PDO

There are many tutorials on PDO already, but unfortunately, most of them fail to explain the real benefits of PDO, or even promote rather bad practices. The only two exceptions are phptherightway.com and hashphp.org, but they miss a lot of important information. As a result, half of PDO's features remain in obscurity and are almost never used by PHP developers, who, as a result, are constantly trying to reinvent the wheel which *already exists in PDO*.

Unlike those, this tutorial is written by someone who has used PDO for many years, dug through it, and answered thousands questions on Stack Overflow (the sole gold PDO badge bearer). Following the mission of this site, this article will disprove various delusions and bad practices, while showing the right way instead.

Although this tutorial is based on **mysql** driver, the information, in general, is applicable for any driver supported.

Why PDO?

First things first. Why PDO at all?

PDO is a Database Access Abstraction Layer. The abstraction, however, is two-fold: one is widely known but less significant, while another is obscure but of most importance.

Everyone knows that PDO offers unified interface to access many different databases. Although this feature is magnificent by itself, it doesn't make a big deal for the particular application, where only one database backend is used anyway. And, despite some rumors, it is impossible to switch database backends by changing a single line in PDO config - due to different SQL flavors (to do so, one needs to use an averaged query language like DQL). Thus, for the average LAMP developer, this point is rather insignificant, and to him, PDO is just a more complicated version of familiar `mysql(i)_query()` function. However, it is not; it is much, much more.

PDO abstracts not only a database API, but also basic operations that otherwise have to be repeated hundreds of times in every application, making your code extremely WET. Unlike *mysql* and *mysqli*, both of which are low level bare APIs not intended to be used directly (but only as a building material for some higher level abstraction layer), `PDO` *is* such an abstraction already. Still incomplete though, but at least usable.

The real PDO benefits are:

- **security** (*usable* prepared statements)
- **usability** (many helper functions to automate routine operations)
- **reusability** (unified API to access multitude of databases, from SQLite to Oracle)

Note that although PDO is the best out of native db drivers, for a modern web-application consider to use an ORM with a Query Builder, or any other higher level abstraction library, with only occasional fallback to vanilla PDO. Good ORMs are Doctrine, Eloquent, RedBean, and Yii::AR. Aura.SQL is a good example of a PDO wrapper with many additional features.

Either way, it's good to know the basic tools first. So, let's begin:

Connecting. DSN

PDO has a fancy connection method called DSN. It's nothing complicated though - instead of one plain and simple list of options, PDO asks you to input different configuration directives in three different places:

- `database driver`, `host`, `db (schema) name` and `charset`, as well as less frequently used `port` and `unix_socket` go into DSN;
- `username` and `password` go to constructor;
- all other options go into options array.

where DSN is a semicolon-delimited string, consists of `param=value` pairs, that begins from the driver name and a colon:

```
      mysql:host=localhost;dbname=test;port=3306;charset=utf8mb4
driver^    ^ colon          ^param=value pair    ^semicolon
```

Note that it's important to follow the proper format - **no spaces or quotes or other decorations have to be used in DSN,** but only parameters, values and delimiters, as shown in the manual.

Here goes an example for mysql:

```
$host = '127.0.0.1';
$db   = 'test';
$user = 'root';
$pass = '';
$charset = 'utf8mb4';

$dsn = "mysql:host=$host;dbname=$db;charset=$charset";
$options = [
    PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    PDO::ATTR_EMULATE_PREPARES   => false,
];
try {
     $pdo = new PDO($dsn, $user, $pass, $options);
} catch (\PDOException $e) {
     throw new \PDOException($e->getMessage(), (int)$e->getCode());
}
```

With all aforementioned variables properly set, we will have proper PDO instance in `$pdo` variable.

Important notes for the late mysql extension users:

1. Unlike old `mysql_*` functions, which can be used anywhere in the code, `PDO` instance is stored in a regular variable, which means it can be inaccessible inside functions - so, one has to make it accessible, by means of passing it via function parameters or using more advanced techniques, such as IoC container.

2. The connection has to be made only once! No connects in every function. No connects in every class constructor. Otherwise, multiple connections will be created, which will eventually kill your database server. Thus, a sole PDO instance has to be created and then used through whole script execution.
3. It is very important to **set charset through DSN** - that's the only proper way because it tells PDO which charset is going to be used. Therefore forget about running `SET NAMES` query manually, either via `query()` or `PDO::MYSQL_ATTR_INIT_COMMAND`. Only if your PHP version is unacceptably outdated (namely below 5.3.6), you have to use `SET NAMES` query and always turn emulation mode off.

More details regarding Mysql can be found in the corresponding chapter, Connecting to MySQL

Running queries. PDO::query()

There are two ways to run a query in PDO. If no variables are going to be used in the query, you can use the PDO::query() method. It will run your query and return special object of PDOStatement class which can be roughly compared to a resource, returned by `mysql_query()`, especially in the way you can get actual rows out of it:

```
$stmt = $pdo->query('SELECT name FROM users');
while ($row = $stmt->fetch())
{
    echo $row['name'] . "\n";
}
```

Also, the `query()` method allows us to use a neat method chaining for SELECT queries, which will be shown below.

Prepared statements. Protection from SQL injections

This is the main and the only important reason why you were deprived from your beloved `mysql_query()` function and thrown into the harsh world of Data Objects: PDO has prepared statements support out of the box. Prepared statement is **the only proper way to run a query**, if any variable is going to be used in it. The reason why it is so important is explained in detail in The Hitchhiker's Guide to SQL Injection prevention.

So, for every query you run, if at least one variable is going to be used, you have to substitute it with a **placeholder**, then prepare your query, and then execute it, passing variables separately.

Long story short, it is not as hard as it seems. In most cases, you need only two functions - prepare() and execute().

First of all, you have to alter your query, adding placeholders in place of variables. Say, a code like this

```
$sql = "SELECT * FROM users WHERE email = '$email' AND status='$status'";
```

will become

```
$sql = 'SELECT * FROM users WHERE email = ? AND status=?';
```

or

```
$sql = 'SELECT * FROM users WHERE email = :email AND status=:status';
```

Note that PDO supports positional (`?`) and named (`:email`) placeholders, the latter always begins from a colon and can be written using letters, digits and underscores only. Also note that **no quotes** have to be ever used around placeholders.

Having a query with placeholders, you have to prepare it, using the `PDO::prepare()` method. This function will return the same `PDOStatement` object we were talking about above, but *without any data attached to it*.

Finally, to get the query executed, you must run `execute()` method of this object, passing variables in it, in the form of array. And after that, you will be able to get the resulting data out of statement (if applicable):

```
$stmt = $pdo->prepare('SELECT * FROM users WHERE email = ? AND status=?');
$stmt->execute([$email, $status]);
$user = $stmt->fetch();
// or
$stmt = $pdo-
>prepare('SELECT * FROM users WHERE email = :email AND status=:status');
$stmt->execute(['email' => $email, 'status' => $status]);
$user = $stmt->fetch();
```

As you can see, for the positional placeholders, you have to supply a regular array with values, while for the named placeholders, it has to be an associative array, where keys have to match the placeholder names in the query. You cannot mix positional and named placeholders in the same query.

Please note that positional placeholders let you write shorter code, but are sensitive to the order of arguments (which have to be exactly the same as the order of the corresponding placeholders in the query). While named placeholders make your code more verbose, they allow random binding order.

Also note that despite a widespread delusion, no ":" in the keys is required.

After the execution you may start getting your data, using all supported methods, as described down in this article.

More examples can be found in the respective article.

Binding methods

Passing data into `execute()` (like shown above) should be considered default and most convenient method. When this method is used, all values will be bound as **strings** (save for `NULL` values, that will be sent to the query as is, i.e. as SQL `NULL`), but most of time it's all right and won't cause any problem.

However, sometimes it's better to set the data type explicitly. Possible cases are:

- LIMIT clause (or any other SQL clause that just cannot accept a string operand) if emulation mode is turned ON.
- complex queries with non-trivial query plan that can be affected by a wrong operand type
- peculiar column types, like `BIGINT` or `BOOLEAN` that require an operand of exact type to be bound (note that in order to bind a BIGINT value with PDO::PARAM_INT you need a mysqlnd-based installation).

In such a case explicit binding have to be used, for which you have a choice of two functions, bindValue() and bindParam(). The former one has to be preferred, because, unlike `bindParam()` it has no side effects to deal with.

Query parts you can bind

It is very important to understand which query parts you can bind using prepared statements and which you cannot. In fact, the list is overwhelmingly short: only string and numeric literals can be bound. So you can tell that as long as your data can be represented in the query as a numeric or a quoted string literal - it can be bound. For all other cases you cannot use PDO prepared statements at all: neither an identifier, or a comma-separated list, or a part of a quoted string literal or whatever else arbitrary query part cannot be bound using a prepared statement.

Workarounds for the most frequent use cases can be found in the corresponding part of the article

Prepared statements. Multiple execution

Sometimes you can use prepared statements for the multiple execution of a prepared query. It is slightly faster than performing the same query again and again, as it does query parsing only once. This feature would have been more useful if it was possible to execute a statement prepared in another PHP instance. But alas - it is not. So, you are limited to repeating the same query only within the same instance, which is seldom needed in regular PHP scripts and which is limiting the use of this feature to repeated inserts or updates:

```
$data = [
    1 => 1000,
    5 =>  300,
    9 =>  200,
];
$stmt = $pdo->prepare('UPDATE users SET bonus = bonus + ? WHERE id = ?');
foreach ($data as $id => $bonus)
{
    $stmt->execute([$bonus, $id]);
}
```

Note that this feature is a bit overrated. Not only it is needed too seldom to talk about, but the performance gain is not that big - query parsing is *real* fast these times.

Note that you can get this advantage only when emulation mode is turned **off**.

Running SELECT INSERT, UPDATE, or DELETE statements

Come on folks. There is absolutely nothing special in these queries. To PDO they all the same. It doesn't matter which query you are running.

Just like it was shown above, what you need is to prepare a query with placeholders, and then execute it, sending variables separately. Either for `DELETE` and `SELECT` query the process is essentially the same. The only difference is (as DML queries do not return any data), that you can use the method chaining and thus call `execute()` right along with `prepare()`:

```
$sql = "UPDATE users SET name = ? WHERE id = ?";
$pdo->prepare($sql)->execute([$name, $id]);
```

However, if you want to get the number of affected rows, the code will have to be the same boresome three lines:

```
$stmt = $pdo->prepare("DELETE FROM goods WHERE category = ?");
$stmt->execute([$cat]);
$deleted = $stmt->rowCount();
```

More examples can be found in the respective article.

Getting data out of statement. foreach()

The most basic and direct way to get multiple rows from a statement would be `foreach()` loop. Thanks to Traversable interface, `PDOStatement` can be iterated over by using `foreach()` operator:

```
$stmt = $pdo->query('SELECT name FROM users');
foreach ($stmt as $row)
{
    echo $row['name'] . "\n";
}
```

Note that this method is memory-friendly, as it doesn't load all the resulting rows in the memory but delivers them one by one (though keep in mind this issue).

Getting data out of statement. fetch()

We have seen this function already, but let's take a closer look. It fetches a single row from database, and moves the internal pointer in the result set, so consequent calls to this function will return all the resulting rows one by one. Which makes this method a rough analogue to `mysql_fetch_array()` but it works in a slightly different way: instead of many separate functions (`mysql_fetch_assoc()`, `mysql_fetch_row()`, etc), there is only one, but its behavior can be changed by a parameter. There are many fetch modes in PDO, and we will discuss them later, but here are few for starter:

- `PDO::FETCH_NUM` returns enumerated array
- `PDO::FETCH_ASSOC` returns associative array
- `PDO::FETCH_BOTH` - both of the above
- `PDO::FETCH_OBJ` returns object

- `PDO::FETCH_LAZY` allows all three (numeric associative and object) methods without memory overhead.

From the above you can tell that this function have to be used in two cases:

1. When only one row is expected - to get that only row. For example,

   ```
   $row = $stmt->fetch(PDO::FETCH_ASSOC);
   ```

   Will give you single row from the statement, in the form of associative array.

2. When we need to process the returned data somehow before use. In this case it have to be run through usual while loop, like one shown above.

Another useful mode is `PDO::FETCH_CLASS`, which can create an object of particular class

```
$news = $pdo->query('SELECT * FROM news')-
>fetchAll(PDO::FETCH_CLASS, 'News');
```

will produce an array filled with objects of News class, setting class properties from returned values. Note that in this mode

- properties are set *before* constructor call
- for all undefined properties `__set` magic method will be called
- if there is no `__set` method in the class, then new property will be created
- private properties will be filled as well, which is a bit unexpected but quite handy

Note that default mode is `PDO::FETCH_BOTH`, but you can change it using `PDO::ATTR_DEFAULT_FETCH_MODE` configuration option as shown in the connection example. Thus, once set, it can be omitted most of the time.

Return types.

Only when PDO is built upon mysqlnd *and* emulation mode is **off**, then PDO will return `int` and `float` values with respective types. Say, if we create a table

```
create table typetest (string varchar(255), `int` int, `float` float, `null
` int);
insert into typetest values('foo',1,1.1,NULL);
```

And then query it from mysqlnd-based PDO with emulation turned off, the output will be

```
array(4) {
  ["string"] => string(3) "foo"
  ["int"]    => int(1)
  ["float"]  => float(1.1)
  ["null"]   => NULL
}
```

Otherwise the familiar `mysql_fetch_array()` behavior will be followed - all values returned as strings with only `NULL` returned as `NULL`.

If for some reason you don't like this behavior and prefer the old style with strings and NULLs only, then you can use the following configuration option to override it:

```
$pdo->setAttribute(PDO::ATTR_STRINGIFY_FETCHES, true);
```

Note that for the `DECIMAL` type the string is always returned, due to nature of this type intended to retain the precise value, unlike deliberately non-precise FLOAT and DOUBLE types.

Getting data out of statement. fetchColumn()

A neat helper function that returns value of the singe field of returned row. Very handy when we are selecting only one field:

```
// Getting the name based on id
$stmt = $pdo->prepare("SELECT name FROM table WHERE id=?");
$stmt->execute([$id]);
$name = $stmt->fetchColumn();

// getting number of rows in the table utilizing method chaining
$count = $pdo->query("SELECT count(*) FROM table")->fetchColumn();
```

Getting data out of statement in dozens different formats. fetchAll()

That's most interesting function, with most astonishing features. Mostly thanks to its existence one can call PDO a wrapper, as this function can automate many operations otherwise performed manually.

`PDOStatement::fetchAll()` returns an array that consists of **all the rows** returned by the query. From this fact we can make two conclusions:

1. This function should not be used, if many rows has been selected. In such a case conventional while loop ave to be used, fetching rows one by one instead of getting them all into array at once. "Many" means more than it is suitable to be shown on the average web page.
2. This function is mostly useful in a modern web application that never outputs data right away during fetching, but rather passes it to template.

You'd be amazed, in how many different formats this function can return data in (and how little an average PHP user knows of them), all controlled by `PDO::FETCH_*` variables. Some of them are:

Getting a plain array.

By default, this function will return just simple enumerated array consists of all the returned rows. Row formatting constants, such as `PDO::FETCH_NUM`, `PDO::FETCH_ASSOC`, `PDO::FETCH_OBJ` etc can change the row format.

```
$data = $pdo->query('SELECT name FROM users')->fetchAll(PDO::FETCH_ASSOC);
var_export($data);
/*
array (
```

```
  0 => array('John'),
  1 => array('Mike'),
  2 => array('Mary'),
  3 => array('Kathy'),
)*/
```

Getting a column.

It is often very handy to get plain one-dimensional array right out of the query, if only one column out of many rows being fetched. Here you go:

```
$data = $pdo->query('SELECT name FROM users')->fetchAll(PDO::FETCH_COLUMN);
/* array (
  0 => 'John',
  1 => 'Mike',
  2 => 'Mary',
  3 => 'Kathy',
)*/
```

Getting key-value pairs.

Also extremely useful format, when we need to get the same column, but indexed not by numbers in order but by another field. Here goes `PDO::FETCH_KEY_PAIR` constant:

```
$data = $pdo->query('SELECT id, name FROM users')-
>fetchAll(PDO::FETCH_KEY_PAIR);
/* array (
  104 => 'John',
  110 => 'Mike',
  120 => 'Mary',
  121 => 'Kathy',
)*/
```

Note that you have to select only two columns for this mode, first of which have to be unique.

Getting rows indexed by unique field

Same as above, but getting not one column but full row, yet indexed by an unique field, thanks to `PDO::FETCH_UNIQUE` constant:

```
$data = $pdo->query('SELECT * FROM users')->fetchAll(PDO::FETCH_UNIQUE);
/* array (
  104 => array (
    'name' => 'John',
    'car' => 'Toyota',
  ),
  110 => array (
    'name' => 'Mike',
    'car' => 'Ford',
  ),
  120 => array (
    'name' => 'Mary',
    'car' => 'Mazda',
  ),
  121 => array (
    'name' => 'Kathy',
```

```
    'car' => 'Mazda',
  ),
)*/
```

Note that first column selected have to be unique (in this query it is assumed that first column is id, but to be sure better list it explicitly).

Getting rows grouped by some field

`PDO::FETCH_GROUP` will group rows into a nested array, where indexes will be unique values from the first columns, and values will be arrays similar to ones returned by regular `fetchAll()`. The following code, for example, will separate boys from girls and put them into different arrays:

```
$data = $pdo->query('SELECT sex, name, car FROM users')-
>fetchAll(PDO::FETCH_GROUP);
array (
  'male' => array (
    0 => array (
      'name' => 'John',
      'car' => 'Toyota',
    ),
    1 => array (
      'name' => 'Mike',
      'car' => 'Ford',
    ),
  ),
  'female' => array (
    0 => array (
      'name' => 'Mary',
      'car' => 'Mazda',
    ),
    1 => array (
      'name' => 'Kathy',
      'car' => 'Mazda',
    ),
  ),
)
```

So, this is the ideal solution for such a popular demand like "group events by date" or "group goods by category". Some real life use cases:

- How to multiple query results in order to reduce the query number?
- List records grouped by category name

Other modes

Of course, there is a `PDO::FETCH_FUNC` for the functional programming fans.

More modes are coming soon.

Error handling. Exceptions

Although there are several error handling modes in PDO, the only proper one is `PDO::ERRMODE_EXCEPTION`. So, one ought to always set it this way, either by adding this line after creation of PDO instance,

```
$dbh->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
```

or as a connection option, as demonstrated in the example above. And this is *all* you need for the basic error reporting.

Reporting PDO errors

TL;DR:
Despite what all other tutorials say, **you don't need a `try..catch` operator to report PDO errors**. Catch an exception only if you have a handling scenario other than just reporting it. Otherwise just let it bubble up to a site-wide handler (note that you don't have to write one, there is a basic built-in handler in PHP, which is quite good).

The only exception (pun not intended) is the creation of the PDO instance, which in case of error might reveal the connection credentials (that would be the part of the stack trace). In order to hide them, we can wrap the connection code into a `try..catch` operator and then throw a new `ErrorException` that contains only the message but not the credentials.

A long rant on the matter:

Despite a widespread delusion, you should never catch errors to report them. A module (like a database layer) should not report its errors. This function has to be delegated to an application-wide handler. All we need is to raise an error (in the form of exception) - which we already did. That's all. Nor should you "*always wrap your PDO operations in a `try/catch`*" like the most popular tutorial from tutsplus recommends. Quite contrary, catching an exception should be rather an exceptional case (pun intended).

In fact, there is nothing special in PDO exceptions - they are errors all the same. Thus, you have to treat them exactly the same way as other errors. If you had an error handler before, you shouldn't create a dedicated one for PDO. If you didn't care - it's all right too, as PHP is good with basic error handling and will conduct PDO exceptions all right.

Exception handling is one of the problems with PDO tutorials. Being acquainted with exceptions for the first time when starting with PDO, authors consider exceptions dedicated to this library, and start diligently (but improperly) handling exceptions for PDO only. This is utter nonsense. If one paid no special attention to any exceptions before, they shouldn't have changed their habit for PDO. If one didn't use `try..catch` before, they should keep with that, eventually learning how to use exceptions and when it is suitable to catch them.

So now you can tell that the PHP manual is wrong, stating that

If your application does not catch the exception thrown from the PDO constructor, the default action taken by the zend engine is to terminate the script and display a back trace. This back trace will likely reveal the full database connection details, including the username and password.

However, **there is no such thing as "the displaying of a back trace"!** What zend engine *really* does is just convert an uncaught exception into a fatal error. And then this fatal error is treated **like any other error** - so it will be displayed only if appropriate `php.ini` directive is set. Thus, although you may or you may not catch an exception, it has absolutely nothing to do with displaying sensitive information, because it's **a totally different configuration setting** in response to this. So, do not catch PDO exceptions to report them. Instead, configure your server properly:

On a development server just turn displaying errors on:

```
ini_set('display_errors', 1);
```

While on a production server turn displaying errors off while logging errors on:

```
ini_set('display_errors', 0);
ini_set('log_errors', 1);
```

- keep in mind that *there are other errors that shouldn't be revealed to the user as well.*

Catching PDO exceptions

You may want to catch PDO errors only in two cases:

1. If you are writing a wrapper for PDO, and you want to augment the error info with some additional data, like query string. In this case, catch the exception, gather the required information, and **re-throw another Exception**.
2. If you have a **certain scenario** for handling errors in the particular part of code. Some examples are:
    - if the error can be bypassed, you can use try..catch for this. However, do not make it a habit. Empty catch in every aspect works as error suppression operator, and so equally evil it is.
    - if there is an action that has to be taken in case of failure, i.e. transaction rollback.
    - if you are waiting for a particular error to handle. In this case, catch the exception, see if the error is one you're looking for, and then handle this one. Otherwise just throw it again - so it will bubble up to the handler in the usual way.

E.g.:

```
try {
    $pdo->prepare("INSERT INTO users VALUES (NULL,?,?,?,?)")-
>execute($data);
} catch (PDOException $e) {
    $existingkey = "Integrity constraint violation: 1062 Duplicate entry";
    if (strpos($e->getMessage(), $existingkey) !== FALSE) {

        // Take some action if there is a key constraint violation, i.e. du
plicate name
    } else {
        throw $e;
    }
}
```

However, in general, no dedicated treatment for PDO exceptions is ever needed. In short, to have PDO errors properly reported:

1. Set PDO in exception mode.
2. Do not use `try..catch` to report errors.
3. Configure PHP for proper error reporting
    - on a **live** site set `display_errors=off` and `log_errors=on`
    - on a **development** site, you may want to set `display_errors=on`
    - of course, `error_reporting` has to be set to E_ALL in both cases

As a result, you will be always notified of all database errors without a single line of extra code! Further reading.

Getting row count with PDO

You don't needed it.

Although PDO offers a function for returning the number of rows found by the query, `PDOstatement::rowCount()`, you scarcely need it. Really.

If you think it over, you will see that this is a most misused function in the web. Most of time it is used not to *count* anything, but as a mere flag - just to see if there was any data returned. But for such a case you have the data itself! Just get your data, using either `fetch()` or `fetchAll()` - and it will serve as such a flag all right! Say, to see if there is any user with such a name, just select a row:

```
$stmt = $pdo->prepare("SELECT 1 FROM users WHERE name=?");
$stmt->execute([$name]);
$userExists = $stmt->fetchColumn();
```

Exactly the same thing with getting either a single row or an array with rows:

```
$data = $pdo->query("SELECT * FROM table")->fetchAll();
if ($data) {
    // You have the data! No need for the rowCount() ever!
}
```

Remember that here you don't need the *count*, the actual number of rows, but rather a boolean flag. So you got it.

Not to mention that the second most popular use case for this function should never be used at all. One should never use the `rowCount()` to count rows in database! Instead, one has to ask a database to count them, and return the result in a **single** row:

```
$count = $pdo->query("SELECT count(1) FROM t")->fetchColumn();
```

is the only proper way.

In essence:

- if you need to know how many rows in the table, use `SELECT COUNT(*)` query.

- if you need to know whether your query returned any data - check that data.
- if you still need to know how many rows has been returned by some query (though I hardly can imagine a case), then you can either use `rowCount()` or simply call `count()` on the array returned by `fetchAll()` (if applicable).

Thus you could tell that the top answer for this question on Stack Overflow is essentially pointless and harmful - a call to `rowCount()` could be never substituted with `SELECT count(*)` query - their purpose is essentially different, while running an extra query only to get the number of rows returned by other query makes absolutely no sense.

Affected rows and insert id

PDO is using the same function for returning both number of rows returned by SELECT statement and number of rows affected by DML queries - `PDOstatement::rowCount()`. Thus, to get the number of rows affected, just call this function after performing a query.

Another frequently asked question is caused by the fact that mysql won't update the row, if new value is the same as old one. Thus number of rows affected could differ from the number of rows matched by the WHERE clause. Sometimes it is required to know this latter number.

Although you can tell `rowCount()` to return the number of rows matched instead of rows affected by setting `PDO::MYSQL_ATTR_FOUND_ROWS` option to TRUE, but, as this is a connection-only option and thus you cannot change it's behavior during runtime, you will have to stick to only one mode for the application, which could be not very convenient.

Note that `PDO::MYSQL_ATTR_FOUND_ROWS` is not guaranteed to work, as it's described in the comment below.

Unfortunately, there is no PDO counterpart for the `mysql(i)_info()` function which output can be easily parsed and desired number found. This is one of minor PDO drawbacks.

An auto-generated identifier from a sequence or auto_inclement field in mysql can be obtained from the PDO::lastInsertId function. An answer to a frequently asked question, "whether this function is safe to use in concurrent environment?" is positive: yes, it is safe. Being just an interface to MySQL C API mysql_insert_id() function it's perfectly safe.

Prepared statements and LIKE clause

Despite PDO's overall ease of use, there are some gotchas anyway, and I am going to explain some.

One of them is using placeholders with `LIKE` SQL clause. At first one would think that such a query will do:

```
$stmt = $pdo->prepare("SELECT * FROM table WHERE name LIKE '%?%'");
```

but soon they will learn that it will produce an error. To understand its nature one has to understand that, like it was said above, *a placeholder have to represent a complete data literal only* - a string or a number namely. And by no means can it represent either a part of a

literal or some arbitrary SQL part. So, when working with LIKE, we have to prepare our **complete literal** first, and then send it to the query the usual way:

```
$search = "%$search%";
$stmt   = $pdo->prepare("SELECT * FROM table WHERE name LIKE ?");
$stmt->execute([$search]);
$data = $stmt->fetchAll();
```

Prepared statements and IN clause

Just like it was said above, it is impossible to substitute an arbitrary query part with a placeholder. Any string you bind through a placeholder will be put into query as a single string literal. For example, a *string* `'1,2,3'` will be bound as a *string*, resulting in

```
SELECT * FROM table WHERE column IN ('1,2,3')
```

making SQL to search for just *one* value.

To make it right, one needs separated values, to make a query look like

```
SELECT * FROM table WHERE column IN ('1','2','3')
```

Thus, for the comma-separated values, like for `IN()` SQL operator, one must create a set of `?`s manually and put them into the query:

```
$arr = [1,2,3];
$in  = str_repeat('?,', count($arr) - 1) . '?';
$sql = "SELECT * FROM table WHERE column IN ($in)";
$stm = $db->prepare($sql);
$stm->execute($arr);
$data = $stm->fetchAll();
```

Not very convenient, but compared to mysqli it's *amazingly* concise.

In case there are other placeholders in the query, you could use `array_merge()` function to join all the variables into a single array, adding your other variables in the form of arrays, in the order they appear in your query:

```
$arr = [1,2,3];
$in  = str_repeat('?,', count($arr) - 1) . '?';
$sql = "SELECT * FROM table WHERE foo=? AND column IN ($in) AND bar=? AND b
az=?";
$stm = $db->prepare($sql);
$params = array_merge([$foo], $arr, [$bar, $baz]);
$stm->execute($params);
$data = $stm->fetchAll();
```

In case you are using named placeholders, the code would be a little more complex, as you have to create a sequence of the named placeholders, e.g. `:id0,:id1,:id2`. So the code would be:

```
// other parameters that are going into query
$params = ["foo" => "foo", "bar" => "bar"];
```

```php
$ids = [1,2,3];
$in = "";
foreach ($ids as $i => $item)
{
    $key = ":id".$i;
    $in .= "$key,";
    $in_params[$key] = $item; // collecting values into key-value array
}
$in = rtrim($in,","); // :id0,:id1,:id2

$sql = "SELECT * FROM table WHERE foo=:foo AND id IN ($in) AND bar=:bar";
$stm = $db->prepare($sql);
$stm->execute(array_merge($params,$in_params)); // just merge two arrays
$data = $stm->fetchAll();
```

Luckily, for the named placeholders we don't have to follow the strict order, so we can merge our arrays in any order.

Prepared statements and table names

On Stack Overflow I've seen overwhelming number of PHP users implementing the most fatal PDO code, thinking that only data values have to be protected. But of course it is not.

Unfortunately, PDO has no placeholder for identifiers (table and field names), so a developer must manually format them.

For **mysql** to format an identifier, follow these two rules:

- Enclose identifier in backticks.
- Escape backticks inside by doubling them.

so the code would be:

```php
$table = "`".str_replace("`","``",$table)."`";
```

After such formatting, it is safe to insert the `$table` variable into query.

For other databases rules will be different but it is essential to understand that using only delimiters is not enough - delimiters themselves should be escaped.

It is also important to always check dynamic identifiers against a list of allowed values. Here is a brief example:

```php
$orders  = ["name","price","qty"]; //field names
$key     = array_search($_GET['sort'],$orders); // see if we have such a na
me
$orderby = $orders[$key]; //if not, first one will be set automatically. sm
art enuf :)
$query   = "SELECT * FROM `table` ORDER BY $orderby"; //value is safe
```

Or, extending this approach for the INSERT/UPDATE statements (as Mysql supports SET for both),

```
$data = ['name' => 'foo', 'submit' => 'submit']; // data for insert
$allowed = ["name", "surname", "email"]; // allowed fields
$values = [];
$set = "";
foreach ($allowed as $field) {
    if (isset($data[$field])) {
        $set.="`".str_replace("`", "``", $field)."`". "=:$field, ";
        $values[$field] = $data[$field];
    }
}
$set = substr($set, 0, -2);
```

This code will produce the correct sequence for SET operator that will contain only allowed fields and placeholders like this:

```
`name`=:foo
```

as well as `$values` array for `execute()`, which can be used like this

```
$stmt = $pdo->prepare("INSERT INTO users SET $set");
$stmt->execute($values);
```

Yes, it looks extremely ugly, but that is all PDO can offer.

A problem with LIMIT clause

Another problem is related to the SQL `LIMIT` clause. When in emulation mode (which is on by default), PDO substitutes placeholders with actual data, instead of sending it separately. And with "lazy" binding (using array in `execute()`), PDO treats every parameter as a string. As a result, the prepared `LIMIT ?,?` query becomes `LIMIT '10', '10'` which is invalid syntax that causes query to fail.

There are two solutions:

One is turning emulation off (as MySQL can sort all placeholders properly). To do so one can run this code:

```
$conn->setAttribute( PDO::ATTR_EMULATE_PREPARES, false );
```

And parameters can be kept in `execute()`:

```
$conn->setAttribute( PDO::ATTR_EMULATE_PREPARES, false );
$stmt = $pdo->prepare('SELECT * FROM table LIMIT ?, ?');
$stmt->execute([$offset, $limit]);
$data = $stmt->fetchAll();
```

Another way would be to bind these variables explicitly while setting the proper param type:

```
$stmt = $pdo->prepare('SELECT * FROM table LIMIT ?, ?');
$stmt->bindParam(1, $offset,PDO::PARAM_INT);
$stmt->bindParam(2, $limit,PDO::PARAM_INT);
$stmt->execute();
$data = $stmt->fetchAll();
```

One peculiar thing about `PDO::PARAM_INT`: for some reason it does not enforce the type casting. Thus, using it on a number that has a string type will cause the aforementioned error:

```
$stmt = $pdo->prepare("SELECT 1 LIMIT ?");
$stmt->bindValue(1, "1", PDO::PARAM_INT);
$stmt->execute();
```

But change `"1"` in the example to `1` - and everything will go smooth.

Transactions

To successfully run a transaction, you have to make sure that error mode is set to exceptions, and learn three canonical methods:

- `beginTransaction()` to start a transaction
- `commit()` to commit one
- `rollback()` to cancel all the changes you made since transaction start.

Exceptions are essential for transactions because they can be caught. So in case one of the queries failed, the execution will be stopped and moved straight to the catch block, where the whole transaction will be rolled back.

So a typical example would be like

```
try {
    $pdo->beginTransaction();
    $stmt = $pdo->prepare("INSERT INTO users (name) VALUES (?)");
    foreach (['Joe','Ben'] as $name)
    {
        $stmt->execute([$name]);
    }
    $pdo->commit();
}catch (Exception $e){
    $pdo->rollback();
    throw $e;
}
```

Please note the following important things:

- PDO error reporting mode should be set to `PDO::ERRMODE_EXCEPTION`
- you have catch an `Exception`, not `PDOException`, as it doesn't matter what particular exception aborted the execution.
- you should re-throw an exception after rollback, to be notified of the problem the usual way.
- also make sure that a table engine supports transactions (i.e. for Mysql it should be InnoDB, not MyISAM)
- there are no Data definition language (DDL) statements that define or modify database schema among queries in your transaction, as such a query will cause an implicit commit

Calling stored procedures in PDO

There is one thing about stored procedures any programmer stumbles upon at first: every stored procedure always returns **one extra result set**: one (or many) results with actual data and one just empty. Which means if you try to call a procedure and then proceed to another query, then **"Cannot execute queries while other unbuffered queries are active"** error will occur, because you have to clear that extra empty result first. Thus, after calling a stored procedure that is intended to return only one result set, just call `PDOStatement::nextRowset()` once (of course after fetching all the returned data from statement, or it will be discarded):

```
$stmt = $pdo->query("CALL bar()");
$data = $stmt->fetchAll();
$stmt->nextRowset();
```

While for the stored procedures returning many result sets the behavior will be the same as with multiple queries execution:

```
$stmt = $pdo->prepare("CALL foo()");
$stmt->execute();
do {
    $data = $stmt->fetchAll();
    var_dump($data);
} while ($stmt->nextRowset() && $stmt->columnCount());
```

However, as you can see here is another trick have to be used: remember that extra result set? It is so essentially *empty* that even an attempt to fetch from it will produce an error. So, we cannot use just `while ($stmt->nextRowset())`. Instead, we have to check also for empty result. For which purpose `PDOStatement::columnCount()` is just excellent.

This feature is one of essential differences between old mysql ext and modern libraries: after calling a stored procedure with `mysql_query()` there was no way to continue working with the same connection, because there is no `nextResult()` function for `mysql ext`. One had to close the connection and then open a new one again in order to run other queries after calling a stored procedure.

Calling a stored procedure is a rare case where `bindParam()` use is justified, as it's the only way to handle `OUT` and `INOUT` parameters. The example can be found in the corresponding manual chapter. However, **for mysql it doesn't work.** You have to resort to an SQL variable and an extra call.

Note that for the different databases the syntax could be different as well. For example, to run a sored procedure against Microsoft SQL server, use the following format

```
$stmt = $pdo->prepare("EXEC stored_procedure ? ?");
```

where ? marks are placeholders. Note that no braces should be used in the call.

Running multiple queries with PDO

When in emulation mode, PDO can run mutiple queries in the same statement, either via query() or `prepare()/execute()`. To access the result of consequent queries one has to use `PDOStatement::nextRowset()`:

```php
$stmt = $pdo->prepare("SELECT ?;SELECT ?");
$stmt->execute([1,2]);
do {
    $data = $stmt->fetchAll();
    var_dump($data);
} while ($stmt->nextRowset());
```

Within this loop you'll be able to gather all the related information from the every query, like affected rows, auto-generated id or errors occurred.

It is important to understand that at the point of `execute()` PDO will report the error **for the first query only**. But if error occurred at any of consequent queries, to get that error one has to iterate over results. Despite some ignorant opinions, PDO can not and should not report all the errors at once. Some people just cannot grasp the problem at whole, and don't understand that error message is not the only outcome from the query. There could be a dataset returned, or some metadata like insert id. To get these, one has to iterate over resultsets, one by one. But to be able to throw an error immediately, PDO would have to iterate automatically, and thus **discard some results**. Which would be a clear nonsense.

Unlike `mysqli_multi_query()` PDO doesn't make an asynchronous call, so you can't "fire and forget" - send bulk of queries to mysql and close connection, PHP will wait until last query gets executed.

Emulation mode. PDO::ATTR_EMULATE_PREPARES

One of the most controversial PDO configuration options is `PDO::ATTR_EMULATE_PREPARES`. What does it do? PDO can run your queries in two ways:

1. It can use a **real** or native prepared statement:
   When prepare() is called, your query with placeholders gets sent to mysql as is, with all the question marks you put in (in case named placeholders are used, they are substituted with ?s as well), while actual data goes later, when execute() is called.
2. It can use **emulated** prepared statement, when your query is sent to mysql as proper SQL, with all the data in place, **properly formatted**. In this case only one roundtrip to database happens, with `execute()` call. For some drivers (including mysql) emulation mode is turned `ON` by default.

Both methods has their drawbacks and advantages but, and - I have to stress on it - both being **equally secure**, if used properly. Despite rather appealing tone of the popular article on Stack Overflow, in the end it says that **if you are using supported versions of PHP and MySQL properly, you are 100% safe**. All you have to do is to set encoding in the DSN, as it shown in the example above, and your emulated prepared statements will be as secure as real ones.

Note that when native mode is used, **the data is never appears in the query**, which is parsed by the engine as is, with all the placeholders in place. If you're looking into Mysql query log for your prepared query, you have to understand that it's just an artificial query that has been created solely for logging purpose, but not a real one that has been executed.

Other issues with emulation mode as follows:

When emulation mode is turned ON

one can use a handy feature of named prepared statements - a placeholder with same name could be used any number of times in the same query, while corresponding variable have to be bound only once. For some obscure reason this functionality is disabled when emulation mode is off:

```
$stmt = $pdo-
>prepare("SELECT * FROM t WHERE foo LIKE :search OR bar LIKE :search");
$stmt->execute(['search'] => "%$search%");`
```

Also, when emulation is `ON`, PDO is able to run multiple queries in one prepared statement.

Also, as native prepared statements support only certain query types, you can run some queries with prepared statements only when emulation is `ON`. The following code will return table names in emulation mode and error otherwise:

```
$stmt = $pdo->prepare("SHOW TABLES LIKE ?");
$stmt->execute(["%$name%"]);
var_dump($stmt->fetchAll());
```

When emulation mode is turned OFF

One could bother not with parameter types, as mysql will sort all the types properly. Thus, even string can be bound to LIMIT parameters, as it was noted in the corresponding chapter.

Also, this mode will allow to use the advantage of single prepare-multiple execute feature.

It's hard to decide which mode have to be preferred, but for usability sake I would rather turn it `OFF`, to avoid a hassle with `LIMIT` clause. Other issues could be considered negligible in comparison.

Mysqlnd and buffered queries. Huge datasets.

Recently all PHP extensions that work with mysql database were updated based on a low-level library called `mysqlnd`, which replaced old `libmysql` client. Thus some changes in the PDO behavior, mostly described above and one that follows:

There is one thing called buffered queries. Although you probably didn't notice it, you were using them all the way. Unfortunately, here are bad news for you: unlike old PHP versions, where you were using buffered queries virtually for free, modern versions built upon mysqlnd driver won't let you to do that anymore:

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.**

The whole thing is about a resultset, which stands for all the data found by the query.

When your SELECT query gets executed, there are two ways to deliver the results in your script: buffered and unbuffered one. When buffered method is used, all the data returned by the query **gets copied in the script's memory** at once. While in unbuffered mode a database server feeds the found rows one by one.

So you can tell that in buffered mode a resultset is always burdening up the memory on the server *even if fetching weren't started at all*. Which is why it is not advisable to select huge datasets if you don't need all the data from it.

Nonetheless, when old libmysql-based clients were used, this problem didn't bother PHP uers too much, because the memory consumed by the resultset didn't count in the the `memory_get_usage()` and `memory_limit`.

But with mysqlnd things got changed, and the resultset returned by the buffered query will be count towards both `memory_get_usage()` and `memory_limit`, no matter which way you choose to get the result:

```
$pdo->setAttribute(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, FALSE);
$stmt = $pdo->query("SELECT * FROM Board");
$mem = memory_get_usage();
while($row = $stmt->fetch());
echo "Memory used: ".round((memory_get_usage() -
 $mem) / 1024 / 1024, 2)."M\n";

$pdo->setAttribute(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, TRUE);
$stmt = $pdo->query("SELECT * FROM Board");
$mem = memory_get_usage();
while($row = $stmt->fetch());
echo "Memory used: ".round((memory_get_usage() -
 $mem) / 1024 / 1024, 2)."M\n";
```

will give you (for my data)

```
Memory used: 0.02M
Memory used: 2.39M
```

which means that with buffered query the memory is consumed **even if you're fetching rows one by one!**

So, keep in mind that if you are selecting a really huge amount of data, always set `PDO::MYSQL_ATTR_USE_BUFFERED_QUERY` to `FALSE`.

Of course, there are some drawbacks, two minor ones:

1. With unbuffered query you can't use `rowCount()` method (which is useless, as we learned above)
2. Moving (seeking) the current resultset internal pointer back and forth (which is useless as well).

And a rather important one:

1. While an unbufferered query is active, you cannot execute any other query using the same connection, so use this mode wisely.