High Performance mit *PHP & MySQL*

von
Thomas Burgard
Thomas Burgard (BURGARDsoft) Softwareentwicklung & Webdesign

Versionsstand: 1
Datum: 16.05.2012

Thomas Burgard Softwareentwicklung & Webdesign

BURGARDsoft
Softwareentwicklung & Webdesign

Telefon: +49 89 540 70 710
Telefax: +49 89 540 70 711
E-Mail: *thomas.burgard@burgardsoft.de*
Internet: **www.burgardsoft.de**

Inhaltsverzeichnis

1 Einführung

Websites, die stark frequentiert werden, sollten in jedem Augenblick optimale Ladezeiten für den Besucher aufweisen. Kommt PHP als objektorientierte Interpreter- Sprache zum Einsatz, muss besonderes Know How mitgebracht werden. Dieser Artikel gibt einen Überblick über die Performance-Maßnahmen, -Methoden, -Produkten und -Werkzeugen in der PHP- Programmierung.

2 Performance-Maßnahmen in PHP

2.1 Was bedeutet Performance eigentlich?

Wenn man Berichte über Performance einer Software liest, wird meistens über die Ablaufgeschwindigkeit eines Scriptes geschrieben, d.h. je schneller das PHP-Script desto besser die Performance. Prinzipiell stimmt das auch, es gibt jedoch Situationen, in denen Kompromisse geschlossen werden müssen:

- Für die Performance-Optimierung wird ein “Caching” verwendet, so dass das PHP-Script direkt aus dem Speicher geladen wird.
  Vorteil: Ladegeschwindigkeit des PHP-Scriptes ist schnell.
  Nachteil: Die Daten können schnell veraltet sein.
- Ein PHP-Script lädt alles in den Speicher.
  Vorteil: Ablaufgeschwindigkeit des PHP-Scriptes ist schnell.
  Nachteil: Skalierbarkeit fehlt.
  Besser wäre in diesem Fall ein Laden der Daten in so genannte “Chunks”. Ein Kompromiss zwischen Geschwindigkeit und Skalierbarkeit wäre hierbei gegeben.

Letztendlich muss der PHP-Softwareentwickler für seine Anwendung genau abwägen, welche Kompromisse sinnvoll und notwendig sind.

Performance - eine bessere Definition
Performance ist also immer ein

- Kompromiss zwischen Ablauf- bzw. Ladegeschwindigkeit und Datenaktualität
- Kompromiss zwischen Ablauf- bzw. Ladegeschwindigkeit und Skalierbarkeit

3 Performantes programmieren in PHP

3.1 PHP Code Optimierungsmaßnahmen

PHP ist die derzeit meist genutzte Programmiersprache für die Server seitige Programmierung für dynamische Webseiten. PHP ist äußerst mächtig und verfügt über ein große Anzahl von Funktionen. Um ein Server seitiges PHP-Script performant zu schreiben, ist einiges zu beachten. Dieses Kapitel gibt einen Überblick über die Möglichkeiten, welche PHP dafür bietet.

- Die allererste und wichtigste Maßnahme zur Performance-Steigerung in PHP ist das ***saubere und fehlerfreie Programmierung auf Basis eines optimalen Designs***.

  Ein „optimales Design richtet sich auch an die Anforderungen (z.B. Performance-Anforderungen) der Software. Der Einsatz von entsprechenden Design-Patterns kann in vielen Fällen Probleme besser lösen und auch die Performance steigern. Mehr Informationen zum Thema „Design-Pattern“ im Kapitel 7.

- Wenn möglich, eine a***ktuelle Version von PHP verwenden***.

- ***Wenn möglich mehr statische HTML-Seiten verwenden als PHP-Scripte***. Die Ladezeit eines PHP-Scriptes ist deutlich länger als eine statische HTML-Seite.

- Verwende ***Autoloading*** zum Laden der PHP-Dateien in einem Script. Mit ***Autoloading*** werden nur die wirklich benötigten PHP-Dateien für die Verarbeitung geladen.

- ***Static***-Methoden sind schneller als normale Klassen-Methoden, d.h. wenn es sinnvoll und möglich ist, dann ***Static***-Methoden verwenden (*gut überlegen*). Der Grund ist der, dass bei ***Static***-Methoden kein Klassen-Objekt erzeugt werden muss, was natürlich mehr Zeit kostet.

- ***echo*** ist ***print*** vorzuziehen, da ***echo*** nichts zurückliefert. ***print*** dagegen gibt 0 oder 1 zurück.

- Man sollte keine Variablen kopieren. Siehe folgendes Beispiel:

  Schlechte Performance

  ```
  $last_name = $_POST['lastname'];
  processSomething($last_name);
  ```

  Bessere Performance

  ```
  processSomething($_POST['lastname']);
  ```

- ***Absolute Pfadangaben*** sind *relativen Pfadangaben* beim Laden von Dateien vorzuziehen. Der Performance-Unterschied kommt daher zustande, dass bei relativen Pfadangaben die relativen Pfade zuerst aufgelöst werden müssen, so dass

die Dateien gefunden werden können.

- Wenn möglich ***Getter- & Setter-Methoden in Klassen vermeiden***, auch wenn das nicht gerade objektorientiertes Programmieren unterstützt. Ein direkter Zugriff auf Klassen-Eigenschaften ist deutlich schneller.
  Beispiel:
  ```
  $car_obj = new car();
  $car_obj->name = 'BMW';
  ```

- Man sollte die so genannten ***Built-In-Funktionen*** von PHP verwenden und nicht dieselbe Funktionalität mit selbst geschriebenen Funktionen entwickeln. PHP bietet bereits eine große Anzahl an Built-In-Funktionen, die bereits viele Themen im alltäglichen Programmiergeschäft abdecken.

- ***Pass-By-Reference*** in Funktions-Parameter nutzen. Der Grund liegt darin, dass keine Variable zuerst kopiert werden muss, bevor die Funktion aufgerufen wird. Folgendes Beispiel zeigt die Unterschiede:

  Schlechte Performance
  ```
  function doSomething($text_str)
  ```

  Bessere Performance
  ```
  function doSomething(&text_str)
  ```

- Für die Zusammensetzung von Strings Kommas (**,**) anstatt Punkte (**.**) verwenden. Beispiel:

  Schlechte Performance
  ```
  echo ‘Mein ’ . ‘Name ’ .  ‘ist ’ . ‘Thomas ’ . ‘Burgard’;
  ```

  Bessere Performance
  ```
  echo ‘Mein ’ , ‘Name ’ , ‘ist ’ , ‘Thomas ’ , ‘Burgard’;
  ```

  Diese Maßnahme bringt nur eine minimale Performance-Verbesserung.

- Verwende ***SIngle-Quotes*** anstatt Double-Quotes.

  Schlechte Performance
  ```
  $count = 10;
  echo “I have $count nuts!”;
  ```

  Bessere Performance
  ```
  $count = 10;
  echo ‘I have’ . $count . ‘nuts!’;
  ```
  Bei Double-Quotes muss eine Substitution vorgenommen werden, um dann die Variable(n) mit dem Wert bzw. Werten zu ersetzen.

- Zum Anfügen von String-Inhalten verwende ***$str_tmp .= "some additional text";*** anstelle von **$str_tmp = $str_tmp . "some additional text";**

- Für den Zugriff auf Array-Elemente ***$array_tmp['key']*** anstelle von ***$array_tmp[key]*** verwenden.

- Eine lokale Variable erhöhen ist die schnellste Methode.

- Eine globale Variable erhöhen ist ca. zweimal langsamer als eine lokale Variable.

- Eine Objekt-Eigenschaft erhöhen (z.B. $this->var_tmp++) ist ca. dreimal langsamer als eine lokale Variable.

- Eine undefinierte lokale Variable ist ca. 10 mal langsamer als eine bereits initialisierte Variable.
- Verwende ***Ternäre Operatoren*** wie ***?:*** anstatt ***if***, ***elseif*** und ***else*** oder ***switch-case***. Folgendes einfaches Beispiel zeigt den Unterschied:

  Schlechte Performance
  ```
  if ($a > $b) {
     $str_tmp = 'yes';
  } else {
     $str_tmp = 'no';
  ```

  Bessere Performance
  ```
  ($a > $b) ? true : false;
  ```

  Der Tenäre-Operator ***?:-Operator*** ist um einiges schneller als die ***if***, ***elseif*** und ***else*** oder ***switch-case*** Bedingungen, da keine Anweisungen ausführt werden sondern nur Werte zurück gibt.

- Die Verwendung des ***@-Operators*** zur Fehlerunterdrückung ist zu vermeiden.

  Schlechte Performance
  ```
  for ($i = 0; $i < 10000; $i++) {
     echo @$var_tmp;
  }
  ```

  Bessere Performance
  ```
  for ($i = 0; $i < 10000; $i++) {
     if (isset($var_tmp)) { echo $var_tmp; }
  }
  ```

- Wenn möglich bei Abfragen den Identitätsoperator **===** verwenden anstelle **==**.

  Schlechte Performance
  ```
  if ($_POST['name'] == 'xzy') {routine()}
  ```

Bessere Performance

```
if ($_POST['name'] === ‘xzy’) {routine()}
```

Um den Dateityp bereits von Anfang an festzulegen, gibt es in PHP den Identitätsoperator **===**. Es wird nur dann das Ergebnis TRUE zurückgeliefert, wenn Wert und Typ übereinstimmen. Der Grund der besseren Performance liegt darin, dass hier keine interne Umwandlung stattfindet, wenn der Typ nicht übereinstimmt, also FALSE zurückliefert.
Verwendet man **==**, so findet immer eine Typumwandlung statt, die natürlich Zeit kostet.

Beispiel:
1 == ”1" liefert das Ergebnis TRUE. Das kostet jedoch Zeit, da PHP intern trotzdem die Dateitypen umwandeln muss.

- Für die Speicheroptimierung Variablen (*insbesondere Arrays*) mit ***unset*** wieder freigeben.

- Performance-Optimierung in ***for-Schleifen*** durch die Bestimmung des Schleifenzählers vor dem Schleifenstart:

Schlechte Performance

```
$array_items = array(1,2,3,4,5,6,7,8,9,10);

for ($idx = 0, $idx < count($array_items), $idx++) {
    // … do something!
}
```

Bessere Performance

```
$array_items = array(1,2,3,4,5,6,7,8,9,10);
$counter = count($array_items);
for ($idx = 0, $idx < $counter, $idx++) {
    // … do something!
}
```

Wird der Schleifenzähler innerhalb der ***for-Schleife*** bestimmt, also ***for ($idx = 0, $idx < count($array_items), $idx++) { … }***, wird bei jedem Schleifendurchlauf die Funktion “count($array_items)” aufgerufan, was natürlich zur Performance-Verschlechterung führt.

- Zugriff auf Array-Element mit der ***foreach***-Anweisung anstelle der ***for***- oder ***while***-Schleife.

Beispiel:

```
$array_items = array_fill(1, 2, 3, 4, 5, 10, 100, 1000);

foreach($array_items as $array_item) {
```

```
        $element = $array_item;
    }
```

- Prüfen, ob ein Wert in einem Array existiert mit ***isset()*** anstelle von ***in_array()***. Besonders bei großen Arrays macht sich die Performance-Verschlechterung bei ***in_array()*** bemerkbar.

  Schlechte Performance

```
$array_tmp = array("Windows", "Linux", "Mac", "Android", ...);
if (in_array("", $array_tmp)) {
    // Hier steht der Aktions-Code
}
```

  Bessere Performance

```
$array_tmp = array("Windows", "Linux", "Mac", "Android", ...);
if (isset($array_tmp['Windows'])) {
    // Hier steht der Aktions-Code
}
```

- Den Inhalt von Dateien mit ***file_get_contents()*** anstelle von ***fread()***. Die Funktion ***file_get_contents()*** verwendet "Memory Mapping" für die Performance-Steigerung. Gerade bei größeren Dateien zeigt ***file_get_contents()*** deutliche Performance-Steigerung.

- Einem Array Werte hinzufügen mit ***$array_tmp[] = 'new value';*** wenn es keine Rolle spielt, wo das Element hinzugefügt wird.
  Schlechtere Performance
  Fügt das Element an das Ende des Arrays
  `array_push($array_tmp, 'new value');`
  Fügt das Element an den Anfang des Arrays
  `array_unshift($array_tmp, 'new value');`

- Wenn möglich ***const*** anstelle von ***define()*** verwenden. ***define()*** sollte nur dann verwendet werden, wenn auf globale Einstellungsdaten zugegriffen wird. ***define()*** definiert während der Laufzeit eine benannte globale Konstante.

  const dagegen definiert eine Konstante innerhalb einer Klasse und wird zur Compilezeit definiert, was natürlich dann performanter während der Laufzeit ist.

  Schlechtere Performance

```
define('KONSTANTE', 10);
```

  Bessere Performance

```
const KONSTANTE = 10;
```

- Wenn möglich auf „Regular Expressions" verzichten. Besser ***str_replace()*** als ***preg_replace()***, da die String-Funktionen schneller sind.

- Mit der PHP-Funktion ***microtime()*** kann die Laufzeit eines Scriptes oder einer Aktion gemessen werden.

  ```
  $start_time = microtime();
   // Hier steht der Script-Code oder die Script-Aktion

  $laufzeit = microtime() - $start_time;
  echo "Laufzeit des Skripts: $laufzeit Sek.";
  ```

  Die Funktion ***microtime()*** wurde in PHP5 hinzugefügt. Zurückgegeben wird der Unix-Zeitstempel inkl. Microsekunden als float-Wert.

- ***Bit-Operatoren*** sind schneller als ***arithmetische Operatoren***.

  Schlechtere Performance
  ```
  $result = $value_tmp * 8;
  ```

  Bessere Performance
  ```
  $result = $value_tmp << 3; // Verschiebt um drei Bits nach links, was einer
                             //  Multiplikation mit 8 entspricht!
  ```

4 Caching-Verfahren

Definition des Begriffs “Cache” (Quelle Wikipedia: http://de.wikipedia.org/wiki/Caching)
Cache *bezeichnet in der EDV einen schnellen Puffer-Speicher, der (erneute) Zugriffe auf ein langsames Hintergrundmedium oder aufwändige Neuberechnungen zu vermeiden hilft. Inhalte/Daten, die bereits einmal beschafft/berechnet wurden, verbleiben im Cache, so dass sie bei späterem Bedarf schneller zur Verfügung stehen. Auch können Daten, die vermutlich bald benötigt werden, vorab vom Hintergrundmedium abgerufen und vorerst im Cache bereitgestellt werden.*

Caches können als Hardware- oder Softwarestruktur ausgebildet sein. In ihnen werden Kopien zwischengespeichert.

Die Vorteile von Caching bei PHP-Webseiten liegen auf der Hand:

- Ein erneutes Interpretieren der PHP-Skripte ist nicht notwendig.
- Entlastung des Webservers, da deutlich weniger Resourcen benötigt werden.
- Erneute Datenbankzugriffe sind nicht mehr notwendig. Somit brauchen keine Datenbankverbindungen neu aufgebaut werden.

Die aufgezählten Vorteile bringen insgesamt eine deutliche Performance-Steigerung bei Website-Zugriffen.

Wann sollte “Caching” verwendet werden?

Immer dann, wenn die Website hohe Zugriffsraten aufweist und Seiten, bei denen sich der Inhalt nicht zu oft ändert.

Wann sollte "Caching" nicht verwendet werden?
Hat man Seiten, bei denen sich der Inhalt oft ändert, sollte ein Caching vermieden werden.

4.1 Opcode Caching

Ein ***Opcode Cache*** speichert den kompilierten PHP-QuellCode (*den Opcode*) in einem "Shared Memory" des Webservers zwischen. Ein erneutes Kompilieren ist somit quasi nicht mehr notwendig.

PHP-Script ohne Opcode Caching:
Request → PHP-Script → Parser → Compiler → Opcode Ausführung → Response

PHP-Script mit Opcode Caching:
Request → PHP-Script → **Opcode Cache** → Opcode Ausführung → Response

Die Performance-Unterschiede von APC, XCache und eAccelerator sind nicht bedeutend unterschiedlich, so dass hier keine eindeutige Empfehlung gegeben werden kann.

4.1.1 APC-Cache

APC steht für **A**lternative **P**HP **C**ache und ist ein Quellcode offener (Open Source) ***Opcode Caching Framework***. Der ***APC*** stellt auch einen User-Cache zur Verfügung, mit dem auch einfache Werte und komplexe Objekte auf dem lokalen Webserver (*kein verteiltres Memory caching*) zwischengespeichert werden können. Implementiert ist der ***APC Cache*** als **Hashtabelle**, wodurch ein schneller Zugriff gewährleistet ist. ***APC*** ist derzeit noch kein Bestandteil von PHP (*kommt in PHP 6!*) und muss separat als **PECL**-Erweiterung installiert werden.

Der Vorteil von APC: Wird in PHP 6 ein Bestandteil von PHP sein.

APC in PHP verwenden
Beispiel für das Speichern einer Variable im Cache:

```
$apc_var = "This is a value for storing in APC Cache.";
apc_store("Key1", $apc_var);

// Aus dem Cache lesen
echo apc_fetch("Key1");
```

4.1.2 XCache

XCache ist ebenfalls wie sein Konkurrent ***APC*** ein Quellcode offener (Open-Source) ***Opcode Caching Framework***. Wie beim APC Cache können auch Daten in den Cache

geladen werden.

4.1.3 eAccelerator

Beschreibung siehe ***APC*** und ***Xcache***.

4.1.4 Memcache/MemcacheD

Memcache/MemcacheD ist ebenfalls Open-Source und ist ein so genanntes ***distributed memory object caching system***, auch als ***Cache-Server*** bezeichnet, für das Speichern von Programmteilen in den Arbeitsspeicher. Verwendet wird ***Memcache/MemcacheD*** bei größeren PHP-Projekten mit großer Datenbank (mehrere Millionen Einträge) und Websites mit sehr hohem Traffic. Bei kleinen PHP-Projekten bringt ***Memcache/MemcacheD*** nicht sonderlich viel.
Memcache/MemcacheD ist wie bereits beschrieben ein verteiltes Caching System, d.h. es können mehrere unabhängig voneinander betriebene Server für den Aufbau eines Caching-Systems herangezogen werden. Auf jedem Server wird ein so genannter ***Memcached-Daemon*** installiert und gestartet.
In der PHP-Anwendung muss eine extra Bibliothek für ***Memcache/MemcacheD*** verwendet werden, mit der mittels Schlüsselwörter dann die Informationen gespeichert werden.

Die Vorteile von ***Memcache/MemcacheD***:

- Verteiltes Caching-System, da mehrere Rechnersysteme verwendet werden können
- Sehr schnelles Caching-System
- Voll skalierbar. Wenn notwendig, können weitere Rechnersysteme für das Caching herangezogen werden
- Webserver wird schneller, da Datenbank-Daten in das Caching-System verlagert werden können. Wenn das Caching-System auf dem Webserver gestartet ist, können die Daten direkt aus dem Cache gelesen werden. Ein Datenbankzugriff auf einen eigenen Datenbank-Server ist somit nicht mehr notwendig. Das Resultat ist eine deutliche Verringerung des Datenverkehrs auf dem Webserver.

4.2 Opcode Optimizer

Im Gegensatz zu Opcode Caching wird beim ***Opcode Optimizer*** bereits während des Kompilierens eine Optimierung des Codes vorgenommen, so dass dann die Laufzeit des PHP-Scriptes schneller ist.

4.2.1 Zend Gaurd Loader für PHP ab der Version 5.3

Die kostenfreie Laufzeit-Applikation ***Zend Guard Loader*** ist ein PHP Opcode Optimizer und stellt eine PHP Engine Erweiterung dar. Voraussetzung ist, dass der PHP-Code mit dem kostenpflichtigem Programm “Zend Guard” (PHP-Encoder) verschlüsselt wurde.

Weiter Informationen unter: http://www.zend.com/de/products/guard/runtime-decoders

Für PHP bis zur Version 5.2 stellt Zend das ebenfalls kostenfreie Laufzeit-Applikation ***Zend Optimizer*** zur Verfügung!

5 PHP Performance-ToolsDesign-Pattern

5.1 Xdebug-Profiler

Eine gute Möglichkeit, die Performance eines PHP-Scriptes zu ermitteln, ist der Einsatz des Performance-Analyse-Tools ***XDEBUG-Profiler***.

In der **php.ini** folgende Parameter setzen:
- xdebug.profiler_enable = 0
- xdebug.profiler_append = 1
- xdebug.profiler_enable_trigger = 1
- xdebug.profiler_output_dir = "xxxxxxxxxxx"
- xdebug.profiler_output_name = yyyyyyyyyyyy

"xxxxxxxxxxx" durch Pfadangaben ersetzen (z.B. unter Windows: "C:/xampp/tmp")!
yyyyyyyyyyyy z.B. **cachegrind.out.%p**

Das PHP-Script kann dann mit
http://localhost/script.php?XDEBUG_PROFILE=1 aufgerufen werden.

Mit dem Zusatz **?XDEBUG_PROFILE=1** wird dann erst der Profiler aktiviert und nicht permanent für alle Scripts, wenn der Parameter in der **php.ini** aktiviert ist.

5.1.1 Grafische Auswertung der Profiledaten

- auf einem Linux-System mit ***KCachegrind***_
 http://kcachegrind.sourceforge.net/html/Home.html

- auf einem Windows-System mit ***WinCacheGrind***
 http://sourceforge.net/projects/wincachegrind/

- als Web-Frontend mit ***Webgrind***
 http://code.google.com/p/webgrind/

- und mit ***xdebugtoolkit***
 http://code.google.com/p/xdebugtoolkit/

Weitere Informationen zu Xdebug-Profiler unter: http://xdebug.org/docs/profiler

6 MySQL - Einige Performance-Optimierungen

Diese Optimierungsmaßnahmen sind mit Sicherheit sehr komplex und ***müssen prinzipiell äußerst umsichtig und bedacht vorgenommen werden***. Wählt man als Beispiel eine Storage-Engine, die besonders schnell ist, ist das für die Performance gut, jedoch für bestimmte Aufgaben sehr schlecht, da Funktionen nicht vorhanden sind.

Was bedeutet die Performance bei einer MySQL-Datenbank?

- ***Latenz*** => schnelle Antwortzeit
- ***Durchsatz*** => Viele Antworten pro Zeit
- ***Skalierbarkeit*** => Beibehaltung der Latenz bei Erhöhung der Parallelität)

Es soll in diesem Kapitel nur auf einige Performance-Optimierungs-Maßnahmen eingegangen werden, da aus Komplexitätsgründen der Rahmen dieses Buches gesprengt werden würde.

6.2 Design

Das Wichtigste überhaupt ist ein gut durchdachtes Datenbank-Design. Man sollte alle Normalisierungsregeln beachten, aber bitte ***mit den Normalformen nicht übertreiben***, denn manchmal sind kleine Redundanzen bei der Abfrage performanter. Der Grund liegt darin, dass MySQL bei einer Abfrage die Daten von verschiedenen Tabellen zusammenfügen muss. Sind alle Redundanzen aufgelöst, existieren natürlich auch mehr Tabellen, somit wird eine Abfrage langsamer.

- Verwendung von ***Indizes***
  Die Verwendung von Indizes erhöhen die Abfrage-Performance. Trotzdem sollten Indizes nur bei Spalten verwendet werden, bei denen häufiger gesucht bzw. sortiert wird.
- Verwendung des Unicode-Zeichensatzes ***utf8_general_ci***
  In der MySQL-Dokumentation steht dazu:
  utf8_general_ci ist eine ältere Sortierfolge, die Erweiterungen nicht unterstützt. Hier können nur 1:1-Vergleiche zwischen Zeichen durchgeführt werden: **Vergleiche für die utf8_general_ci-Sortierfolge sind also schneller**, aber unter Umständen weniger korrekt als Vergleiche für utf8_unicode_ci.
- Verwendung der ***richtigen Datentypen beim Anlegen von Tabellen***.
  Den Datentyp ***so kurz wählen wie nötig***.
- Wenn möglich, ***große Bild- und Videodaten im Dateisystem*** und nicht in der Datenbank ***speichern***.
  Bei kleinen Bilddateien kann auch in der Datenbank gespeichert werden, aber man muss dies gut überlegen und auch von der jeweiligen Funktion abhängig machen.

6.3 Datenbankzugriff (Abfragen)

- Bei großen Tabellen unbedingt ***SELECT ** FROM vermeiden. Stattdessen ***Paging*** verwenden oder mit ***limit*** arbeiten.
- ***Kleine Ergebnismengen des SELECT-Befehls bevorzugen***, da hier der PHP-Interpreter schneller auswerten kann.
- Unbedingt ***SQL-Abfragen in einer PHP-Schleife vermeiden***.

6.4 MySQL Storage Engines

Auch bei der Auswahl der Storage-Engine ist Vorsicht geboten. Jede Sorage-Engine hat seine Vorteile, aber auch Nachteile. Man sollte vorher genau alle Features studieren und entsprechend dem Anwendungsfall die Storage-Engine auswählen.

- ***MyISAM*** Storage-Engine immer dann, wenn überwiegend aus der Datenbank gelesen wird.
- Auf ***InnoDB***-Tabellen kann parallel (mittels Transaktionen) zugegriffen werden, da diese ACID- (**A**tomicity, **C**onsistency, **I**solation und **D**urability) konform sind. Somit hat man gegenüber MyISAM-Tabellen einen Performance-Gewinn (InnoDB-Tabellen können aber nicht immer eingesetzt werden, da entscheidet der spezielle Anwendungsfall).

  InnoDB Storage-Engine immer dann, wenn viel und parallel in die Datenbank geschrieben wird.

- Bei der Verwaltung von kleineren Datenmangen kann eine ***HEAP***-Tabelle verwendet werden, die nach dem Beenden der Datenbankverbindung wieder automatisch geschlossen wird.
  Bei einer HEAP-Tabelle wird eine Tabelle im Hauptspeicher und nicht auf der Festplatte erzeugt, was natürlich die Performance steigert.
  Der Nachteil von HEAP-Tabellen ist ein abgespeckter Funktionsumfang als bei MyISAM- oder InnoDB-Tabellen. Zusätzlich geht die HEAP-Tabelle verloren, wenn der MySQL-Server beendet wird.

6.5 MySQL Konfiguration in der Datei my.cnf

- ***Query Cache*** aktivieren.
  Die Aktivierung erfolgt im Abschnitt [mysqld].

  Beispiel:
  # Max. Größe Abfrage-Ergebnis, damit gecached wird
  query_cache_limit = 2M
  # Gesamtgröße Cache
  query_cache_size = 32M

- Die ***maximale Paketgröße*** für den Client-Server-Datenaustausch ***max_allowed_packet*** muss mindestens so groß wie das größte BLOB in einer Tabelle.
- ***Erhöhung des Speichers für Indexblöcke*** mit dem Parameter ***key_buffer_size***. Der Defaultwert ist 8M.
- Die ***Anzahl der geöffneten Tabellen*** mit dem Parameter ***table_cache*** erhöhen, denn das Öffnen und wieder Schließen von Tabellendateien kostet Performance. Der Defaultwert ist 64.
- ***Sortier-Puffe***r mit dem Parameter ***sort_buffer*** erhöhen (bei SELECT-Befehl mit ORDER bzw. GROUP BY, wenn kein Index vorhanden ist). Ist der Puffer zu klein gewählt, wird eine temporäre Datei herangezogen . Der Defaultwert ist 2 MB.
- Mit ***max_connections*** die Anzahl der ***Datenbank-Verbindungen erhöhen***. Der Defaultwert ist 100. Vorsicht bei der Wahl des Wertes ist geboten. Für jede Verbindung wird Speicherplatz und einen Datei-Deskriptor benötigt.
- Mit dem Parameter ***tmp_table_size*** kann der ***Speicherplatz für HEAP-Tabellen*** erhöht werden. Der Defaultwert ist 32M.
  Hier gilt:
  Wird der Wert für tmp_table_size überschritten, werden die HEAP-Tabellen in MyISAM-Tabellen konvertiert und als temporäre Datei gespeichert.
- Mit ***thread_cache_size*** kann der Cache-Speicher für nicht verwendete Verbindungs-Threads erhöht werden. Für jede Datenbankverbindung wird normalerweise ein eigener Thread erzeugt. Nicht aktive Threads werden in den Thread-Cache gespeichrt, so dass MySQL für eine neue Verbindung zuerst im Thread-Cache nachschaut, bevor ein neuer Thread erzeugt wird. Wird der Cache erhöht, braucht erst mal kein neuer Thread erzeugt werden, was natürlich die Performance steigert.

7 Design-Patterns

Design Patterns sollten prinzipiell eingesetzt werden ***um Code-Redundanzen zu vermeiden***. Das alleine ist schon guter Performance-Gewinn. Die folgenden Design Patterns verbessern die Performance bzw. verringern den Speicherbedarf.

- ***Singleton***-Pattern
  In der Applikation existiert nur eine Instanz für eine Klasse.
- ***Factory***-Pattern
  Stellt eine Schnittstelle für die Erzeugung von Objekten zur Verfügung. Unterklassen instanziieren dann die Objekte und entscheiden über die konkrete Implementierung.
- ***Flyweight***-Pattern
  Mit diesem Design Pattern kann die Anzahl der Objektinstanzen geringer gehalten werden.
- ***Proxy***-Pattern
  Eine Stellvertreter-Klasse steuert den Zugriff auf ein Objekt. es wird nicht nur die Sicherheit für ein Objekt erhöht, sondern auch Mehrfach-Implementierung wird verhindert, was wiederum den Speicherplatz optimiert und somit die Performance steigert.

8 Anhang

8.1 Links zum Thema PHP und Performance

Apache
http://httpd.apache.org/docs/2.0/programs/ab.html

PHP
http://www.code.google.com/speed
http://www.php-performance.de/
https://developers.google.com/speed/articles/optimizing-php
http://de.wikipedia.org/wiki/PHP

MySQL
http://dev.mysql.com/doc/refman/5.6/en/
http://dev.mysql.com/usingmysql/php/
http://de.wikipedia.org/wiki/MySQL