Select

Alle select-Anweisungen in parametrisierten Abfragen beginnen ungefähr gleich. Es gibt jedoch einen wesentlichen Unterschied zum tatsächlichen Speichern und Abrufen der Ergebnisse. Die beiden vorhandenen Methoden sind `get_result()` und `bind_result()`.

One Row

- `$result->fetch_assoc()` - Fetch an associative array
- `$result->fetch_row()` - Fetch a numeric array
- `$result->fetch_object()` - Fetch an object array

All Rows

- `$result->fetch_all(MYSQLI_ASSOC)` - Fetch an associative array
- `$result->fetch_all(MYSQLI_NUM)` - Fetch a numeric array

get_result()

Dies ist die vielseitigere von beiden, da sie für jedes Szenario verwendet werden kann. Es ist zu beachten, dass hierfür mysqlnd erforderlich ist, das seit 5.3 in PHP enthalten ist und seit 5.4 der standardmäßige native Treiber ist, wie hier angegeben. Ich bezweifle, dass viele Leute ältere Versionen verwenden, daher sollten Sie sich im Allgemeinen an `get_result()` halten.
Dies macht im Wesentlichen die reguläre, nicht vorbereitete mysqli_result-API verfügbar. Das heißt, wenn Sie $result = `get_result()` ausführen, können Sie es genauso verwenden, wie Sie $result = $ mysqli-> query() verwenden würden.
Jetzt können Sie die folgenden Methoden verwenden, um jeweils eine Zeile oder alle gleichzeitig abzurufen. Hier sind nur einige der häufigsten, aber Sie können sich die gesamte mysqli_result-Klasse für alle ihre Methoden ansehen.

```
$stmt = $mysqli->prepare("SELECT * FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$result = $stmt->get_result();
if($result->num_rows === 0) exit('No rows');
while($row = $result->fetch_assoc())                    // One Row
{
  $ids[] = $row['id'];
  $names[] = $row['name'];
  $ages[] = $row['age'];
}
var_export($ages);
$stmt->close();
```

bind_result()

Sie fragen sich vielleicht, warum Sie `bind_result()` überhaupt verwenden sollten? Ich persönlich finde es in jedem Szenario weitaus schlechter als `get_result()`, außer wenn eine einzelne Zeile in separaten Variablen abgerufen wird. Bevor `get_result()` existierte und mysqlnd in PHP integriert wurde, war dies Ihre einzige Option, weshalb möglicherweise viel Legacy-Code verwendet wird.
Das nervigste an der Verwendung von `bind_result()` ist, dass Sie jede einzelne Spalte, die Sie auswählen, binden und dann die Werte in einer Schleife durchlaufen müssen. Dies ist offensichtlich nicht ideal für eine Vielzahl von Werten oder zur Verwendung mit *. Die Verwendung des Sternselektors mit `bind_result()` ist besonders ärgerlich, da Sie nicht einmal wissen, was diese Werte sind, ohne in der Datenbank nachzuschauen. Darüber hinaus ist Ihr Code bei Änderungen an der Tabelle äußerst unhaltbar. Dies spielt normalerweise keine Rolle, da Sie den Platzhalter-Selektor ohnehin nicht im Produktionsmodus verwenden sollten.

```
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$stmt->store_result();
if($stmt->num_rows === 0) exit('No rows');
$stmt->bind_result($idRow, $nameRow, $ageRow);
while($stmt->fetch())                                   // One Row
{
  $ids[] = $idRow; $names[] = $nameRow; $ages[] = $ageRow;
}
var_export($ids);
$stmt->close();
```

Fetch Associative Array

I find this to be the most common use case typically. I will also be utilizing chaining in the following, though that's obviously not necessary.

```
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$arr = $stmt->get_result()->fetch_all(MYSQLI_ASSOC); // All Rows
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

If you need to modify the result set, then you should probably use a while loop with `fetch_assoc()` and fetch each row one at a time.

```
$arr = [];
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$result = $stmt->get_result();
while($row = $result->fetch_assoc())                    // One Row
{
  $arr[] = $row;
}
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

Output:

```
['id' => 27, 'name' => 'Jessica', 'age' => 27]
```

Sie können dies auch mit `bind_result()` tun, obwohl es eindeutig nicht dafür entwickelt wurde. Hier ist eine clevere Lösung, obwohl ich persönlich der Meinung bin, dass es cool ist zu wissen, dass dies möglich ist, aber realistisch gesehen nicht verwendet werden sollte.

Fetch Numeric Array

Dies folgt dem gleichen Format wie ein assoziatives Array. Um das gesamte Array in einem Befehl ohne Schleife abzurufen, würden Sie verwenden `mysqli_result->fetch_all(MYSQLI_NUM)`. Wenn Sie die Ergebnisse in einer Schleife abrufen, müssen Sie verwenden `mysqli_result->fetch_row()`.

```
$stmt = $mysqli->prepare("SELECT location, favorite_color, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$arr = $stmt->get_result()->fetch_all(MYSQLI_NUM);    // All Rows
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

And of course, the while loop adaptation.

```
$arr = [];
$stmt = $mysqli->prepare("SELECT location, favorite_color, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$result = $stmt->get_result();
while($row = $result->fetch_row())                    // One Row
{ $arr[] = $row; }
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

Output:

```
[
  ['Boston', 'green', 28],
  ['Seattle', 'blue', 49],
  ['Atlanta', 'pink', 24]
]
```

Fetch Single Row

I personally find it simpler to use `bind_result()` when I know for fact that I will only be fetching one row, as I can access the variables in a cleaner manner.

```
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$stmt->store_result();
if($stmt->num_rows === 0) exit('No rows');
$stmt->bind_result($id, $name, $age);
$stmt->fetch();                                 // One Row
echo $name; //Output: 'Ryan'
$stmt->close();
```

Now you can use just simply use the variables in `bind_result()`, like `$name` since you know they will only contain one value, not an array.

Here's the get_result() version:

```
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$arr = $stmt->get_result()->fetch_assoc();          // One Row
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

You would then use the variable as `$arr['id']` for example.

Output:

```
['id' => 36, 'name' => 'Kevin', 'age' => 39]
```

Fetch Array of Objects

This very similar to fetching an associative array. The only main difference is that you'll be accessing it like `$arr[0]->age`. Also, in case you didn't know, objects are pass by value, while arrays are by reference.

```
$arr = []
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE id = ?");
$stmt->bind_param("s", $_SESSION['id']);
$stmt->execute();
$result = $stmt->get_result();
while($row = $result->fetch_object())               // One Row
{
  $arr[] = $row;
}
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

Output:

```
[
  stdClass Object ['id' => 27, 'name' => 'Jessica', 'age' => 27],
  stdClass Object ['id' => 432, 'name' => 'Jimmy', 'age' => 19]
]
```

You can even add property values to an existing class as well. However, it should be noted that there is a potential gotcha, according to this comment in the PHP docs. The problem is that if you have a default value in your constructor with a duplicate variable name, it will fetch the object first and *then* set the constructor value, therefore overwriting the fetched result. Weirdly enough, there was a "bug" from PHP 5.6.21 to 7.0.6 where this wouldn't happen. Even though this violates principles of OOP, some people would like this feature, even though it was bug in certain versions. Something like `PDO::FETCH_PROPS_LATE` in PDO should be implemented in MySQLi to give you the option to choose.

```
class myClass {}
$arr = [];
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE id = ?");
$stmt->bind_param("s", $_SESSION['id']);
$stmt->execute();
$result = $stmt->get_result();
while($row = $result->fetch_object('myClass'))      // One Row
{ $arr[] = $row; }
if(!$arr) exit('No rows');
```

```
var_export($arr);
$stmt->close();
```

As the comment states, this is how you would do it correctly. All you need is a simple if condition to check if the variable equals the constructor value — if it doesn't, just don't set it in the constructor. This is essentially the same as using `PDO::FETCH_PROPS_LATE` in PDO.

```
class myClass {
  private $id;
  public function __construct($id = 0) {
    if($this->id === 0) $this->id = $id;
  }
}
$arr = [];
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE id = ?");
$stmt->bind_param("s", $_SESSION['id']);
$stmt->execute();
$result = $stmt->get_result();
while($row = $result->fetch_object('myClass'))        // One Row
{
  $arr[] = $row;
}
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

Another unexpected, yet potentially useful behavior of using `fetch_object('myClass')` is that you can modify private variables. I'm really not sure how I feel about this, as this seems to violate principles of encapsulation.

Conclusion (Fazit)

bind_result()

`Am besten zum Abrufen einer einzelnen Zeile ohne zu viele Spalten oder *; extrem unelegant für assoziative Arrays.`

get_result()

`ist die bevorzugte für fast jeden Anwendungsfall.`

Like

You would probably think that you could do something like:

```
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE Name LIKE %?%");
```

But this is not allowed. The `?` placeholder must be the entire string or integer literal value. This is how you would do it correctly.

```
$search = "%{$_POST['search']}%";
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name LIKE ?");
$stmt->bind_param("s", $search);
$stmt->execute();
$arr = $stmt->get_result()->fetch_all(MYSQLI_ASSOC); // All Rows
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

Where In Array

This is definitely something I'd like to see improved in MySQLi. For now, using MySQLi prepared statements with `WHERE IN` is possible, but feels a little long-winded.

Side note: The following two examples use the splat operator for argument unpacking, which requires PHP 5.6+. If you are using a version lower than that, then you can substitute it with `call_user_func_array()`.

```
$inArr = [12, 23, 44];
$clause = implode(',', array_fill(0, count($inArr), '?')); //create 3 question marks
$types = str_repeat('i', count($inArr)); //create 3 ints for bind_param
$stmt = $mysqli->prepare("SELECT id, name FROM myTable WHERE id IN ($clause)");
$stmt->bind_param($types, ...$inArr);
$stmt->execute();
$resArr = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);// All Rows
if(!$resArr) exit('No rows');
var_export($resArr);
$stmt->close()
```

With Other Placeholders

The first example showed how to use the `WHERE IN` clause with dummy placeholder solely inside of it. What if you wanted to use other placeholders in different places?

```
$inArr = [12, 23, 44];
$clause = implode(',', array_fill(0, count($inArr), '?')); //create 3 question marks
$types = str_repeat('i', count($inArr)); //create 3 ints for bind_param
$types .= 'i'; //add 1 more int type
$fullArr = array_merge($inArr, [26]); //merge WHERE IN array with other value(s)
$stmt = $mysqli->prepare("SELECT id, name FROM myTable WHERE id IN ($clause) AND age > ?");
$stmt->bind_param($types, ...$fullArr); //4 placeholders to bind
$stmt->execute();
$resArr = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);// All Rows
if(!$resArr) exit('No rows');
var_export($resArr);
$stmt->close();
```

Multiple Prepared Statements in Transactions

This might seem odd why it would even warrant its own section, as you can literally just use prepared statements one after another. While this will certainly work, this does not ensure that your queries are atomic. This means that if you were to run ten queries, and one failed, the other nine would still succeed. If you want your SQL queries to execute only if they all succeeded, then you must use transactions.

```
try {
  $mysqli->autocommit(FALSE); //turn on transactions
  $stmt1 = $mysqli->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
  $stmt2 = $mysqli->prepare("UPDATE myTable SET name = ? WHERE id = ?");
  $stmt1->bind_param("si", $_POST['name'], $_POST['age']);
  $stmt2->bind_param("si", $_POST['name'], $_SESSION['id']);
  $stmt1->execute();
  $stmt2->execute();
  $stmt1->close();
  $stmt2->close();
  $mysqli->autocommit(TRUE); //turn off transactions + commit queued queries
} catch(Exception $e) {
  $mysqli->rollback(); //remove all queries from queue if error (undo)
  throw $e;
}
```

Reuse Same Template, Different Values

```
try {
  $mysqli->autocommit(FALSE); //turn on transactions
  $stmt = $mysqli->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
  $stmt->bind_param("si", $name, $age);
  $name = 'John';
  $age = 21;
  $stmt->execute();
  $name = 'Rick';
  $age = 24;
  $stmt->execute();
  $stmt->close();
  $mysqli->autocommit(TRUE); //turn off transactions + commit queued queries
} catch(Exception $e) {
  $mysqli->rollback(); //remove all queries from queue if error (undo)
  throw $e;
}
```

Error Handling

Fatal error: Uncaught Error: Call to a member function bind_param() on boolean

Jeder, der von MySQLi vorbereitete Anweisungen verwendet hat, hat diese Nachricht irgendwann gesehen, aber was bedeutet das? So ziemlich gar nichts. Wie können Sie das beheben, fragen Sie sich vielleicht? Vergessen Sie zunächst nicht, die Ausnahmebehandlung zu aktivieren, anstatt die Fehlerbehandlung mysqli_report (MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT), wenn Sie eine neue Verbindung erstellen.

Exception Handling

Alle mysqli-Funktionen geben bei einem Fehler false zurück, sodass Sie einfach die Richtigkeit jeder Funktion überprüfen und Fehler mit $mysqli->error melden können. Dies ist jedoch sehr langwierig, und es gibt eine elegantere Möglichkeit, dies zu tun, wenn Sie die interne Berichterstellung aktivieren. Ich empfehle dies auf diese Weise, da es von der Entwicklung bis zur Produktion viel portabler ist.

Dies kann auch in der Produktion verwendet werden, sofern für alle Fehler ein Fehlerprotokoll eingerichtet ist. Dies muss in der php.ini eingestellt werden. Bitte melden Sie Fehler niemals direkt auf Ihrer Site in der Produktion. Sie werden sich für solch einen dummen Fehler treten. Die Platzierung von mysqli_report() ist ebenfalls wichtig. Wenn Sie es vor dem Erstellen einer neuen Verbindung platzieren, wird auch Ihr Kennwort ausgegeben. Andernfalls wird nur alles nachher gemeldet, wie Ihre Abfragen.

So sollte Ihre php.ini-Datei in der Produktion aussehen: Führen Sie sowohl display_errors = Off als auch log_errors = On aus. Denken Sie auch daran, dass jede Seite eigentlich nur einen einzigen globalen Try / Catch-Block verwenden sollte, anstatt jede Abfrage einzeln zu verpacken. Die einzige Ausnahme hiervon sind Transaktionen, die verschachtelt wären, aber eine eigene Ausnahme auslösen, sodass der globale try/catch sie "fangen" kann

```
try {
  $stmt = $mysqli->prepare("DELETE FROM myTable WHERE id = ?");
  $stmt->bind_param("i", $_SESSION['id']);
  $stmt->execute();
  $stmt->close();

  $stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
  $stmt->bind_param("s", $_POST['name']);
  $stmt->execute();
  $arr = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);
  $stmt->close();

  try {
    $mysqli->autocommit(FALSE); //turn on transactions
    $stmt = $mysqli->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
    $stmt->bind_param("si", $name, $age);
    $name = 'John';
    $age = 21;
    $stmt->execute();
    $name = 'Rick';
    $age = 24;
    $stmt->execute();
    $stmt->close();
    $mysqli->autocommit(TRUE); //turn off transactions + commit queued queries
  } catch(Exception $e) {
    $mysqli->rollback(); //remove all queries from queue if error (undo)
    throw $e;
  }
} catch (Exception $e) {
  error_log($e);
  exit('Error message for user to understand');
}
```

Custom Exception Handler

As stated earlier, you can alternatively use `set_exception_handler()` on each page (or a global redirect). This gets rid of the layer of curly brace nesting. If you are using transactions, you should still use a try catch with that, but then throw your own exception.

```
set_exception_handler(function($e) {
  error_log($e);
  exit('Error deleting');
});
$stmt = $mysqli->prepare("DELETE FROM myTable WHERE id = ?");
$stmt->bind_param("i", $_SESSION['id']);
$stmt->execute();
$stmt->close();
```

Gotcha with Exception Handling

You'd expect for all MySQLi errors to be converted to exceptions with `mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT)`. Oddly enough, I noticed that it still gave me a warning error when `bind_param()` had too many or too little bound variables or types. The message outputted is as follows:

Warning: mysqli_stmt::bind_param(): Number of variables doesn't match number of parameters in prepared statement

A solution to this is to use a global error handler to trigger an exception. An example of this could be:

```
set_error_handler(function($errno, $errstr, $errfile, $errline) {
  throw new Exception("$errstr on line $errline in file $errfile");
});
```

This only happened on runtime warnings, but I converted all errors to exceptions. I see no problem doing this, but there are some people who are strongly against it.

Some Extras

Do I Need $stmt->close()?

```
Gute Frage. Sowohl $mysqli->close() als auch $stmt->close() haben im Wesentlichen den
gleichen Effekt. Ersteres schließt die MySQLi-Verbindung, während letzteres die vorbereitete
Anweisung schließt. Beide sind in den meisten Fällen im Allgemeinen nicht einmal
erforderlich, da beide geschlossen werden, sobald die Ausführung des Skripts ohnehin
abgeschlossen ist. Es gibt auch eine Funktion zum einfachen Freigeben des mit dem MySQLi-
Ergebnis bzw. der vorbereiteten Anweisung verknüpften Speichers:
$result->free() und $stmt->free(). Ich selbst werde es wahrscheinlich nie verwenden, aber
wenn Sie interessiert sind, hier ist das für das Ergebnis und für die parametrisierte
Abfrage. Folgendes sollte ebenfalls beachtet werden: Sowohl $stmt->close() als auch das Ende
der Ausführung des Skripts geben den Speicher trotzdem frei.

Endgültiges Urteil: Normalerweise mache ich nur $mysqli->close() und $stmt->close(), obwohl
man argumentieren kann, dass es ein wenig überflüssig ist. Wenn Sie vorhaben, dieselbe
Variable $stmt erneut für andere vorbereitete Anweisungen zu verwenden, müssen Sie sie
entweder schließen oder einen anderen Variablennamen wie $stmt2 verwenden. Schließlich habe
ich nie die Notwendigkeit gefunden, sie einfach zu befreien, ohne sie zu schließen
```

Classes: mysqli vs. mysqli_stmt vs. mysqli_result

One thing you may have realized along the way is that there are certain methods that exist in two of the classes, like an alias almost. I personally believe it would be better to only have one version, like in PDO, to avoid confusion.

- **`mysqli::$affected_rows`** or **`mysqli_stmt::$affected_rows`** - Belongs to `mysqli_stmt`. Works the same with either, but will be an error if called *after* the statement is closed with either method
- **`mysqli_result::$num_rows`** or **`mysqli_stmt::$num_rows`** - `$result->num_rows` can only be used with `get_result()`, while `$stmt->num_rows` can only be used with `bind_result()`.
- **`mysqli::$insert_id`** or **`mysqli_stmt::$insert_id`** - Belongs to `mysqli`. Better to use `$mysqli->insert_id`, since it will still work even after `$stmt->close()` is used. There's also a note on the PHP docs from 2011 stating that `$stmt->insert_id` will only get the first executed query. I tried this on my current version of 7.1 and this doesn't seem to be the case. The recommended one to use is the mysqli class version anyway.

So Using Prepared Statements Means I'm Safe From Attackers?

While you are safe from SQL injection, you still need validate and sanitize your user-inputted data. You can use a function like filter_var() to validate *before* inserting it into the database and htmlspecialchars() to sanitize *after* retrieving it.