection_stats());
?>
```

The above example will output:

```
proxy::getStatistics(array (
  0 => NULL,
))
proxy::getStatistics returns array (
  'bytes_sent' => '73',
  'bytes_received' => '77',
  'packets_sent' => '2',
  'packets_received' => '2',
  'protocol_overhead_in' => '8',
  'protocol_overhead_out' => '8',
  'bytes_received_ok_packet' => '0',
  'bytes_received_eof_packet' => '0',
  'bytes_received_rset_header_packet' => '0',
  'bytes_received_rset_field_meta_packet' => '0',
  'bytes_received_rset_row_packet' => '0',
  'bytes_received_prepare_response_packet' => '0',
  'bytes_received_change_user_packet' => '0',
  'packets_sent_command' => '0',
  'packets_received_ok' => '0',
  'packets_received_eof' => '0',
  'packets_received_rset_header' => '0',
  'packets_received_rset_field_meta' => '0',
  'packets_received_rset_row' => '0',
  'packets_received_prepare_response' => '0',
  'packets_received_change_user' => '0',
  'result_set_queries' => '0',
  'non_result_set_queries' => '0',
  'no_index_used' => '0',
  'bad_index_used' => '0',
  'slow_queries' => '0',
  'buffered_sets' => '0',
  'unbuffered_sets' => '0',
  'ps_buffered_sets' => '0',
  'ps_unbuffered_sets' => '0',
  'flushed_normal_sets' => '0',
  'flushed_ps_sets' => '0',
  'ps_prepared_never_executed' => '0',
  'ps_prepared_once_executed' => '0',
  'rows_fetched_from_server_normal' => '0',
  'rows_fetched_from_server_ps' => '0',
  'rows_buffered_from_client_normal' => '0',
  'rows_buffered_from_client_ps' => '0',
  'rows_fetched_from_client_normal_buffered' => '0',
  'rows_fetched_from_client_normal_unbuffered' => '0',
  'rows_fetched_from_client_ps_buffered' => '0',
  'rows_fetched_from_client_ps_unbuffered' => '0',
  'rows_fetched_from_client_ps_cursor' => '0',
  'rows_affected_normal' => '0',
  'rows_affected_ps' => '0',
  'rows_skipped_normal' => '0',
  'rows_skipped_ps' => '0',
  'copy_on_write_saved' => '0',
  'copy_on_write_performed' => '0',
  'command_buffer_too_small' => '0',
  'connect_success' => '1',
  'connect_failure' => '0',
  'connection_reused' => '0',
  'reconnect' => '0',
  'pconnect_success' => '0',
  'active_connections' => '1',
  'active_persistent_connections' => '0',
```

```
  'explicit_close' => '0',
  'implicit_close' => '0',
  'disconnect_close' => '0',
  'in_middle_of_command_close' => '0',
  'explicit_free_result' => '0',
  'implicit_free_result' => '0',
  'explicit_stmt_close' => '0',
  'implicit_stmt_close' => '0',
  'mem_emalloc_count' => '0',
  'mem_emalloc_amount' => '0',
  'mem_ecalloc_count' => '0',
  'mem_ecalloc_amount' => '0',
  'mem_erealloc_count' => '0',
  'mem_erealloc_amount' => '0',
  'mem_efree_count' => '0',
  'mem_efree_amount' => '0',
  'mem_malloc_count' => '0',
  'mem_malloc_amount' => '0',
  'mem_calloc_count' => '0',
  'mem_calloc_amount' => '0',
  'mem_realloc_count' => '0',
  'mem_realloc_amount' => '0',
  'mem_free_count' => '0',
  'mem_free_amount' => '0',
  'mem_estrndup_count' => '0',
  'mem_strndup_count' => '0',
  'mem_estndup_count' => '0',
  'mem_strdup_count' => '0',
  'proto_text_fetched_null' => '0',
  'proto_text_fetched_bit' => '0',
  'proto_text_fetched_tinyint' => '0',
  'proto_text_fetched_short' => '0',
  'proto_text_fetched_int24' => '0',
  'proto_text_fetched_int' => '0',
  'proto_text_fetched_bigint' => '0',
  'proto_text_fetched_decimal' => '0',
  'proto_text_fetched_float' => '0',
  'proto_text_fetched_double' => '0',
  'proto_text_fetched_date' => '0',
  'proto_text_fetched_year' => '0',
  'proto_text_fetched_time' => '0',
  'proto_text_fetched_datetime' => '0',
  'proto_text_fetched_timestamp' => '0',
  'proto_text_fetched_string' => '0',
  'proto_text_fetched_blob' => '0',
  'proto_text_fetched_enum' => '0',
  'proto_text_fetched_set' => '0',
  'proto_text_fetched_geometry' => '0',
  'proto_text_fetched_other' => '0',
  'proto_binary_fetched_null' => '0',
  'proto_binary_fetched_bit' => '0',
  'proto_binary_fetched_tinyint' => '0',
  'proto_binary_fetched_short' => '0',
  'proto_binary_fetched_int24' => '0',
  'proto_binary_fetched_int' => '0',
  'proto_binary_fetched_bigint' => '0',
  'proto_binary_fetched_decimal' => '0',
  'proto_binary_fetched_float' => '0',
  'proto_binary_fetched_double' => '0',
  'proto_binary_fetched_date' => '0',
  'proto_binary_fetched_year' => '0',
  'proto_binary_fetched_time' => '0',
  'proto_binary_fetched_datetime' => '0',
  'proto_binary_fetched_timestamp' => '0',
  'proto_binary_fetched_string' => '0',
  'proto_binary_fetched_blob' => '0',
  'proto_binary_fetched_enum' => '0',
```

```
  'proto_binary_fetched_set' => '0',
  'proto_binary_fetched_geometry' => '0',
  'proto_binary_fetched_other' => '0',
  'init_command_executed_count' => '0',
  'init_command_failed_count' => '0',
  'com_quit' => '0',
  'com_init_db' => '0',
  'com_query' => '0',
  'com_field_list' => '0',
  'com_create_db' => '0',
  'com_drop_db' => '0',
  'com_refresh' => '0',
  'com_shutdown' => '0',
  'com_statistics' => '0',
  'com_process_info' => '0',
  'com_connect' => '0',
  'com_process_kill' => '0',
  'com_debug' => '0',
  'com_ping' => '0',
  'com_time' => '0',
  'com_delayed_insert' => '0',
  'com_change_user' => '0',
  'com_binlog_dump' => '0',
  'com_table_dump' => '0',
  'com_connect_out' => '0',
  'com_register_slave' => '0',
  'com_stmt_prepare' => '0',
  'com_stmt_execute' => '0',
  'com_stmt_send_long_data' => '0',
  'com_stmt_close' => '0',
  'com_stmt_reset' => '0',
  'com_stmt_set_option' => '0',
  'com_stmt_fetch' => '0',
  'com_deamon' => '0',
  'bytes_received_real_data_normal' => '0',
  'bytes_received_real_data_ps' => '0',
)
array(160) {
  ["bytes_sent"]=>
  string(2) "73"
  ["bytes_received"]=>
  string(2) "77"
  ["packets_sent"]=>
  string(1) "2"
  ["packets_received"]=>
  string(1) "2"
  ["protocol_overhead_in"]=>
  string(1) "8"
  ["protocol_overhead_out"]=>
  string(1) "8"
  ["bytes_received_ok_packet"]=>
  string(1) "0"
  ["bytes_received_eof_packet"]=>
  string(1) "0"
  ["bytes_received_rset_header_packet"]=>
  string(1) "0"
  ["bytes_received_rset_field_meta_packet"]=>
  string(1) "0"
  ["bytes_received_rset_row_packet"]=>
  string(1) "0"
  ["bytes_received_prepare_response_packet"]=>
  string(1) "0"
  ["bytes_received_change_user_packet"]=>
  string(1) "0"
  ["packets_sent_command"]=>
  string(1) "0"
  ["packets_received_ok"]=>
```

```
  string(1) "0"
  ["packets_received_eof"]=>
  string(1) "0"
  ["packets_received_rset_header"]=>
  string(1) "0"
  ["packets_received_rset_field_meta"]=>
  string(1) "0"
  ["packets_received_rset_row"]=>
  string(1) "0"
  ["packets_received_prepare_response"]=>
  string(1) "0"
  ["packets_received_change_user"]=>
  string(1) "0"
  ["result_set_queries"]=>
  string(1) "0"
  ["non_result_set_queries"]=>
  string(1) "0"
  ["no_index_used"]=>
  string(1) "0"
  ["bad_index_used"]=>
  string(1) "0"
  ["slow_queries"]=>
  string(1) "0"
  ["buffered_sets"]=>
  string(1) "0"
  ["unbuffered_sets"]=>
  string(1) "0"
  ["ps_buffered_sets"]=>
  string(1) "0"
  ["ps_unbuffered_sets"]=>
  string(1) "0"
  ["flushed_normal_sets"]=>
  string(1) "0"
  ["flushed_ps_sets"]=>
  string(1) "0"
  ["ps_prepared_never_executed"]=>
  string(1) "0"
  ["ps_prepared_once_executed"]=>
  string(1) "0"
  ["rows_fetched_from_server_normal"]=>
  string(1) "0"
  ["rows_fetched_from_server_ps"]=>
  string(1) "0"
  ["rows_buffered_from_client_normal"]=>
  string(1) "0"
  ["rows_buffered_from_client_ps"]=>
  string(1) "0"
  ["rows_fetched_from_client_normal_buffered"]=>
  string(1) "0"
  ["rows_fetched_from_client_normal_unbuffered"]=>
  string(1) "0"
  ["rows_fetched_from_client_ps_buffered"]=>
  string(1) "0"
  ["rows_fetched_from_client_ps_unbuffered"]=>
  string(1) "0"
  ["rows_fetched_from_client_ps_cursor"]=>
  string(1) "0"
  ["rows_affected_normal"]=>
  string(1) "0"
  ["rows_affected_ps"]=>
  string(1) "0"
  ["rows_skipped_normal"]=>
  string(1) "0"
  ["rows_skipped_ps"]=>
  string(1) "0"
  ["copy_on_write_saved"]=>
  string(1) "0"
```

```
  ["copy_on_write_performed"]=>
  string(1) "0"
  ["command_buffer_too_small"]=>
  string(1) "0"
  ["connect_success"]=>
  string(1) "1"
  ["connect_failure"]=>
  string(1) "0"
  ["connection_reused"]=>
  string(1) "0"
  ["reconnect"]=>
  string(1) "0"
  ["pconnect_success"]=>
  string(1) "0"
  ["active_connections"]=>
  string(1) "1"
  ["active_persistent_connections"]=>
  string(1) "0"
  ["explicit_close"]=>
  string(1) "0"
  ["implicit_close"]=>
  string(1) "0"
  ["disconnect_close"]=>
  string(1) "0"
  ["in_middle_of_command_close"]=>
  string(1) "0"
  ["explicit_free_result"]=>
  string(1) "0"
  ["implicit_free_result"]=>
  string(1) "0"
  ["explicit_stmt_close"]=>
  string(1) "0"
  ["implicit_stmt_close"]=>
  string(1) "0"
  ["mem_emalloc_count"]=>
  string(1) "0"
  ["mem_emalloc_amount"]=>
  string(1) "0"
  ["mem_ecalloc_count"]=>
  string(1) "0"
  ["mem_ecalloc_amount"]=>
  string(1) "0"
  ["mem_erealloc_count"]=>
  string(1) "0"
  ["mem_erealloc_amount"]=>
  string(1) "0"
  ["mem_efree_count"]=>
  string(1) "0"
  ["mem_efree_amount"]=>
  string(1) "0"
  ["mem_malloc_count"]=>
  string(1) "0"
  ["mem_malloc_amount"]=>
  string(1) "0"
  ["mem_calloc_count"]=>
  string(1) "0"
  ["mem_calloc_amount"]=>
  string(1) "0"
  ["mem_realloc_count"]=>
  string(1) "0"
  ["mem_realloc_amount"]=>
  string(1) "0"
  ["mem_free_count"]=>
  string(1) "0"
  ["mem_free_amount"]=>
  string(1) "0"
  ["mem_estrndup_count"]=>
```

```
  string(1) "0"
  ["mem_strndup_count"]=>
  string(1) "0"
  ["mem_estndup_count"]=>
  string(1) "0"
  ["mem_strdup_count"]=>
  string(1) "0"
  ["proto_text_fetched_null"]=>
  string(1) "0"
  ["proto_text_fetched_bit"]=>
  string(1) "0"
  ["proto_text_fetched_tinyint"]=>
  string(1) "0"
  ["proto_text_fetched_short"]=>
  string(1) "0"
  ["proto_text_fetched_int24"]=>
  string(1) "0"
  ["proto_text_fetched_int"]=>
  string(1) "0"
  ["proto_text_fetched_bigint"]=>
  string(1) "0"
  ["proto_text_fetched_decimal"]=>
  string(1) "0"
  ["proto_text_fetched_float"]=>
  string(1) "0"
  ["proto_text_fetched_double"]=>
  string(1) "0"
  ["proto_text_fetched_date"]=>
  string(1) "0"
  ["proto_text_fetched_year"]=>
  string(1) "0"
  ["proto_text_fetched_time"]=>
  string(1) "0"
  ["proto_text_fetched_datetime"]=>
  string(1) "0"
  ["proto_text_fetched_timestamp"]=>
  string(1) "0"
  ["proto_text_fetched_string"]=>
  string(1) "0"
  ["proto_text_fetched_blob"]=>
  string(1) "0"
  ["proto_text_fetched_enum"]=>
  string(1) "0"
  ["proto_text_fetched_set"]=>
  string(1) "0"
  ["proto_text_fetched_geometry"]=>
  string(1) "0"
  ["proto_text_fetched_other"]=>
  string(1) "0"
  ["proto_binary_fetched_null"]=>
  string(1) "0"
  ["proto_binary_fetched_bit"]=>
  string(1) "0"
  ["proto_binary_fetched_tinyint"]=>
  string(1) "0"
  ["proto_binary_fetched_short"]=>
  string(1) "0"
  ["proto_binary_fetched_int24"]=>
  string(1) "0"
  ["proto_binary_fetched_int"]=>
  string(1) "0"
  ["proto_binary_fetched_bigint"]=>
  string(1) "0"
  ["proto_binary_fetched_decimal"]=>
  string(1) "0"
  ["proto_binary_fetched_float"]=>
  string(1) "0"
```

```
  ["proto_binary_fetched_double"]=>
  string(1) "0"
  ["proto_binary_fetched_date"]=>
  string(1) "0"
  ["proto_binary_fetched_year"]=>
  string(1) "0"
  ["proto_binary_fetched_time"]=>
  string(1) "0"
  ["proto_binary_fetched_datetime"]=>
  string(1) "0"
  ["proto_binary_fetched_timestamp"]=>
  string(1) "0"
  ["proto_binary_fetched_string"]=>
  string(1) "0"
  ["proto_binary_fetched_blob"]=>
  string(1) "0"
  ["proto_binary_fetched_enum"]=>
  string(1) "0"
  ["proto_binary_fetched_set"]=>
  string(1) "0"
  ["proto_binary_fetched_geometry"]=>
  string(1) "0"
  ["proto_binary_fetched_other"]=>
  string(1) "0"
  ["init_command_executed_count"]=>
  string(1) "0"
  ["init_command_failed_count"]=>
  string(1) "0"
  ["com_quit"]=>
  string(1) "0"
  ["com_init_db"]=>
  string(1) "0"
  ["com_query"]=>
  string(1) "0"
  ["com_field_list"]=>
  string(1) "0"
  ["com_create_db"]=>
  string(1) "0"
  ["com_drop_db"]=>
  string(1) "0"
  ["com_refresh"]=>
  string(1) "0"
  ["com_shutdown"]=>
  string(1) "0"
  ["com_statistics"]=>
  string(1) "0"
  ["com_process_info"]=>
  string(1) "0"
  ["com_connect"]=>
  string(1) "0"
  ["com_process_kill"]=>
  string(1) "0"
  ["com_debug"]=>
  string(1) "0"
  ["com_ping"]=>
  string(1) "0"
  ["com_time"]=>
  string(1) "0"
  ["com_delayed_insert"]=>
  string(1) "0"
  ["com_change_user"]=>
  string(1) "0"
  ["com_binlog_dump"]=>
  string(1) "0"
  ["com_table_dump"]=>
  string(1) "0"
  ["com_connect_out"]=>
```

```
  string(1) "0"
  ["com_register_slave"]=>
  string(1) "0"
  ["com_stmt_prepare"]=>
  string(1) "0"
  ["com_stmt_execute"]=>
  string(1) "0"
  ["com_stmt_send_long_data"]=>
  string(1) "0"
  ["com_stmt_close"]=>
  string(1) "0"
  ["com_stmt_reset"]=>
  string(1) "0"
  ["com_stmt_set_option"]=>
  string(1) "0"
  ["com_stmt_fetch"]=>
  string(1) "0"
  ["com_deamon"]=>
  string(1) "0"
  ["bytes_received_real_data_normal"]=>
  string(1) "0"
  ["bytes_received_real_data_ps"]=>
  string(1) "0"
}
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_get_connection_stats

9.7.21 MysqlndUhConnection::getThreadId

- MysqlndUhConnection::getThreadId

  Returns the thread ID for the current connection

Description

```
  public int MysqlndUhConnection::getThreadId(
    mysqlnd_connection connection);
```

Returns the thread ID for the current connection.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Connection thread id.

Examples

Example 9.28 MysqlndUhConnection::getThreadId example

```
<?php
class proxy extends MysqlndUhConnection {
```

```
 public function getThreadId($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::getThreadId($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
var_dump($mysqli->thread_id);
?>
```

The above example will output:

```
proxy::getThreadId(array (
  0 => NULL,
))
proxy::getThreadId returns 27646
int(27646)
```

See Also

`mysqlnd_uh_set_connection_proxy`
`mysqli_thread_id`
`mysql_thread_id`

9.7.22 `MysqlndUhConnection::getWarningCount`

- `MysqlndUhConnection::getWarningCount`

  Returns the number of warnings from the last query for the given link

Description

```
  public int MysqlndUhConnection::getWarningCount(
    mysqlnd_connection connection);
```

Returns the number of warnings from the last query for the given link.

Parameters

`connection`	Mysqlnd connection handle. Do not modify!

Return Values

Number of warnings.

Examples

Example 9.29 `MysqlndUhConnection::getWarningCount` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function getWarningCount($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::getWarningCount($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
var_dump($mysqli->warning_count);
?>
```

The above example will output:

```
proxy::getWarningCount(array (
  0 => NULL,
))
proxy::getWarningCount returns 0
int(0)
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_warning_count

9.7.23 MysqlndUhConnection::init

- MysqlndUhConnection::init

  Initialize mysqlnd connection

Description

```
public bool MysqlndUhConnection::init(
  mysqlnd_connection connection);
```

Initialize mysqlnd connection. This is an mysqlnd internal call to initialize the connection object.

> **Note**
>
> Failing to call the parent implementation may cause memory leaks or crash PHP. This is not considered a bug. Please, keep in mind that the mysqlnd library functions have never been designed to be exposed to the user space.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.30 `MysqlndUhConnection::init` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function init($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::init($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
?>
```

The above example will output:

```
proxy::init(array (
  0 => NULL,
))
proxy::init returns true
```

See Also

`mysqlnd_uh_set_connection_proxy`

9.7.24 `MysqlndUhConnection::killConnection`

- `MysqlndUhConnection::killConnection`

  Asks the server to kill a MySQL thread

Description

```
  public bool MysqlndUhConnection::killConnection(
    mysqlnd_connection connection,
    int pid);
```

Asks the server to kill a MySQL thread.

Parameters

`connection`	Mysqlnd connection handle. Do not modify!
`pid`	Thread Id of the connection to be killed.

Return Values

Returns `TRUE` on success. Otherwise, returns `FALSE`

Examples

Example 9.31 `MysqlndUhConnection::kill` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function killConnection($res, $pid) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::killConnection($res, $pid);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->kill($mysqli->thread_id);
?>
```

The above example will output:

```
proxy::killConnection(array (
  0 => NULL,
  1 => 27650,
))
proxy::killConnection returns true
```

See Also

`mysqlnd_uh_set_connection_proxy`
`mysqli_kill`

9.7.25 `MysqlndUhConnection::listFields`

- `MysqlndUhConnection::listFields`

  List MySQL table fields

Description

```
  public array MysqlndUhConnection::listFields(
    mysqlnd_connection connection,
    string table,
    string achtung_wild);
```

List MySQL table fields.

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

connection	Mysqlnd connection handle. Do not modify!
table	The name of the table that's being queried.
pattern	Name pattern.

Return Values

Examples

Example 9.32 `MysqlndUhConnection::listFields` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function listFields($res, $table, $pattern) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::listFields($res, $table, $pattern);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysql = mysql_connect("localhost", "root", "");
mysql_select_db("test", $mysql);
mysql_query("DROP TABLE IF EXISTS test_a", $mysql);
mysql_query("CREATE TABLE test_a(id INT, col1 VARCHAR(255))", $mysql);
$res = mysql_list_fields("test", "test_a", $mysql);
printf("num_rows = %d\n", mysql_num_rows($res));
while ($row = mysql_fetch_assoc($res))
 var_dump($row);
?>
```

The above example will output:

```
proxy::listFields(array (
  0 => NULL,
  1 => 'test_a',
  2 => '',
))
proxy::listFields returns NULL
num_rows = 0
```

See Also

`mysqlnd_uh_set_connection_proxy`
`mysql_list_fields`

9.7.26 `MysqlndUhConnection::listMethod`

- `MysqlndUhConnection::listMethod`

  Wrapper for assorted list commands

Description

```
  public void MysqlndUhConnection::listMethod(
    mysqlnd_connection connection,
    string query,
    string achtung_wild,
    string par1);
```

Wrapper for assorted list commands.

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

connection	Mysqlnd connection handle. Do not modify!
query	SHOW command to be executed.
achtung_wild	
par1	

Return Values

Return Values

TODO

Examples

Example 9.33 `MysqlndUhConnection::listMethod` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function listMethod($res, $query, $pattern, $par1) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::listMethod($res, $query, $pattern, $par1);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysql = mysql_connect("localhost", "root", "");
$res = mysql_list_dbs($mysql);
printf("num_rows = %d\n", mysql_num_rows($res));
while ($row = mysql_fetch_assoc($res))
 var_dump($row);
?>
```

The above example will output:

```
proxy::listMethod(array (
  0 => NULL,
```

```
  1 => 'SHOW DATABASES',
  2 => '',
  3 => '',
))
proxy::listMethod returns NULL
num_rows = 6
array(1) {
  ["Database"]=>
  string(18) "information_schema"
}
array(1) {
  ["Database"]=>
  string(5) "mysql"
}
array(1) {
  ["Database"]=>
  string(8) "oxid_new"
}
array(1) {
  ["Database"]=>
  string(7) "phptest"
}
array(1) {
  ["Database"]=>
  string(7) "pushphp"
}
array(1) {
  ["Database"]=>
  string(4) "test"
}
```

See Also

mysqlnd_uh_set_connection_proxy
mysql_list_dbs

9.7.27 MysqlndUhConnection::moreResults

- MysqlndUhConnection::moreResults

  Check if there are any more query results from a multi query

Description

```
  public bool MysqlndUhConnection::moreResults(
    mysqlnd_connection connection);
```

Check if there are any more query results from a multi query.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.34 `MysqlndUhConnection::moreResults` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function moreResults($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::moreResults($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->multi_query("SELECT 1 AS _one; SELECT 2 AS _two");
do {
  $res = $mysqli->store_result();
  var_dump($res->fetch_assoc());
  printf("%s\n", str_repeat("-", 40));
} while ($mysqli->more_results() && $mysqli->next_result());
?>
```

The above example will output:

```
array(1) {
  ["_one"]=>
  string(1) "1"
}
----------------------------------------
proxy::moreResults(array (
  0 => NULL,
))
proxy::moreResults returns true
proxy::moreResults(array (
  0 => NULL,
))
proxy::moreResults returns true
array(1) {
  ["_two"]=>
  string(1) "2"
}
----------------------------------------
proxy::moreResults(array (
  0 => NULL,
))
proxy::moreResults returns false
```

See Also

`mysqlnd_uh_set_connection_proxy`
`mysqli_more_results`

9.7.28 `MysqlndUhConnection::nextResult`

- `MysqlndUhConnection::nextResult`

Prepare next result from multi_query

Description

```
public bool MysqlndUhConnection::nextResult(
  mysqlnd_connection connection);
```

Prepare next result from multi_query.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.35 MysqlndUhConnection::nextResult example

```
<?php
class proxy extends MysqlndUhConnection {
 public function nextResult($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::nextResult($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->multi_query("SELECT 1 AS _one; SELECT 2 AS _two");
do {
  $res = $mysqli->store_result();
  var_dump($res->fetch_assoc());
  printf("%s\n", str_repeat("-", 40));
} while ($mysqli->more_results() && $mysqli->next_result());
?>
```

The above example will output:

```
array(1) {
  ["_one"]=>
  string(1) "1"
}
----------------------------------------
proxy::nextResult(array (
  0 => NULL,
))
proxy::nextResult returns true
array(1) {
  ["_two"]=>
  string(1) "2"
}
----------------------------------------
```

```
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_next_result

9.7.29 MysqlndUhConnection::ping

- MysqlndUhConnection::ping

  Pings a server connection, or tries to reconnect if the connection has gone down

Description

```
public bool MysqlndUhConnection::ping(
  mysqlnd_connection connection);
```

Pings a server connection, or tries to reconnect if the connection has gone down.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.36 MysqlndUhConnection::ping example

```
<?php
class proxy extends MysqlndUhConnection {
 public function ping($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::ping($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->ping();
?>
```

The above example will output:

```
proxy::ping(array (
  0 => NULL,
))
proxy::ping returns true
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_ping
mysql_ping

9.7.30 MysqlndUhConnection::query

- MysqlndUhConnection::query

  Performs a query on the database

Description

```
public bool MysqlndUhConnection::query(
  mysqlnd_connection connection,
  string query);
```

Performs a query on the database (COM_QUERY).

Parameters

connection	Mysqlnd connection handle. Do not modify!
query	The query string.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.37 MysqlndUhConnection::query example

```
<?php
class proxy extends MysqlndUhConnection {
 public function query($res, $query) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $query = "SELECT 'How about query rewriting?'";
  $ret = parent::query($res, $query);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$res = $mysqli->query("SELECT 'Welcome mysqlnd_uh!' FROM DUAL");
var_dump($res->fetch_assoc());
?>
```

The above example will output:

```
proxy::query(array (
```

```
  0 => NULL,
  1 => 'SELECT \'Welcome mysqlnd_uh!\' FROM DUAL',
))
proxy::query returns true
array(1) {
  ["How about query rewriting?"]=>
  string(26) "How about query rewriting?"
}
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_query
mysql_query

9.7.31 MysqlndUhConnection::queryReadResultsetHeader

- MysqlndUhConnection::queryReadResultsetHeader

  Read a result set header

Description

```
  public bool MysqlndUhConnection::queryReadResultsetHeader(
    mysqlnd_connection connection,
    mysqlnd_statement mysqlnd_stmt);
```

Read a result set header.

Parameters

connection	Mysqlnd connection handle. Do not modify!
mysqlnd_stmt	Mysqlnd statement handle. Do not modify! Set to NULL, if function is not used in the context of a prepared statement.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.38 MysqlndUhConnection::queryReadResultsetHeader example

```
<?php
class proxy extends MysqlndUhConnection {
 public function queryReadResultsetHeader($res, $stmt) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::queryReadResultsetHeader($res, $stmt);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
```

```
$res = $mysqli->query("SELECT 'Welcome mysqlnd_uh!' FROM DUAL");
var_dump($res->fetch_assoc());
?>
```

The above example will output:

```
proxy::queryReadResultsetHeader(array (
  0 => NULL,
  1 => NULL,
))
proxy::queryReadResultsetHeader returns true
array(1) {
  ["Welcome mysqlnd_uh!"]=>
  string(19) "Welcome mysqlnd_uh!"
}
```

See Also

`mysqlnd_uh_set_connection_proxy`

9.7.32 `MysqlndUhConnection::reapQuery`

- `MysqlndUhConnection::reapQuery`

  Get result from async query

Description

```
  public bool MysqlndUhConnection::reapQuery(
    mysqlnd_connection connection);
```

Get result from async query.

Parameters

`connection`	Mysqlnd connection handle. Do not modify!

Return Values

Returns `TRUE` on success. Otherwise, returns `FALSE`

Examples

Example 9.39 `MysqlndUhConnection::reapQuery` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function reapQuery($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::reapQuery($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
```

```
}
mysqlnd_uh_set_connection_proxy(new proxy());

$conn1 = new mysqli("localhost", "root", "", "test");
$conn2 = new mysqli("localhost", "root", "", "test");

$conn1->query("SELECT 1 as 'one', SLEEP(1) AS _sleep FROM DUAL", MYSQLI_ASYNC |  MYSQLI_USE_RESULT);
$conn2->query("SELECT 1.1 as 'one dot one' FROM DUAL", MYSQLI_ASYNC |  MYSQLI_USE_RESULT);

$links = array(
 $conn1->thread_id => array('link' => $conn1, 'processed' => false),
 $conn2->thread_id => array('link' => $conn2, 'processed' => false)
);

$saved_errors = array();
do {
 $poll_links = $poll_errors = $poll_reject = array();
 foreach ($links as $thread_id => $link) {
  if (!$link['processed']) {
   $poll_links[] = $link['link'];
   $poll_errors[] = $link['link'];
   $poll_reject[] = $link['link'];
  }
 }
 if (0 == count($poll_links))
  break;

 if (0 == ($num_ready = mysqli_poll($poll_links, $poll_errors, $poll_reject, 0, 200000)))
  continue;

 if (!empty($poll_errors)) {
  die(var_dump($poll_errors));
 }

 foreach ($poll_links as $link) {
  $thread_id = mysqli_thread_id($link);
  $links[$thread_id]['processed'] = true;

  if (is_object($res = mysqli_reap_async_query($link))) {
   // result set object
   while ($row = mysqli_fetch_assoc($res)) {
    // eat up all results
    var_dump($row);
   }
   mysqli_free_result($res);
  } else {
   // either there is no result (no SELECT) or there is an error
   if (mysqli_errno($link) > 0) {
    $saved_errors[$thread_id] = mysqli_errno($link);
    printf("'%s' caused %d\n", $links[$thread_id]['query'],     mysqli_errno($link));
   }
  }
 }
} while (true);
?>
```

The above example will output:

```
proxy::reapQuery(array (
  0 => NULL,
))
proxy::reapQuery returns true
array(1) {
```

```
  ["one dot one"]=>
  string(3) "1.1"
}
proxy::reapQuery(array (
  0 => NULL,
))
proxy::reapQuery returns true
array(2) {
  ["one"]=>
  string(1) "1"
  ["_sleep"]=>
  string(1) "0"
}
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_real_async_query

9.7.33 MysqlndUhConnection::refreshServer

- MysqlndUhConnection::refreshServer

  Flush or reset tables and caches

Description

```
  public bool MysqlndUhConnection::refreshServer(
    mysqlnd_connection connection,
    int options);
```

Flush or reset tables and caches.

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

connection	Mysqlnd connection handle. Do not modify!
options	What to refresh.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.40 MysqlndUhConnection::refreshServer example

```
<?php
class proxy extends MysqlndUhConnection {
 public function refreshServer($res, $option) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::refreshServer($res, $option);
```

```
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
mysqli_refresh($mysqli, 1);
?>
```

The above example will output:

```
proxy::refreshServer(array (
  0 => NULL,
  1 => 1,
))
proxy::refreshServer returns false
```

See Also

mysqlnd_uh_set_connection_proxy

9.7.34 MysqlndUhConnection::restartPSession

- MysqlndUhConnection::restartPSession

  Restart a persistent mysqlnd connection

Description

```
  public bool MysqlndUhConnection::restartPSession(
    mysqlnd_connection connection);
```

Restart a persistent mysqlnd connection.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.41 MysqlndUhConnection::restartPSession example

```
<?php
class proxy extends MysqlndUhConnection {
 public function ping($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::ping($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
```

```
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->ping();
?>
```

The above example will output:

```
proxy::restartPSession(array (
  0 => NULL,
))
proxy::restartPSession returns true
```

See Also

mysqlnd_uh_set_connection_proxy

9.7.35 MysqlndUhConnection::selectDb

- MysqlndUhConnection::selectDb

  Selects the default database for database queries

Description

```
  public bool MysqlndUhConnection::selectDb(
    mysqlnd_connection connection,
    string database);
```

Selects the default database for database queries.

Parameters

connection	Mysqlnd connection handle. Do not modify!
database	The database name.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.42 MysqlndUhConnection::selectDb example

```
<?php
class proxy extends MysqlndUhConnection {
 public function selectDb($res, $database) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::selectDb($res, $database);
```

```
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->select_db("mysql");
?>
```

The above example will output:

```
proxy::selectDb(array (
  0 => NULL,
  1 => 'mysql',
))
proxy::selectDb returns true
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_select_db
mysql_select_db

9.7.36 MysqlndUhConnection::sendClose

- MysqlndUhConnection::sendClose

  Sends a close command to MySQL

Description

```
  public bool MysqlndUhConnection::sendClose(
    mysqlnd_connection connection);
```

Sends a close command to MySQL.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.43 MysqlndUhConnection::sendClose example

```
<?php
class proxy extends MysqlndUhConnection {
 public function sendClose($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
```

```
  $ret = parent::sendClose($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->close();
?>
```

The above example will output:

```
proxy::sendClose(array (
  0 => NULL,
))
proxy::sendClose returns true
proxy::sendClose(array (
  0 => NULL,
))
proxy::sendClose returns true
```

See Also

mysqlnd_uh_set_connection_proxy

9.7.37 MysqlndUhConnection::sendQuery

- MysqlndUhConnection::sendQuery

  Sends a query to MySQL

Description

```
  public bool MysqlndUhConnection::sendQuery(
    mysqlnd_connection connection,
    string query);
```

Sends a query to MySQL.

Parameters

connection	Mysqlnd connection handle. Do not modify!
query	The query string.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.44 MysqlndUhConnection::sendQuery example

```
<?php
class proxy extends MysqlndUhConnection {
 public function sendQuery($res, $query) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::sendQuery($res, $query);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->query("SELECT 1");
?>
```

The above example will output:

```
proxy::sendQuery(array (
  0 => NULL,
  1 => 'SELECT 1',
))
proxy::sendQuery returns true
```

See Also

mysqlnd_uh_set_connection_proxy

9.7.38 MysqlndUhConnection::serverDumpDebugInformation

- MysqlndUhConnection::serverDumpDebugInformation

  Dump debugging information into the log for the MySQL server

Description

```
  public bool MysqlndUhConnection::serverDumpDebugInformation(
    mysqlnd_connection connection);
```

Dump debugging information into the log for the MySQL server.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.45 MysqlndUhConnection::serverDumpDebugInformation example

```
<?php
```

```
class proxy extends MysqlndUhConnection {
 public function serverDumpDebugInformation($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::serverDumpDebugInformation($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->dump_debug_info();
?>
```

The above example will output:

```
proxy::serverDumpDebugInformation(array (
  0 => NULL,
))
proxy::serverDumpDebugInformation returns true
```

See Also

`mysqlnd_uh_set_connection_proxy`
`mysqli_dump_debug_info`

9.7.39 `MysqlndUhConnection::setAutocommit`

- `MysqlndUhConnection::setAutocommit`

  Turns on or off auto-committing database modifications

Description

```
  public bool MysqlndUhConnection::setAutocommit(
    mysqlnd_connection connection,
    int mode);
```

Turns on or off auto-committing database modifications

Parameters

`connection`	Mysqlnd connection handle. Do not modify!
`mode`	Whether to turn on auto-commit or not.

Return Values

Returns `TRUE` on success. Otherwise, returns `FALSE`

Examples

Example 9.46 `MysqlndUhConnection::setAutocommit` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function setAutocommit($res, $mode) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::setAutocommit($res, $mode);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->autocommit(false);
$mysqli->autocommit(true);
?>
```

The above example will output:

```
proxy::setAutocommit(array (
  0 => NULL,
  1 => 0,
))
proxy::setAutocommit returns true
proxy::setAutocommit(array (
  0 => NULL,
  1 => 1,
))
proxy::setAutocommit returns true
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_autocommit

9.7.40 MysqlndUhConnection::setCharset

- MysqlndUhConnection::setCharset

  Sets the default client character set

Description

```
  public bool MysqlndUhConnection::setCharset(
    mysqlnd_connection connection,
    string charset);
```

Sets the default client character set.

Parameters

connection	Mysqlnd connection handle. Do not modify!
charset	The charset to be set as default.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.47 `MysqlndUhConnection::setCharset` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function setCharset($res, $charset) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::setCharset($res, $charset);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->set_charset("latin1");
?>
```

The above example will output:

```
proxy::setCharset(array (
  0 => NULL,
  1 => 'latin1',
))
proxy::setCharset returns true
```

See Also

`mysqlnd_uh_set_connection_proxy`
`mysqli_set_charset`

9.7.41 `MysqlndUhConnection::setClientOption`

- `MysqlndUhConnection::setClientOption`

  Sets a client option

Description

```
  public bool MysqlndUhConnection::setClientOption(
    mysqlnd_connection connection,
    int option,
    int value);
```

Sets a client option.

Parameters

`connection`	Mysqlnd connection handle. Do not modify!
`option`	The option to be set.

value Optional option value, if required.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.48 MysqlndUhConnection::setClientOption example

```
<?php
function client_option_to_string($option) {
 static $mapping = array(
  MYSQLND_UH_MYSQLND_OPTION_OPT_CONNECT_TIMEOUT => "MYSQLND_UH_MYSQLND_OPTION_OPT_CONNECT_TIMEOUT",
  MYSQLND_UH_MYSQLND_OPTION_OPT_COMPRESS => "MYSQLND_UH_MYSQLND_OPTION_OPT_COMPRESS",
  MYSQLND_UH_MYSQLND_OPTION_OPT_NAMED_PIPE => "MYSQLND_UH_MYSQLND_OPTION_OPT_NAMED_PIPE",
  MYSQLND_UH_MYSQLND_OPTION_INIT_COMMAND => "MYSQLND_UH_MYSQLND_OPTION_INIT_COMMAND",
  MYSQLND_UH_MYSQLND_READ_DEFAULT_FILE => "MYSQLND_UH_MYSQLND_READ_DEFAULT_FILE",
  MYSQLND_UH_MYSQLND_READ_DEFAULT_GROUP => "MYSQLND_UH_MYSQLND_READ_DEFAULT_GROUP",
  MYSQLND_UH_MYSQLND_SET_CHARSET_DIR => "MYSQLND_UH_MYSQLND_SET_CHARSET_DIR",
  MYSQLND_UH_MYSQLND_SET_CHARSET_NAME => "MYSQLND_UH_MYSQLND_SET_CHARSET_NAME",
  MYSQLND_UH_MYSQLND_OPT_LOCAL_INFILE => "MYSQLND_UH_MYSQLND_OPT_LOCAL_INFILE",
  MYSQLND_UH_MYSQLND_OPT_PROTOCOL => "MYSQLND_UH_MYSQLND_OPT_PROTOCOL",
  MYSQLND_UH_MYSQLND_SHARED_MEMORY_BASE_NAME => "MYSQLND_UH_MYSQLND_SHARED_MEMORY_BASE_NAME",
  MYSQLND_UH_MYSQLND_OPT_READ_TIMEOUT => "MYSQLND_UH_MYSQLND_OPT_READ_TIMEOUT",
  MYSQLND_UH_MYSQLND_OPT_WRITE_TIMEOUT => "MYSQLND_UH_MYSQLND_OPT_WRITE_TIMEOUT",
  MYSQLND_UH_MYSQLND_OPT_USE_RESULT => "MYSQLND_UH_MYSQLND_OPT_USE_RESULT",
  MYSQLND_UH_MYSQLND_OPT_USE_REMOTE_CONNECTION => "MYSQLND_UH_MYSQLND_OPT_USE_REMOTE_CONNECTION",
  MYSQLND_UH_MYSQLND_OPT_USE_EMBEDDED_CONNECTION => "MYSQLND_UH_MYSQLND_OPT_USE_EMBEDDED_CONNECTION",
  MYSQLND_UH_MYSQLND_OPT_GUESS_CONNECTION => "MYSQLND_UH_MYSQLND_OPT_GUESS_CONNECTION",
  MYSQLND_UH_MYSQLND_SET_CLIENT_IP => "MYSQLND_UH_MYSQLND_SET_CLIENT_IP",
  MYSQLND_UH_MYSQLND_SECURE_AUTH => "MYSQLND_UH_MYSQLND_SECURE_AUTH",
  MYSQLND_UH_MYSQLND_REPORT_DATA_TRUNCATION => "MYSQLND_UH_MYSQLND_REPORT_DATA_TRUNCATION",
  MYSQLND_UH_MYSQLND_OPT_RECONNECT => "MYSQLND_UH_MYSQLND_OPT_RECONNECT",
  MYSQLND_UH_MYSQLND_OPT_SSL_VERIFY_SERVER_CERT => "MYSQLND_UH_MYSQLND_OPT_SSL_VERIFY_SERVER_CERT",
  MYSQLND_UH_MYSQLND_OPT_NET_CMD_BUFFER_SIZE => "MYSQLND_UH_MYSQLND_OPT_NET_CMD_BUFFER_SIZE",
  MYSQLND_UH_MYSQLND_OPT_NET_READ_BUFFER_SIZE => "MYSQLND_UH_MYSQLND_OPT_NET_READ_BUFFER_SIZE",
  MYSQLND_UH_MYSQLND_OPT_SSL_KEY => "MYSQLND_UH_MYSQLND_OPT_SSL_KEY",
  MYSQLND_UH_MYSQLND_OPT_SSL_CERT => "MYSQLND_UH_MYSQLND_OPT_SSL_CERT",
  MYSQLND_UH_MYSQLND_OPT_SSL_CA => "MYSQLND_UH_MYSQLND_OPT_SSL_CA",
  MYSQLND_UH_MYSQLND_OPT_SSL_CAPATH => "MYSQLND_UH_MYSQLND_OPT_SSL_CAPATH",
  MYSQLND_UH_MYSQLND_OPT_SSL_CIPHER => "MYSQLND_UH_MYSQLND_OPT_SSL_CIPHER",
  MYSQLND_UH_MYSQLND_OPT_SSL_PASSPHRASE => "MYSQLND_UH_MYSQLND_OPT_SSL_PASSPHRASE",
  MYSQLND_UH_SERVER_OPTION_PLUGIN_DIR => "MYSQLND_UH_SERVER_OPTION_PLUGIN_DIR",
  MYSQLND_UH_SERVER_OPTION_DEFAULT_AUTH => "MYSQLND_UH_SERVER_OPTION_DEFAULT_AUTH",
  MYSQLND_UH_SERVER_OPTION_SET_CLIENT_IP => "MYSQLND_UH_SERVER_OPTION_SET_CLIENT_IP"
 );
 if (version_compare(PHP_VERSION, '5.3.99-dev', '>')) {
  $mapping[MYSQLND_UH_MYSQLND_OPT_MAX_ALLOWED_PACKET] = "MYSQLND_UH_MYSQLND_OPT_MAX_ALLOWED_PACKET";
  $mapping[MYSQLND_UH_MYSQLND_OPT_AUTH_PROTOCOL] = "MYSQLND_UH_MYSQLND_OPT_AUTH_PROTOCOL";
 }
 if (defined("MYSQLND_UH_MYSQLND_OPT_INT_AND_FLOAT_NATIVE")) {
  /* special mysqlnd build */
  $mapping["MYSQLND_UH_MYSQLND_OPT_INT_AND_FLOAT_NATIVE"] = "MYSQLND_UH_MYSQLND_OPT_INT_AND_FLOAT_NATIVE";
 }
 return (isset($mapping[$option])) ? $mapping[$option] : 'unknown';
}

class proxy extends MysqlndUhConnection {
 public function setClientOption($res, $option, $value) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  printf("Option '%s' set to %s\n", client_option_to_string($option), var_export($value, true));
  $ret = parent::setClientOption($res, $option, $value);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
```

```
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
?>
```

The above example will output:

```
proxy::setClientOption(array (
  0 => NULL,
  1 => 210,
  2 => 3221225472,
))
Option 'MYSQLND_UH_MYSQLND_OPT_MAX_ALLOWED_PACKET' set to 3221225472
proxy::setClientOption returns true
proxy::setClientOption(array (
  0 => NULL,
  1 => 211,
  2 => 'mysql_native_password',
))
Option 'MYSQLND_UH_MYSQLND_OPT_AUTH_PROTOCOL' set to 'mysql_native_password'
proxy::setClientOption returns true
proxy::setClientOption(array (
  0 => NULL,
  1 => 8,
  2 => 1,
))
Option 'MYSQLND_UH_MYSQLND_OPT_LOCAL_INFILE' set to 1
proxy::setClientOption returns true
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_real_connect
mysqli_options

9.7.42 MysqlndUhConnection::setServerOption

- MysqlndUhConnection::setServerOption

  Sets a server option

Description

```
  public void MysqlndUhConnection::setServerOption(
    mysqlnd_connection connection,
    int option);
```

Sets a server option.

Parameters

connection	Mysqlnd connection handle. Do not modify!
option	The option to be set.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.49 MysqlndUhConnection::setServerOption example

```
<?php
function server_option_to_string($option) {
 $ret = 'unknown';
 switch ($option) {
  case MYSQLND_UH_SERVER_OPTION_MULTI_STATEMENTS_ON:
   $ret = 'MYSQLND_UH_SERVER_OPTION_MULTI_STATEMENTS_ON';
   break;
  case MYSQLND_UH_SERVER_OPTION_MULTI_STATEMENTS_OFF:
   $ret = 'MYSQLND_UH_SERVER_OPTION_MULTI_STATEMENTS_ON';
   break;
 }
 return $ret;
}

class proxy extends MysqlndUhConnection {
 public function setServerOption($res, $option) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  printf("Option '%s' set\n", server_option_to_string($option));
  $ret = parent::setServerOption($res, $option);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->multi_query("SELECT 1; SELECT 2");
?>
```

The above example will output:

```
proxy::setServerOption(array (
  0 => NULL,
  1 => 0,
))
Option 'MYSQLND_UH_SERVER_OPTION_MULTI_STATEMENTS_ON' set
proxy::setServerOption returns true
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_real_connect
mysqli_options
mysqli_multi_query

9.7.43 MysqlndUhConnection::shutdownServer

- `MysqlndUhConnection::shutdownServer`

  The shutdownServer purpose

Description

```
public void MysqlndUhConnection::shutdownServer(
  string MYSQLND_UH_RES_MYSQLND_NAME,
  string "level");
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

`MYSQLND_UH_RES_MYSQLND_NAME`

`"level"`

Return Values

9.7.44 `MysqlndUhConnection::simpleCommand`

- `MysqlndUhConnection::simpleCommand`

  Sends a basic COM_* command

Description

```
public bool MysqlndUhConnection::simpleCommand(
  mysqlnd_connection connection,
  int command,
  string arg,
  int ok_packet,
  bool silent,
  bool ignore_upsert_status);
```

Sends a basic COM_* command to MySQL.

Parameters

`connection`	Mysqlnd connection handle. Do not modify!
`command`	The COM command to be send.
`arg`	Optional COM command arguments.
`ok_packet`	The OK packet type.
`silent`	Whether mysqlnd may emit errors.
`ignore_upsert_status`	Whether to ignore `UPDATE`/`INSERT` status.

Return Values

Returns `TRUE` on success. Otherwise, returns `FALSE`

Examples

Example 9.50 `MysqlndUhConnection::simpleCommand` example

```
<?php
function server_cmd_2_string($command) {
 $mapping = array(
  MYSQLND_UH_MYSQLND_COM_SLEEP => "MYSQLND_UH_MYSQLND_COM_SLEEP",
  MYSQLND_UH_MYSQLND_COM_QUIT => "MYSQLND_UH_MYSQLND_COM_QUIT",
  MYSQLND_UH_MYSQLND_COM_INIT_DB => "MYSQLND_UH_MYSQLND_COM_INIT_DB",
  MYSQLND_UH_MYSQLND_COM_QUERY => "MYSQLND_UH_MYSQLND_COM_QUERY",
  MYSQLND_UH_MYSQLND_COM_FIELD_LIST => "MYSQLND_UH_MYSQLND_COM_FIELD_LIST",
  MYSQLND_UH_MYSQLND_COM_CREATE_DB => "MYSQLND_UH_MYSQLND_COM_CREATE_DB",
  MYSQLND_UH_MYSQLND_COM_DROP_DB => "MYSQLND_UH_MYSQLND_COM_DROP_DB",
  MYSQLND_UH_MYSQLND_COM_REFRESH => "MYSQLND_UH_MYSQLND_COM_REFRESH",
  MYSQLND_UH_MYSQLND_COM_SHUTDOWN => "MYSQLND_UH_MYSQLND_COM_SHUTDOWN",
  MYSQLND_UH_MYSQLND_COM_STATISTICS => "MYSQLND_UH_MYSQLND_COM_STATISTICS",
  MYSQLND_UH_MYSQLND_COM_PROCESS_INFO => "MYSQLND_UH_MYSQLND_COM_PROCESS_INFO",
  MYSQLND_UH_MYSQLND_COM_CONNECT => "MYSQLND_UH_MYSQLND_COM_CONNECT",
  MYSQLND_UH_MYSQLND_COM_PROCESS_KILL => "MYSQLND_UH_MYSQLND_COM_PROCESS_KILL",
  MYSQLND_UH_MYSQLND_COM_DEBUG => "MYSQLND_UH_MYSQLND_COM_DEBUG",
  MYSQLND_UH_MYSQLND_COM_PING => "MYSQLND_UH_MYSQLND_COM_PING",
  MYSQLND_UH_MYSQLND_COM_TIME => "MYSQLND_UH_MYSQLND_COM_TIME",
  MYSQLND_UH_MYSQLND_COM_DELAYED_INSERT => "MYSQLND_UH_MYSQLND_COM_DELAYED_INSERT",
  MYSQLND_UH_MYSQLND_COM_CHANGE_USER => "MYSQLND_UH_MYSQLND_COM_CHANGE_USER",
  MYSQLND_UH_MYSQLND_COM_BINLOG_DUMP => "MYSQLND_UH_MYSQLND_COM_BINLOG_DUMP",
  MYSQLND_UH_MYSQLND_COM_TABLE_DUMP => "MYSQLND_UH_MYSQLND_COM_TABLE_DUMP",
  MYSQLND_UH_MYSQLND_COM_CONNECT_OUT => "MYSQLND_UH_MYSQLND_COM_CONNECT_OUT",
  MYSQLND_UH_MYSQLND_COM_REGISTER_SLAVED => "MYSQLND_UH_MYSQLND_COM_REGISTER_SLAVED",
  MYSQLND_UH_MYSQLND_COM_STMT_PREPARE => "MYSQLND_UH_MYSQLND_COM_STMT_PREPARE",
  MYSQLND_UH_MYSQLND_COM_STMT_EXECUTE => "MYSQLND_UH_MYSQLND_COM_STMT_EXECUTE",
  MYSQLND_UH_MYSQLND_COM_STMT_SEND_LONG_DATA => "MYSQLND_UH_MYSQLND_COM_STMT_SEND_LONG_DATA",
  MYSQLND_UH_MYSQLND_COM_STMT_CLOSE => "MYSQLND_UH_MYSQLND_COM_STMT_CLOSE",
  MYSQLND_UH_MYSQLND_COM_STMT_RESET => "MYSQLND_UH_MYSQLND_COM_STMT_RESET",
  MYSQLND_UH_MYSQLND_COM_SET_OPTION => "MYSQLND_UH_MYSQLND_COM_SET_OPTION",
  MYSQLND_UH_MYSQLND_COM_STMT_FETCH => "MYSQLND_UH_MYSQLND_COM_STMT_FETCH",
  MYSQLND_UH_MYSQLND_COM_DAEMON => "MYSQLND_UH_MYSQLND_COM_DAEMON",
  MYSQLND_UH_MYSQLND_COM_END => "MYSQLND_UH_MYSQLND_COM_END",
 );
 return (isset($mapping[$command])) ? $mapping[$command] : 'unknown';
}

function ok_packet_2_string($ok_packet) {
 $mapping = array(
  MYSQLND_UH_MYSQLND_PROT_GREET_PACKET => "MYSQLND_UH_MYSQLND_PROT_GREET_PACKET",
  MYSQLND_UH_MYSQLND_PROT_AUTH_PACKET => "MYSQLND_UH_MYSQLND_PROT_AUTH_PACKET",
  MYSQLND_UH_MYSQLND_PROT_OK_PACKET => "MYSQLND_UH_MYSQLND_PROT_OK_PACKET",
  MYSQLND_UH_MYSQLND_PROT_EOF_PACKET => "MYSQLND_UH_MYSQLND_PROT_EOF_PACKET",
  MYSQLND_UH_MYSQLND_PROT_CMD_PACKET => "MYSQLND_UH_MYSQLND_PROT_CMD_PACKET",
  MYSQLND_UH_MYSQLND_PROT_RSET_HEADER_PACKET => "MYSQLND_UH_MYSQLND_PROT_RSET_HEADER_PACKET",
  MYSQLND_UH_MYSQLND_PROT_RSET_FLD_PACKET => "MYSQLND_UH_MYSQLND_PROT_RSET_FLD_PACKET",
  MYSQLND_UH_MYSQLND_PROT_ROW_PACKET => "MYSQLND_UH_MYSQLND_PROT_ROW_PACKET",
  MYSQLND_UH_MYSQLND_PROT_STATS_PACKET => "MYSQLND_UH_MYSQLND_PROT_STATS_PACKET",
  MYSQLND_UH_MYSQLND_PREPARE_RESP_PACKET => "MYSQLND_UH_MYSQLND_PREPARE_RESP_PACKET",
  MYSQLND_UH_MYSQLND_CHG_USER_RESP_PACKET => "MYSQLND_UH_MYSQLND_CHG_USER_RESP_PACKET",
  MYSQLND_UH_MYSQLND_PROT_LAST => "MYSQLND_UH_MYSQLND_PROT_LAST",
 );
 return (isset($mapping[$ok_packet])) ? $mapping[$ok_packet] : 'unknown';
}

class proxy extends MysqlndUhConnection {
 public function simpleCommand($conn, $command, $arg, $ok_packet, $silent, $ignore_upsert_status) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  printf("Command '%s'\n", server_cmd_2_string($command));
  printf("OK packet '%s'\n",  ok_packet_2_string($ok_packet));
```

```
  $ret = parent::simpleCommand($conn, $command, $arg, $ok_packet, $silent, $ignore_upsert_status);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->query("SELECT 1");
?>
```

The above example will output:

```
proxy::simpleCommand(array (
  0 => NULL,
  1 => 3,
  2 => 'SELECT 1',
  3 => 13,
  4 => false,
  5 => false,
))
Command 'MYSQLND_UH_MYSQLND_COM_QUERY'
OK packet 'MYSQLND_UH_MYSQLND_PROT_LAST'
proxy::simpleCommand returns true
:)proxy::simpleCommand(array (
  0 => NULL,
  1 => 1,
  2 => '',
  3 => 13,
  4 => true,
  5 => true,
))
Command 'MYSQLND_UH_MYSQLND_COM_QUIT'
OK packet 'MYSQLND_UH_MYSQLND_PROT_LAST'
proxy::simpleCommand returns true
```

See Also

mysqlnd_uh_set_connection_proxy

9.7.45 MysqlndUhConnection::simpleCommandHandleResponse

- MysqlndUhConnection::simpleCommandHandleResponse

  Process a response for a basic COM_* command send to the client

Description

```
  public bool MysqlndUhConnection::simpleCommandHandleResponse(
    mysqlnd_connection connection,
    int ok_packet,
    bool silent,
    int command,
    bool ignore_upsert_status);
```

Process a response for a basic COM_* command send to the client.

Parameters

connection	Mysqlnd connection handle. Do not modify!
ok_packet	The OK packet type.
silent	Whether mysqlnd may emit errors.
command	The COM command to process results from.
ignore_upsert_status	Whether to ignore UPDATE/INSERT status.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.51 MysqlndUhConnection::simpleCommandHandleResponse example

```
<?php
function server_cmd_2_string($command) {
 $mapping = array(
  MYSQLND_UH_MYSQLND_COM_SLEEP => "MYSQLND_UH_MYSQLND_COM_SLEEP",
  MYSQLND_UH_MYSQLND_COM_QUIT => "MYSQLND_UH_MYSQLND_COM_QUIT",
  MYSQLND_UH_MYSQLND_COM_INIT_DB => "MYSQLND_UH_MYSQLND_COM_INIT_DB",
  MYSQLND_UH_MYSQLND_COM_QUERY => "MYSQLND_UH_MYSQLND_COM_QUERY",
  MYSQLND_UH_MYSQLND_COM_FIELD_LIST => "MYSQLND_UH_MYSQLND_COM_FIELD_LIST",
  MYSQLND_UH_MYSQLND_COM_CREATE_DB => "MYSQLND_UH_MYSQLND_COM_CREATE_DB",
  MYSQLND_UH_MYSQLND_COM_DROP_DB => "MYSQLND_UH_MYSQLND_COM_DROP_DB",
  MYSQLND_UH_MYSQLND_COM_REFRESH => "MYSQLND_UH_MYSQLND_COM_REFRESH",
  MYSQLND_UH_MYSQLND_COM_SHUTDOWN => "MYSQLND_UH_MYSQLND_COM_SHUTDOWN",
  MYSQLND_UH_MYSQLND_COM_STATISTICS => "MYSQLND_UH_MYSQLND_COM_STATISTICS",
  MYSQLND_UH_MYSQLND_COM_PROCESS_INFO => "MYSQLND_UH_MYSQLND_COM_PROCESS_INFO",
  MYSQLND_UH_MYSQLND_COM_CONNECT => "MYSQLND_UH_MYSQLND_COM_CONNECT",
  MYSQLND_UH_MYSQLND_COM_PROCESS_KILL => "MYSQLND_UH_MYSQLND_COM_PROCESS_KILL",
  MYSQLND_UH_MYSQLND_COM_DEBUG => "MYSQLND_UH_MYSQLND_COM_DEBUG",
  MYSQLND_UH_MYSQLND_COM_PING => "MYSQLND_UH_MYSQLND_COM_PING",
  MYSQLND_UH_MYSQLND_COM_TIME => "MYSQLND_UH_MYSQLND_COM_TIME",
  MYSQLND_UH_MYSQLND_COM_DELAYED_INSERT => "MYSQLND_UH_MYSQLND_COM_DELAYED_INSERT",
  MYSQLND_UH_MYSQLND_COM_CHANGE_USER => "MYSQLND_UH_MYSQLND_COM_CHANGE_USER",
  MYSQLND_UH_MYSQLND_COM_BINLOG_DUMP => "MYSQLND_UH_MYSQLND_COM_BINLOG_DUMP",
  MYSQLND_UH_MYSQLND_COM_TABLE_DUMP => "MYSQLND_UH_MYSQLND_COM_TABLE_DUMP",
  MYSQLND_UH_MYSQLND_COM_CONNECT_OUT => "MYSQLND_UH_MYSQLND_COM_CONNECT_OUT",
  MYSQLND_UH_MYSQLND_COM_REGISTER_SLAVED => "MYSQLND_UH_MYSQLND_COM_REGISTER_SLAVED",
  MYSQLND_UH_MYSQLND_COM_STMT_PREPARE => "MYSQLND_UH_MYSQLND_COM_STMT_PREPARE",
  MYSQLND_UH_MYSQLND_COM_STMT_EXECUTE => "MYSQLND_UH_MYSQLND_COM_STMT_EXECUTE",
  MYSQLND_UH_MYSQLND_COM_STMT_SEND_LONG_DATA => "MYSQLND_UH_MYSQLND_COM_STMT_SEND_LONG_DATA",
  MYSQLND_UH_MYSQLND_COM_STMT_CLOSE => "MYSQLND_UH_MYSQLND_COM_STMT_CLOSE",
  MYSQLND_UH_MYSQLND_COM_STMT_RESET => "MYSQLND_UH_MYSQLND_COM_STMT_RESET",
  MYSQLND_UH_MYSQLND_COM_SET_OPTION => "MYSQLND_UH_MYSQLND_COM_SET_OPTION",
  MYSQLND_UH_MYSQLND_COM_STMT_FETCH => "MYSQLND_UH_MYSQLND_COM_STMT_FETCH",
  MYSQLND_UH_MYSQLND_COM_DAEMON => "MYSQLND_UH_MYSQLND_COM_DAEMON",
  MYSQLND_UH_MYSQLND_COM_END => "MYSQLND_UH_MYSQLND_COM_END",
 );
 return (isset($mapping[$command])) ? $mapping[$command] : 'unknown';
}

function ok_packet_2_string($ok_packet) {
 $mapping = array(
  MYSQLND_UH_MYSQLND_PROT_GREET_PACKET => "MYSQLND_UH_MYSQLND_PROT_GREET_PACKET",
  MYSQLND_UH_MYSQLND_PROT_AUTH_PACKET => "MYSQLND_UH_MYSQLND_PROT_AUTH_PACKET",
  MYSQLND_UH_MYSQLND_PROT_OK_PACKET => "MYSQLND_UH_MYSQLND_PROT_OK_PACKET",
  MYSQLND_UH_MYSQLND_PROT_EOF_PACKET => "MYSQLND_UH_MYSQLND_PROT_EOF_PACKET",
  MYSQLND_UH_MYSQLND_PROT_CMD_PACKET => "MYSQLND_UH_MYSQLND_PROT_CMD_PACKET",
```

```
  MYSQLND_UH_MYSQLND_PROT_RSET_HEADER_PACKET => "MYSQLND_UH_MYSQLND_PROT_RSET_HEADER_PACKET",
  MYSQLND_UH_MYSQLND_PROT_RSET_FLD_PACKET => "MYSQLND_UH_MYSQLND_PROT_RSET_FLD_PACKET",
  MYSQLND_UH_MYSQLND_PROT_ROW_PACKET => "MYSQLND_UH_MYSQLND_PROT_ROW_PACKET",
  MYSQLND_UH_MYSQLND_PROT_STATS_PACKET => "MYSQLND_UH_MYSQLND_PROT_STATS_PACKET",
  MYSQLND_UH_MYSQLND_PREPARE_RESP_PACKET => "MYSQLND_UH_MYSQLND_PREPARE_RESP_PACKET",
  MYSQLND_UH_MYSQLND_CHG_USER_RESP_PACKET => "MYSQLND_UH_MYSQLND_CHG_USER_RESP_PACKET",
  MYSQLND_UH_MYSQLND_PROT_LAST => "MYSQLND_UH_MYSQLND_PROT_LAST",
 );
 return (isset($mapping[$ok_packet])) ? $mapping[$ok_packet] : 'unknown';
}

class proxy extends MysqlndUhConnection {
 public function simpleCommandHandleResponse($conn, $ok_packet, $silent, $command, $ignore_upsert_status) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  printf("Command '%s'\n", server_cmd_2_string($command));
  printf("OK packet '%s'\n",  ok_packet_2_string($ok_packet));
  $ret = parent::simpleCommandHandleResponse($conn, $ok_packet, $silent, $command, $ignore_upsert_status);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysql = mysql_connect("localhost", "root", "");
mysql_query("SELECT 1 FROM DUAL", $mysql);
?>
```

The above example will output:

```
proxy::simpleCommandHandleResponse(array (
  0 => NULL,
  1 => 5,
  2 => false,
  3 => 27,
  4 => true,
))
Command 'MYSQLND_UH_MYSQLND_COM_SET_OPTION'
OK packet 'MYSQLND_UH_MYSQLND_PROT_EOF_PACKET'
proxy::simpleCommandHandleResponse returns true
```

See Also

mysqlnd_uh_set_connection_proxy

9.7.46 MysqlndUhConnection::sslSet

- MysqlndUhConnection::sslSet

  Used for establishing secure connections using SSL

Description

```
  public bool MysqlndUhConnection::sslSet(
    mysqlnd_connection connection,
    string key,
    string cert,
    string ca,
```

```
    string capath,
    string cipher);
```

Used for establishing secure connections using SSL.

Parameters

connection	Mysqlnd connection handle. Do not modify!
key	The path name to the key file.
cert	The path name to the certificate file.
ca	The path name to the certificate authority file.
capath	The pathname to a directory that contains trusted SSL CA certificates in PEM format.
cipher	A list of allowable ciphers to use for SSL encryption.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.52 `MysqlndUhConnection::sslSet` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function sslSet($conn, $key, $cert, $ca, $capath, $cipher) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::sslSet($conn, $key, $cert, $ca, $capath, $cipher);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->ssl_set("key", "cert", "ca", "capath", "cipher");
?>
```

The above example will output:

```
proxy::sslSet(array (
  0 => NULL,
  1 => 'key',
  2 => 'cert',
  3 => 'ca',
  4 => 'capath',
  5 => 'cipher',
))
proxy::sslSet returns true
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_ssl_set

9.7.47 MysqlndUhConnection::stmtInit

- MysqlndUhConnection::stmtInit

  Initializes a statement and returns a resource for use with mysqli_statement::prepare

Description

```
public resource MysqlndUhConnection::stmtInit(
  mysqlnd_connection connection);
```

Initializes a statement and returns a resource for use with mysqli_statement::prepare.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Resource of type Mysqlnd Prepared Statement (internal only - you must not modify it!). The documentation may also refer to such resources using the alias name mysqlnd_prepared_statement.

Examples

Example 9.53 MysqlndUhConnection::stmtInit example

```
<?php
class proxy extends MysqlndUhConnection {
 public function stmtInit($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  var_dump($res);
  $ret = parent::stmtInit($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  var_dump($ret);
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());
$mysqli = new mysqli("localhost", "root", "", "test");
$stmt = $mysqli->prepare("SELECT 1 AS _one FROM DUAL");
$stmt->execute();
$one = NULL;
$stmt->bind_result($one);
$stmt->fetch();
var_dump($one);
?>
```

The above example will output:

```
proxy::stmtInit(array (
  0 => NULL,
```

```
))
resource(19) of type (Mysqlnd Connection)
proxy::stmtInit returns NULL
resource(246) of type (Mysqlnd Prepared Statement (internal only - you must not modify it!))
int(1)
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_stmt_init

9.7.48 MysqlndUhConnection::storeResult

- MysqlndUhConnection::storeResult

  Transfers a result set from the last query

Description

```
  public resource MysqlndUhConnection::storeResult(
    mysqlnd_connection connection);
```

Transfers a result set from the last query.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Resource of type Mysqlnd Resultset (internal only - you must not modify it!). The documentation may also refer to such resources using the alias name mysqlnd_resultset.

Examples

Example 9.54 MysqlndUhConnection::storeResult example

```
<?php
class proxy extends MysqlndUhConnection {
 public function storeResult($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::storeResult($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  var_dump($ret);
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$res = $mysqli->query("SELECT 'Also called buffered result' AS _msg FROM DUAL");
var_dump($res->fetch_assoc());

$mysqli->real_query("SELECT 'Good morning!' AS _msg FROM DUAL");
$res = $mysqli->store_result();
var_dump($res->fetch_assoc());
?>
```

```
```

The above example will output:

```
proxy::storeResult(array (
  0 => NULL,
))
proxy::storeResult returns NULL
resource(475) of type (Mysqlnd Resultset (internal only - you must not modify it!))
array(1) {
  ["_msg"]=>
  string(27) "Also called buffered result"
}
proxy::storeResult(array (
  0 => NULL,
))
proxy::storeResult returns NULL
resource(730) of type (Mysqlnd Resultset (internal only - you must not modify it!))
array(1) {
  ["_msg"]=>
  string(13) "Good morning!"
}
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_store_result
mysqli_real_query

9.7.49 MysqlndUhConnection::txCommit

- MysqlndUhConnection::txCommit

  Commits the current transaction

Description

```
  public bool MysqlndUhConnection::txCommit(
    mysqlnd_connection connection);
```

Commits the current transaction.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.55 MysqlndUhConnection::txCommit example

```
```

```
<?php
class proxy extends MysqlndUhConnection {
 public function txCommit($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::txCommit($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->commit();
?>
```

The above example will output:

```
proxy::txCommit(array (
  0 => NULL,
))
proxy::txCommit returns true
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_commit

9.7.50 MysqlndUhConnection::txRollback

- MysqlndUhConnection::txRollback

  Rolls back current transaction

Description

```
  public bool MysqlndUhConnection::txRollback(
    mysqlnd_connection connection);
```

Rolls back current transaction.

Parameters

connection	Mysqlnd connection handle. Do not modify!

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.56 MysqlndUhConnection::txRollback example

```
<?php
class proxy extends MysqlndUhConnection {
 public function txRollback($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::txRollback($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->rollback();
?>
```

The above example will output:

```
proxy::txRollback(array (
  0 => NULL,
))
proxy::txRollback returns true
```

See Also

`mysqlnd_uh_set_connection_proxy`
`mysqli_commit`

9.7.51 `MysqlndUhConnection::useResult`

- `MysqlndUhConnection::useResult`

  Initiate a result set retrieval

Description

```
  public resource MysqlndUhConnection::useResult(
    mysqlnd_connection connection);
```

Initiate a result set retrieval.

Parameters

`connection`	Mysqlnd connection handle. Do not modify!

Return Values

Resource of type `Mysqlnd Resultset (internal only - you must not modify it!)`. The documentation may also refer to such resources using the alias name `mysqlnd_resultset`.

Examples

Example 9.57 `MysqlndUhConnection::useResult` example

```
<?php
class proxy extends MysqlndUhConnection {
 public function useResult($res) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $ret = parent::useResult($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  var_dump($ret);
  return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->real_query("SELECT 'Good morning!' AS _msg FROM DUAL");
$res = $mysqli->use_result();
var_dump($res->fetch_assoc());
?>
```

The above example will output:

```
proxy::useResult(array (
  0 => NULL,
))
proxy::useResult returns NULL
resource(425) of type (Mysqlnd Resultset (internal only - you must not modify it!))
array(1) {
  ["_msg"]=>
  string(13) "Good morning!"
}
```

See Also

mysqlnd_uh_set_connection_proxy
mysqli_use_result
mysqli_real_query

9.8 The MysqlndUhPreparedStatement class


```
MysqlndUhPreparedStatement {
MysqlndUhPreparedStatement

      Methods

  public MysqlndUhPreparedStatement::__construct();

  public bool MysqlndUhPreparedStatement::execute(
    mysqlnd_prepared_statement statement);

  public bool MysqlndUhPreparedStatement::prepare(
    mysqlnd_prepared_statement statement,
    string query);

}
```

9.8.1 MysqlndUhPreparedStatement::__construct

- MysqlndUhPreparedStatement::__construct

  The __construct purpose

Description

```
public MysqlndUhPreparedStatement::__construct();
```

> **Warning**
>
> This function is currently not documented; only its argument list is available.

Parameters

This function has no parameters.

Return Values

9.8.2 MysqlndUhPreparedStatement::execute

- MysqlndUhPreparedStatement::execute

  Executes a prepared Query

Description

```
public bool MysqlndUhPreparedStatement::execute(
  mysqlnd_prepared_statement statement);
```

Executes a prepared Query.

Parameters

statement	Mysqlnd prepared statement handle. Do not modify! Resource of type Mysqlnd Prepared Statement (internal only - you must not modify it!).

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.58 MysqlndUhPreparedStatement::execute example

```
<?php
class stmt_proxy extends MysqlndUhPreparedStatement {
 public function execute($res) {
  printf("%s(", __METHOD__);
  var_dump($res);
```

```
  printf(")\n");
  $ret = parent::execute($res);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  var_dump($ret);
  return $ret;
 }
}
mysqlnd_uh_set_statement_proxy(new stmt_proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$stmt = $mysqli->prepare("SELECT 'Labskaus' AS _msg FROM DUAL");
$stmt->execute();
$msg = NULL;
$stmt->bind_result($msg);
$stmt->fetch();
var_dump($msg);
?>
```

The above example will output:

```
stmt_proxy::execute(resource(256) of type (Mysqlnd Prepared Statement (internal only - you must not modify
)
stmt_proxy::execute returns true
bool(true)
string(8) "Labskaus"
```

See Also

mysqlnd_uh_set_statement_proxy
mysqli_stmt_execute

9.8.3 MysqlndUhPreparedStatement::prepare

- MysqlndUhPreparedStatement::prepare

  Prepare an SQL statement for execution

Description

```
  public bool MysqlndUhPreparedStatement::prepare(
    mysqlnd_prepared_statement statement,
    string query);
```

Prepare an SQL statement for execution.

Parameters

statement	Mysqlnd prepared statement handle. Do not modify! Resource of type Mysqlnd Prepared Statement (internal only - you must not modify it!).
query	The query to be prepared.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.59 `MysqlndUhPreparedStatement::prepare` example

```
<?php
class stmt_proxy extends MysqlndUhPreparedStatement {
 public function prepare($res, $query) {
  printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
  $query = "SELECT 'No more you-know-what-I-mean for lunch, please' AS _msg FROM DUAL";
  $ret = parent::prepare($res, $query);
  printf("%s returns %s\n", __METHOD__, var_export($ret, true));
  var_dump($ret);
  return $ret;
 }
}
mysqlnd_uh_set_statement_proxy(new stmt_proxy());

$mysqli = new mysqli("localhost", "root", "", "test");
$stmt = $mysqli->prepare("SELECT 'Labskaus' AS _msg FROM DUAL");
$stmt->execute();
$msg = NULL;
$stmt->bind_result($msg);
$stmt->fetch();
var_dump($msg);
?>
```

The above example will output:

```
stmt_proxy::prepare(array (
  0 => NULL,
  1 => 'SELECT \'Labskaus\' AS _msg FROM DUAL',
))
stmt_proxy::prepare returns true
bool(true)
string(46) "No more you-know-what-I-mean for lunch, please"
```

See Also

`mysqlnd_uh_set_statement_proxy`
`mysqli_stmt_prepare`
`mysqli_prepare`

9.9 Mysqlnd_uh Functions

9.9.1 `mysqlnd_uh_convert_to_mysqlnd`

- `mysqlnd_uh_convert_to_mysqlnd`

  Converts a MySQL connection handle into a mysqlnd connection handle

Description

```
resource mysqlnd_uh_convert_to_mysqlnd(
  mysqli mysql_connection);
```

Converts a MySQL connection handle into a mysqlnd connection handle. After conversion you can execute mysqlnd library calls on the connection handle. This can be used to access mysqlnd functionality not made available through user space API calls.

The function can be disabled with `mysqlnd_uh.enable`. If `mysqlnd_uh.enable` is set to `FALSE` the function will not install the proxy and always return `TRUE`. Additionally, an error of the type `E_WARNING` may be emitted. The error message may read like `PHP Warning: mysqlnd_uh_convert_to_mysqlnd(): (Mysqlnd User Handler) The plugin has been disabled by setting the configuration parameter mysqlnd_uh.enable = false. You are not allowed to call this function [...]`.

Parameters

`MySQL connection handle` A MySQL connection handle of type mysql, mysqli or PDO_MySQL.

Return Values

A mysqlnd connection handle.

Changelog

Version	Description
5.4.0	The *`mysql_connection`* parameter can now be of type `mysql`, `PDO_MySQL`, or `mysqli`. Before, only the `mysqli` type was allowed.

Examples

Example 9.60 `mysqlnd_uh_convert_to_mysqlnd` example

```
<?php
/* PDO user API gives no access to connection thread id */
$mysql_connection = new PDO("mysql:host=localhost;dbname=test", "root", "");

/* Convert PDO MySQL handle to mysqlnd handle */
$mysqlnd = mysqlnd_uh_convert_to_mysqlnd($mysql_connection);

/* Create Proxy to call mysqlnd connection class methods */
$obj = new MySQLndUHConnection();
/* Call mysqlnd_conn::get_thread_id */
var_dump($obj->getThreadId($mysqlnd));

/* Use SQL to fetch connection thread id */
var_dump($mysql_connection->query("SELECT CONNECTION_ID()")->fetchAll());
?>
```

The above example will output:

```
int(27054)
array(1) {
```

```
  [0]=>
  array(2) {
    ["CONNECTION_ID()"]=>
    string(5) "27054"
    [0]=>
    string(5) "27054"
  }
}
```

See Also

mysqlnd_uh.enable

9.9.2 mysqlnd_uh_set_connection_proxy

- mysqlnd_uh_set_connection_proxy

  Installs a proxy for mysqlnd connections

Description

```
bool mysqlnd_uh_set_connection_proxy(
  MysqlndUhConnection connection_proxy,
  mysqli mysqli_connection);
```

Installs a proxy object to hook mysqlnd's connection objects methods. Once installed, the proxy will be used for all MySQL connections opened with mysqli, mysql or PDO_MYSQL, assuming that the listed extensions are compiled to use the mysqlnd library.

The function can be disabled with mysqlnd_uh.enable. If mysqlnd_uh.enable is set to FALSE the function will not install the proxy and always return TRUE. Additionally, an error of the type E_WARNING may be emitted. The error message may read like PHP Warning: mysqlnd_uh_set_connection_proxy(): (Mysqlnd User Handler) The plugin has been disabled by setting the configuration parameter mysqlnd_uh.enable = false. The proxy has not been installed [...].

Parameters

connection_proxy	A proxy object of type MysqlndUhConnection.
mysqli_connection	Object of type mysqli. If given, the proxy will be set for this particular connection only.

Return Values

Returns TRUE on success. Otherwise, returns FALSE

Examples

Example 9.61 mysqlnd_uh_set_connection_proxy example

```
<?php
$mysqli = new mysqli("localhost", "root", "", "test");
$mysqli->query("SELECT 'No proxy installed, yet'");
```

```
class proxy extends MysqlndUhConnection {
 public function query($res, $query) {
   printf("%s(%s)\n", __METHOD__, var_export(func_get_args(), true));
   $ret = parent::query($res, $query);
   printf("%s returns %s\n", __METHOD__, var_export($ret, true));
   return $ret;
 }
}
mysqlnd_uh_set_connection_proxy(new proxy());

$mysqli->query("SELECT 'mysqlnd rocks!'");

$mysql = mysql_connect("localhost", "root", "", "test");
mysql_query("SELECT 'Ahoy Andrey!'", $mysql);

$pdo = new PDO("mysql:host=localhost;dbname=test", "root", "");
$pdo->query("SELECT 'Moin Johannes!'");
?>
```

The above example will output:

```


proxy::query(array (
  0 => NULL,
  1 => 'SELECT \'mysqlnd rocks!\'',
))
proxy::query returns true
proxy::query(array (
  0 => NULL,
  1 => 'SELECT \'Ahoy Andrey!\'',
))
proxy::query returns true
proxy::query(array (
  0 => NULL,
  1 => 'SELECT \'Moin Johannes!\'',
))
proxy::query returns true
```

See Also

mysqlnd_uh_set_statement_proxy
mysqlnd_uh.enable

9.9.3 mysqlnd_uh_set_statement_proxy

- mysqlnd_uh_set_statement_proxy

  Installs a proxy for mysqlnd statements

Description

```
  bool mysqlnd_uh_set_statement_proxy(
    MysqlndUhStatement statement_proxy);
```

Installs a proxy for mysqlnd statements. The proxy object will be used for all mysqlnd prepared statement objects, regardless which PHP MySQL extension (mysqli, mysql, PDO_MYSQL) has created them as long as the extension is compiled to use the mysqlnd library.

The function can be disabled with `mysqlnd_uh.enable`. If `mysqlnd_uh.enable` is set to `FALSE` the function will not install the proxy and always return `TRUE`. Additionally, an error of the type `E_WARNING` may be emitted. The error message may read like `PHP Warning: mysqlnd_uh_set_statement_proxy(): (Mysqlnd User Handler) The plugin has been disabled by setting the configuration parameter mysqlnd_uh.enable = false. The proxy has not been installed [...]`.

Parameters

`statement_proxy`	The mysqlnd statement proxy object of type `MysqlndUhStatement`

Return Values

Returns `TRUE` on success. Otherwise, returns `FALSE`

See Also

`mysqlnd_uh_set_connection_proxy`
`mysqlnd_uh.enable`

9.10 Change History

The Change History lists major changes users need to be aware if upgrading from one version to another. It is a high level summary of selected changes that may impact applications or might even break backwards compatibility. See also the `CHANGES` file contained in the source for additional changelog information. The commit history is also available.

9.10.1 PECL/mysqlnd_uh 1.0 series

1.0.1-alpha

- Release date: TBD
- Motto/theme: bug fix release

Feature changes

- Support of PHP 5.4.0 or later.
- BC break: `MysqlndUhConnection::changeUser` requires additional `passwd_len` parameter.
- BC break: `MYSQLND_UH_VERSION_STR` renamed to `MYSQLND_UH_VERSION`. `MYSQLND_UH_VERSION` renamed to `MYSQLND_UH_VERSION_ID`.
- BC break: `mysqlnd_uh.enabled` configuration setting renamed to `mysqlnd_uh.enable`.

1.0.0-alpha

- Release date: 08/2010
- Motto/theme: Initial release

Chapter 10 Mysqlnd connection multiplexing plugin

Table of Contents

The mysqlnd multiplexing plugin (`mysqlnd_mux`) multiplexes MySQL connections established by all PHP MySQL extensions that use the MySQL native driver (mysqlnd) for PHP.

The MySQL native driver for PHP features an internal C API for plugins, such as the connection multiplexing plugin, which can extend the functionality of mysqlnd. See the mysqlnd for additional details about its benefits over the MySQL Client Library libmysqlclient.

Mysqlnd plugins like `mysqlnd_mux` operate, for the most part, transparently from a user perspective. The connection multiplexing plugin supports all PHP applications, and all MySQL PHP extensions. It does not change existing APIs. Therefore, it can easily be used with existing PHP applications.

> **Note**
>
> This is a proof-of-concept. All features are at an early stage. Not all kinds of queries are handled by the plugin yet. Thus, it cannot be used in a drop-in fashion at the moment.
>
> Please, do not use this version in production environments.

10.1 Key Features

The key features of mysqlnd_mux are as follows:

- Transparent and therefore easy to use:
  - Supports all of the PHP MySQL extensions.
  - Little to no application changes are required, dependent on the required usage scenario.
- Reduces server load and connection establishment latency:
  - Opens less connections to the MySQL server.

- Less connections to MySQL mean less work for the MySQL server. In a client-server environment scaling the server is often more difficult than scaling the client. Multiplexing helps with horizontal scale-out (scale-by-client).
- Pooling saves connection time.
- Multiplexed connection: multiple user handles share the same network connection. Once opened, a network connection is cached and shared among multiple user handles. There is a 1:n relationship between internal network connection and user connection handles.
- Persistent connection: a network connection is kept open at the end of the web request, if the PHP deployment model allows. Thus, subsequently web requests can reuse a previously opened connection. Like other resources, network connections are bound to the scope of a process. Thus, they can be reused for all web requests served by a process.

10.2 Limitations

The proof-of-concept does not support unbuffered queries, prepared statements, and asynchronous queries.

The connection pool is using a combination of the transport method and hostname as keys. As a consequence, two connections to the same host using the same transport method (TCP/IP, Unix socket, Windows named pipe) will be linked to the same pooled connection even if username and password differ. Be aware of the possible security implications.

The proof-of-concept is transaction agnostic. It does not know about SQL transactions.

> **Note**
>
> Applications must be aware of the consequences of connection sharing connections.

10.3 About the name mysqlnd_mux

The shortcut `mysqlnd_mux` stands for `mysqlnd connection multiplexing plugin`.

10.4 Concepts

This explains the architecture and related concepts for this plugin. Reading and understanding these concepts is required to successfully use this plugin.

10.4.1 Architecture

The mysqlnd connection multiplexing plugin is implemented as a PHP extension. It is written in C and operates under the hood of PHP. During the startup of the PHP interpreter, in the module initialization phase of the PHP engine, it gets registered as a mysqlnd plugin to replace specific mysqlnd C methods.

The mysqlnd library uses PHP streams to communicate with the MySQL server. PHP streams are accessed by the mysqlnd library through its net module. The mysqlnd connection multiplexing plugin proxies methods of the mysqlnd library net module to control opening and closing of network streams.

Upon opening a user connection to MySQL using the appropriate connection functions of either mysqli, PDO_MYSQL or ext/mysql, the plugin will search its connection pool for an open network connection. If the pool contains a network connection to the host specified by the connect function using the transport method requested (TCP/IP, Unix domain socket, Windows named pipe), the pooled connection is linked to the user handle. Otherwise, a new network connection is opened, put into the poolm and associated with the user connection handle. This way, multiple user handles can be linked to the same network connection.

10.4.2 Connection pool

The plugins connection pool is created when PHP initializes its modules (`MINIT`) and free'd when PHP shuts down the modules (`MSHUTDOWN`). This is the same as for persistent MySQL connections.

Depending on the deployment model, the pool is used for the duration of one or multiple web requests. Network connections are bound to the lifespan of an operating system level process. If the PHP process serves multiple web requests as it is the case for Fast-CGI or threaded web server deployments, then the pooled connections can be reused over multiple connections. Because multiplexing means sharing connections, it can even happen with a threaded deployment that two threads or two distinct web requests are linked to one pooled network connections.

A pooled connection is explicitly closed once the last reference to it is released. An implicit close happens when PHP shuts down its modules.

10.4.3 Sharing connections

The PHP mysqlnd connection multiplexing plugin changes the relationship between a users connection handle and the underlying MySQL connection. Without the plugin, every MySQL connection belongs to exactly one user connection at a time. The multiplexing plugin changes. A MySQL connection is shared among multiple user handles. There no one-to-one relation if using the plugin.

Sharing pooled connections has an impact on the connection state. State changing operations from multiple user handles pointing to one MySQL connection are not isolated from each other. If, for example, a session variable is set through one user connection handle, the session variable becomes visible to all other user handles that reference the same underlying MySQL connection.

This is similar in concept to connection state related phenomens described for the PHP mysqlnd replication and load balancing plugin. Please, check the PECL/mysqlnd_ms documentation for more details on the state of a connection.

The proof-of-concept takes no measures to isolate multiplexed connections from each other.

10.5 Installing/Configuring

10.5.1 Requirements

PHP 5.5.0 or newer. Some advanced functionality requires PHP 5.5.0 or newer.

The mysqlnd_mux replication and load balancing plugin supports all PHP applications and all available PHP MySQL extensions (mysqli, mysql, PDO_MYSQL). The PHP MySQL extension must be configured to use mysqlnd in order to be able to use the mysqlnd_mux plugin for mysqlnd.

10.5.2 Installation

Information for installing this PECL extension may be found in the manual chapter titled Installation of PECL extensions. Additional information such as new releases, downloads, source files, maintainer information, and a CHANGELOG, can be located here: http://pecl.php.net/package/mysqlnd_mux

10.5.3 Runtime Configuration

The behaviour of these functions is affected by settings in php.ini.

Table 10.1 Mysqlnd_mux Configure Options

Name	Default	Changeable	Changelog
mysqlnd_mux.enable	0	PHP_INI_SYSTEM	

Here's a short explanation of the configuration directives.

mysqlnd_mux.enable integer — Enables or disables the plugin. If disabled, the extension will not plug into mysqlnd to proxy internal mysqlnd C API calls.

10.6 Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

Other

The plugins version number can be obtained using MYSQLND_MUX_VERSION or MYSQLND_MUX_VERSION_ID. MYSQLND_MUX_VERSION is the string representation of the numerical version number MYSQLND_MUX_VERSION_ID, which is an integer such as 10000. Developers can calculate the version number as follows.

Version (part)	Example
Major*10000	1*10000 = 10000
Minor*100	0*100 = 0
Patch	0 = 0
MYSQLND_MUX_VERSION_ID	10000

MYSQLND_MUX_VERSION (string) — Plugin version string, for example, “1.0.0-prototype”.

MYSQLND_MUX_VERSION_ID (integer) — Plugin version number, for example, 10000.

10.7 Change History

This change history is a high level summary of selected changes that may impact applications and/or break backwards compatibility.

See also the `CHANGES` file in the source distribution for a complete list of changes.

10.7.1 PECL/mysqlnd_mux 1.0 series

1.0.0-pre-alpha

- Release date: no package released, initial check-in 09/2012
- Motto/theme: Proof of concept

Initial check-in. Essentially a demo of the mysqlnd plugin API.

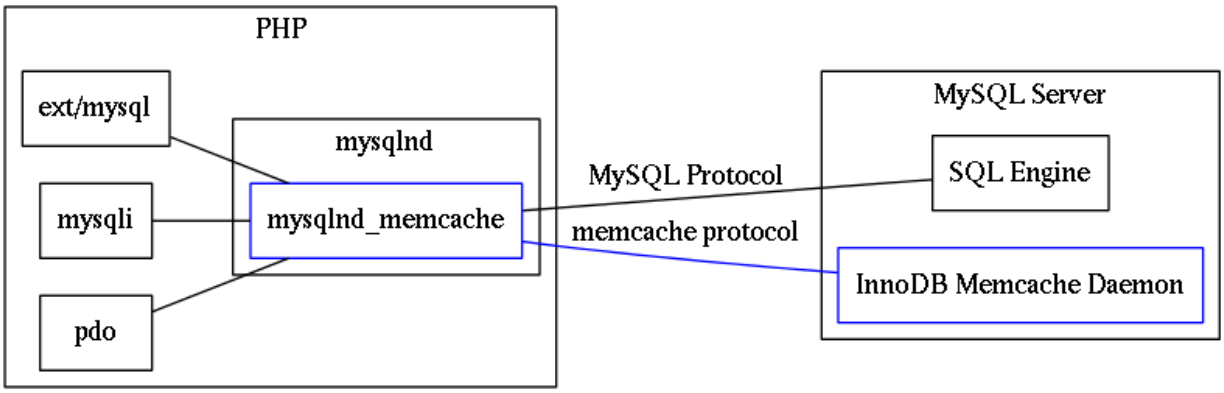
> **Note**
>
> This is the current development series. All features are at an early stage. Changes may happen at any time without prior notice. Please, do not use this version in production environments.
>
> The documentation may not reflect all changes yet.

Chapter 11 Mysqlnd Memcache plugin

Table of Contents

The mysqlnd memcache plugin (`mysqlnd_memcache`) is an PHP extension for transparently translating SQL into requests for the MySQL InnoDB Memcached Daemon Plugin (server plugin). It includes experimental support for the MySQL Cluster Memcached Daemon. The server plugin provides access to data stored inside MySQL InnoDB (respectively MySQL Cluster NDB) tables using the Memcache protocol. This PHP extension, which supports all PHP MySQL extensions that use mysqlnd, will identify tables exported in this way and will translate specific SELECT queries into Memcache requests.

Figure 11.1 mysqlnd_memcache data flow



Note

This plugin depends on the MySQL InnoDB Memcached Daemon Plugin. It is not provided to be used with a stand-alone Memcached. For a generic query cache using Memcached look at the mysqlnd query cache plugin. For direct Memcache access look at the memcache and memcached extensions.

The MySQL native driver for PHP is a C library that ships together with PHP as of PHP 5.3.0. It serves as a drop-in replacement for the MySQL Client Library (libmysqlclient). Using mysqlnd has several advantages: no extra downloads are required because it's bundled with PHP, it's under the PHP license, there is lower memory consumption in certain cases, and it contains new functionality such as asynchronous queries.

The `mysqlnd_mmemcache` operates, for the most part, transparently from a user perspective. The mysqlnd memcache plugin supports all PHP applications, and all MySQL PHP extensions. It does not change existing APIs. Therefore, it can easily be used with existing PHP applications.

The MySQL Memcache plugins add key-value style access method for data stored in InnoDB resp. NDB (MySQL Cluster) SQL tables through the Memcache protocol. This type of key-value access if often faster than using SQL.

11.1 Key Features

The key features of PECL/mysqlnd_memcache are as follows.

- Possible performance benefits
  - Client-side: light-weight protocol.
  - Server-side: no SQL parsing, direct access to the storage.
  - Please, run your own benchmarks! Actual performance results are highly dependent on setup and hardware used.

11.2 Limitations

The initial version is not binary safe. Due to the way the MySQL Memcache plugins works there are restrictions related to separators.

Prepared statements and asynchronous queries are not supported. Result set meta data support is limited.

The mapping information for tables accessible via Memcache is not cached in the plugin between requests but fetched from the MySQL server each time a MySQL connection is associated with a Memcache connection. See `mysqlnd_memcache_set` for details.

11.3 On the name

The shortcut `mysqlnd_memcache` stands for `mysqlnd memcache plugin`. Memcache refers to support of the MySQL Memcache plugins for InnoDB and NDB (MySQL Cluster). The plugin is not related to the Memcached cache server.

11.4 Quickstart and Examples

The mysqlnd memcache plugin is easy to use. This quickstart will demo typical use-cases, and provide practical advice on getting started.

It is strongly recommended to read the reference sections in addition to the quickstart. The quickstart tries to avoid discussing theoretical concepts and limitations. Instead, it will link to the reference sections. It is safe to begin with the quickstart. However, before using the plugin in mission critical environments we urge you to read additionally the background information from the reference sections.

11.4.1 Setup

The plugin is implemented as a PHP extension. See also the installation instructions to install this extension.

Compile or configure the PHP MySQL extension (API) (mysqli, PDO_MYSQL, mysql). That extension must use the mysqlnd library as because mysqlnd_memcache is a plugin for the mysqlnd library. For additional information, refer to the mysqlnd_memcache installation instructions.

Then, load this extension into PHP and activate the plugin in the PHP configuration file using the PHP configuration directive named mysqlnd_memcache.enable.

Example 11.1 Enabling the plugin (php.ini)

```
; On Windows the filename is php_mysqnd_memcache.dll
; Load the extension
extension=mysqlnd_memcache.so
; Enable it
mysqlnd_memcache.enable=1
```

Follow the instructions given in the MySQL Reference Manual on installing the Memcache plugins for the MySQL server. Activate the plugins and configure Memcache access for SQL tables.

The examples in this quickguide assume that the following table exists, and that Memcache is configured with access to it.

Example 11.2 SQL table used for the Quickstart

```
CREATE TABLE test(
  id CHAR(16),
  f1 VARCHAR(255),
  f2 VARCHAR(255),
  f3 VARCHAR(255),
  flags INT NOT NULL,
  cas_column INT,
  expire_time_column INT,
  PRIMARY KEY(id)
  ) ENGINE=InnoDB;

INSERT INTO test (id, f1, f2, f3) VALUES (1, 'Hello', 'World', '!');
INSERT INTO test (id, f1, f2, f3) VALUES (2, 'Lady', 'and', 'the tramp');

INSERT INTO innodb_memcache.containers(
  name, db_schema, db_table, key_columns, value_columns,
  flags, cas_column, expire_time_column, unique_idx_name_on_key)
VALUES (
  'plugin_test', 'test', 'test', 'id', 'f1,f2,f3',
  'flags', 'cas_column', 'expire_time_column', 'PRIMARY KEY');
```

11.4.2 Usage

After associating a MySQL connection with a Memcache connection using `mysqnd_memcache_set` the plugin attempts to transparently replace SQL `SELECT` statements by a memcache access. For that purpose the plugin monitors all SQL statements executed and tries to match the statement string against `MYSQLND_MEMCACHE_DEFAULT_REGEXP`. In case of a match, the mysqlnd memcache plugin checks whether the `SELECT` is accessing only columns of a mapped table and the `WHERE` clause is limited to a single key lookup.

In case of the example SQL table, the plugin will use the Memcache interface of the MySQL server to fetch results for a SQL query like `SELECT f1, f2, f3 WHERE id = n`.

Example 11.3 Basic example.

```
<?php
$mysqli = new mysqli("host", "user", "passwd", "database");
$memc = new Memcached();
$memc->addServer("host", 11211);
mysqlnd_memcache_set($mysqli, $memc);

/*
   This is a query which queries table test using id as key in the WHERE part
   and is accessing fields f1, f2 and f3. Therefore, mysqlnd_memcache
   will intercept it and route it via memcache.
*/
$result = $mysqli->query("SELECT f1, f2, f3 FROM test WHERE id = 1");
while ($row = $result->fetch_row()) {
    print_r($row);
}

/*
   This is a query which queries table test but using f1 in the WHERE clause.
   Therefore, mysqlnd_memcache can't intercept it. This will be executed
   using the MySQL protocol
*/
$mysqli->query("SELECT id FROM test WHERE f1 = 'Lady'");
while ($row = $result->fetch_row()) {
    print_r($row);
}
?>
```

The above example will output:

```
array(
    [f1] => Hello
    [f2] => World
    [f3] => !
)
array(
    [id] => 2
)
```

11.5 Installing/Configuring

11.5.1 Requirements

PHP: this extension requires PHP 5.4+, version PHP 5.4.4 or never. The required PHP extensions are PCRE (enabled by default), and the memcached extension version 2.0.x.

The `mysqlnd_memcache` Memcache plugin supports all PHP applications and all available PHP MySQL extensions (mysqli, mysql, PDO_MYSQL). The PHP MySQL extension must be configured with mysqlnd support.

For accessing `InnoDB` tables, this PHP extension requires `MySQL Server 5.6.6` or newer with the InnoDB Memcache Daemon Plugin enabled.

For accessing `MySQL Cluster NDB` tables, this PHP extension requires `MySQL Cluster 7.2` or newer with the NDB Memcache API nodes enabled.

11.5.2 Installation

This PECL extension is not bundled with PHP.

Information for installing this PECL extension may be found in the manual chapter titled Installation of PECL extensions. Additional information such as new releases, downloads, source files, maintainer information, and a CHANGELOG, can be located here: http://pecl.php.net/package/mysqlnd_memcache

A DLL for this PECL extension is currently unavailable. See also the building on Windows section.

11.5.3 Runtime Configuration

The behaviour of these functions is affected by settings in `php.ini`.

Table 11.1 Mysqlnd_memcache Configure Options

Name	Default	Changeable	Changelog
mysqlnd_memcache.enable	1	`PHP_INI_SYSTEM`	Available since 1.0.0

Here's a short explanation of the configuration directives.

`mysqlnd_memcache.enable`
integer

Enables or disables the plugin. If disabled, the extension will not plug into mysqlnd to proxy internal mysqlnd C API calls.

> **Note**
>
> This option is mainly used by developers to build this extension statically into PHP. General users are encouraged to build this extension as a

shared object, and to unload it completely when it is not needed.

11.6 Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

MySQL Memcache Plugin related

`MYSQLND_MEMCACHE_DEFAULT_REGEXP` (string) — Default regular expression (PCRE style) used for matching `SELECT` statements that will be mapped into a MySQL Memcache Plugin access point, if possible.

It is also possible to use `mysqlnd_memcache_set`, but the default approach is using this regular expression for pattern matching.

Assorted

The version number of this plugin can be obtained by using `MYSQLND_MEMCACHE_VERSION` or `MYSQLND_MEMCACHE_VERSION_ID`. `MYSQLND_MEMCACHE_VERSION` is the string representation of the numerical version number `MYSQLND_MEMCACHE_VERSION_ID`, which is an integer such as 10000. Developers can calculate the version number as follows.

Version (part)	Example
Major*10000	1*10000 = 10000
Minor*100	0*100 = 0
Patch	0 = 0
MYSQLND_MEMCACHE_VERSION_ID	10000

`MYSQLND_MEMCACHE_VERSION` (string) — Plugin version string, for example, “1.0.0-alpha”.

`MYSQLND_MEMCACHE_VERSION_ID` (integer) — Plugin version number, for example, 10000.

11.7 Mysqlnd_memcache Functions

11.7.1 `mysqlnd_memcache_get_config`

- `mysqlnd_memcache_get_config`

  Returns information about the plugin configuration

Description

```
array mysqlnd_memcache_get_config(
  mixed connection);
```

This function returns an array of all mysqlnd_memcache related configuration information that is attached to the MySQL connection. This includes MySQL, the Memcache object provided via `mysqlnd_memcache_set`, and the table mapping configuration that was automatically collected from the MySQL Server.

Parameters

`connection`	A handle to a MySQL Server using one of the MySQL API extensions for PHP, which are PDO_MYSQL, mysqli or ext/mysql.

Return Values

An array of mysqlnd_memcache configuration information on success, otherwise `FALSE`.

The returned array has these elements:

Table 11.2 `mysqlnd_memcache_get_config` array structure

Array Key	Description
memcached	Instance of Memcached associated to this MySQL connection by `mysqlnd_memcache_set`. You can use this to change settings of the memcache connection, or directly by querying the server on this connection.
pattern	The PCRE regular expression used to match the SQL query sent to the server. Queries matching this pattern will be further analyzed to decide whether the query can be intercepted and sent via the memcache interface or whether the query is sent using the general MySQL protocol to the server. The pattern is either the default pattern (`MYSQLND_MEMCACHE_DEFAULT_REGEXP`) or it is set via `mysqlnd_memcache_set`.
mappings	An associative array with a list of all configured containers as they were discovered by this plugin. The key for these elements is the name of the container in the MySQL configuration. The value is described below. The contents of this field is created by querying the MySQL Server during association to MySQL and a memcache connection using `mysqlnd_memcache_set`.
mapping_query	An SQL query used during `mysqlnd_memcache_set` to identify the available containers and mappings. The result of that query is provided in the mappings element.

Table 11.3 Mapping entry structure

Array Key	Description
prefix	A prefix used while accessing data via memcache. With the MySQL InnoDB Memcache Deamon plugin, this usually begins with @@ and ends with a configurable separator. This prefix is placed in front of the key value while using the memcache protocol.

Array Key	Description
schema_name	Name of the schema (database) which contains the table being accessed.
table_name	Name of the table which contains the data accessible via memcache protocol.
id_field_name	Name of the database field (column) with the id used as key when accessing the table via memcache. Often this is the database field having a primary key.
separator	The separator used to split the different field values. This is needed as memcache only provides access to a single value while MySQL can map multiple columns to this value. **Note** The separator, which can be set in the MySQL Server configuration, should not be part of any value retrieved via memcache because proper mapping can't be guaranteed.
fields	An array with the name of all fields available for this mapping.

Examples

Example 11.4 `mysqlnd_memcache_get_config` example

```
<?php
$mysqli = new mysqli("host", "user", "passwd", "database");
$memc = new Memcached();
$memc->addServer("host", 11211);
mysqlnd_memcache_set($mysqli, $memc);

var_dump(mysqlnd_memcache_get_config($mysqli));
?>
```

The above example will output:

```
array(4) {
  ["memcached"]=>
  object(Memcached)#2 (0) {
  }
  ["pattern"]=>
  string(125) "/^\s*SELECT\s*(.+?)\s*FROM\s*`?([a-z0-9_]+)`?\s*WHERE\s*`?([a-z0-9_]+)`?\s*=\s*(?(?=["'])["'](
  ["mappings"]=>
  array(1) {
    ["mymem_test"]=>
    array(6) {
      ["prefix"]=>
```

```
      string(13) "@@mymem_test."
      ["schema_name"]=>
      string(4) "test"
      ["table_name"]=>
      string(10) "mymem_test"
      ["id_field_name"]=>
      string(2) "id"
      ["separator"]=>
      string(1) "|"
      ["fields"]=>
      array(3) {
        [0]=>
        string(2) "f1"
        [1]=>
        string(2) "f2"
        [2]=>
        string(2) "f3"
      }
    }
  }
  ["mapping_query"]=>
  string(209) "    SELECT c.name,
                          CONCAT('@@', c.name, (SELECT value FROM innodb_memcache.config_options WHERE nam
                          c.db_schema,
                          c.db_table,
                          c.key_columns,
                          c.value_columns,
                          (SELECT value FROM innodb_memcache.config_options WHERE name = 'separator') AS s
                     FROM innodb_memcache.containers c"
}
```

See Also

mysqlnd_memcache_set

11.7.2 mysqlnd_memcache_set

- mysqlnd_memcache_set

  Associate a MySQL connection with a Memcache connection

Description

```
bool mysqlnd_memcache_set(
  mixed mysql_connection,
  Memcached memcache_connection,
  string pattern,
  callback callback);
```

Associate *mysql_connection* with *memcache_connection* using *pattern* as a PCRE regular expression, and *callback* as a notification callback or to unset the association of *mysql_connection*.

While associating a MySQL connection with a Memcache connection, this function will query the MySQL Server for its configuration. It will automatically detect whether the server is configured to use the InnoDB Memcache Daemon Plugin or MySQL Cluster NDB Memcache support. It will also query the server to automatically identify exported tables and other configuration options. The results of this automatic configuration can be retrieved using mysqlnd_memcache_get_config.

Parameters

mysql_connection	A handle to a MySQL Server using one of the MySQL API extensions for PHP, which are PDO_MYSQL, mysqli or ext/mysql.
memcache_connection	A Memcached instance with a connection to the MySQL Memcache Daemon plugin. If this parameter is omitted, then *mysql_connection* will be unassociated from any memcache connection. And if a previous association exists, then it will be replaced.
pattern	A regular expression in Perl Compatible Regular Expression syntax used to identify potential Memcache-queries. The query should have three sub patterns. The first subpattern contains the requested field list, the second the name of the ID column from the query and the third the requested value. If this parameter is omitted or os set to NULL, then a default pattern will be used.
callback	A callback which will be used whenever a query is being sent to MySQL. The callback will receive a single boolean parameter telling if a query was sent via Memcache.

Return Values

TRUE if the association or disassociation is successful, otherwise FALSE if there is an error.

Examples

Example 11.5 mysqlnd_memcache_set example with var_dump as a simple debugging callback.

```
<?php
$mysqli = new mysqli("host", "user", "passwd", "database");
$memc = new Memcached();
$memc->addServer("host", 11211);
mysqlnd_memcache_set($mysqli, $memc, NULL, 'var_dump');

/* This query will be intercepted and executed via Memcache protocol */
echo "Sending query for id via Memcache: ";
$mysqli->query("SELECT f1, f2, f3 FROM test WHERE id = 1");

/* f1 is not configured as valid key field, this won't be sent via Memcache */
echo "Sending query for f1 via Memcache: ";
$mysqli->query("SELECT id FROM test WHERE f1 = 1");

mysqlnd_memcache_set($mysqli);

/* Now the regular MySQL protocol will be used */
echo "var_dump won't be invoked: ";
$mysqli->query("SELECT f1, f2, f3 WHERE id = 1");

?>
```

The above example will output:

```
Sending query for id via Memcache: bool(true)
Sending query for f1 via Memcache: bool(false)
var_dump won't be invoked:
```

See Also

`mysqlnd_memcache_get_config`

11.8 Change History

This change history is a high level summary of selected changes that may impact applications and/or break backwards compatibility.

See also the `CHANGES` file in the source distribution for a complete list of changes.

11.8.1 PECL/mysqlnd_memcache 1.0 series

1.0.0-alpha

- Release date: TBD
- Motto/theme: Basic mapping of SQL SELECT to a MySQL Memcache plugin access.

The initial release does map basic SQL SELECT statements to a MySQL Memcache plugin access. This bares potential performance benefits as the direct key-value access to MySQL storage using the Memcache protocol is usually faster than using SQL access.

Chapter 12 Common Problems with MySQL and PHP

- `Error: Maximum Execution Time Exceeded`: This is a PHP limit; go into the `php.ini` file and set the maximum execution time up from 30 seconds to something higher, as needed. It is also not a bad idea to double the RAM allowed per script to 16MB instead of 8MB.
- `Fatal error: Call to unsupported or undefined function mysql_connect() in ...`: This means that your PHP version isn't compiled with MySQL support. You can either compile a dynamic MySQL module and load it into PHP or recompile PHP with built-in MySQL support. This process is described in detail in the PHP manual.
- `Error: Undefined reference to 'uncompress'`: This means that the client library is compiled with support for a compressed client/server protocol. The fix is to add `-lz` last when linking with `-lmysqlclient`.
- `Error: Client does not support authentication protocol`: This is most often encountered when trying to use the older `mysql` extension with MySQL 4.1.1 and later. Possible solutions are: downgrade to MySQL 4.0; switch to PHP 5 and the newer `mysqli` extension; or configure the MySQL server with the `old_passwords` system variable set to 1. (See Client does not support authentication protocol, for more information.)