FOM Hochschule für Oekonomie & Management Essen
Standort München
Berufsbegleitender Studiengang zum B.Sc. Wirtschaftsinformatik

Seminararbeit

Optimierung von MySQL Anfragen unter Zuhilfenahme von Explain

Eingereicht von:

Oliver Kurmis

Matrikel-Nr: 328091

Betreuer: Dipl.-Wirtschaftsinf. (FH) Klaus Arto

Abgegeben am:

1. Juli 2014

Erarbeitet im:

3. Semester

Inhaltsverzeichnis

Abkürzungsverzeichnis **II**

1 Einleitung **1**

2 Theoretische Grundlagen **2**

2.1 Der physische Zugriff auf die Daten 2

2.2 Speicherstrukturen 2

2.2.1 Binärbaum 3

2.2.2 B-Baum 4

2.2.3 Hashing 4

2.2.4 Heap 5

2.3 Bearbeitung von SQL-Statements 5

2.4 einfache Optimierungen 5

3 MySQL-EXPLAIN **5**

3.1 Umschreiben von Nicht-SELECT-Anfragen 6

3.2 Einfache SELECT-Anfragen mit einer Tabelle 6

3.3 Die Spalten der EXPLAIN-Ausgabe 7

3.3.1 EXPLAIN EXTENDED 10

3.3.2 EXPLAIN PARTITIONS 10

3.4 Abfragen mit mehreren Tabellen 10

3.5 Optimierungsmöglichkeiten und Benchmarking 10

3.6 Visuelles EXPLAIN (graphische Werkzeuge) 11

4 Fazit und Ausblick **11**

5 Anhang **12**

Literatur **13**

Abkürzungsverzeichnis

CPU Central Processing Unit (deutsch: Hauptprozessor)

DB Datenbank

HDD Hard Disk Drive (deutsch: Festplattenlaufwerk)

QEP Query Execution Plan (deutsch: Anfrage-Ausführungsplan)

RAM Random Access Memory (deutsch: Hauptspeicher oder Arbeitsspeicher)

RDBMS Relationales Datenbank-Managementsystem

SQL Structured Query Language, standardisierte Datenbank-Abfragesprache

SSD Solid State Drive (deutsch: Halbleiterlaufwerk, i.d.R. mit Flash-Speicher)

1 Einleitung

Datenbank-Systeme finden heute in nahezu allen IT-Systemen Verwendung. Der Optimierung von Datenbank-Anfragen kommt daher eine große Bedeutung zu. Hierfür gibt es eine Vielzahl von Möglichkeiten, z.B. Latenz und Bandbreite der Anbindung der Datenbank, Leistungsfähigkeit des Datenbank-Servers, Anzahl der Datenbank-Anfragen in der Anwendung und diverse Caching-Meachanismen. Hat man andere Flaschenhälse ausgeschlossen oder bereits optimiert, gilt es die für die Performance relevanten SQL-Abfragen des Systems zu identifizieren und gezielt zu optimieren. Dies können lange laufende Abfragen sein, die z.B. bei MySQL mit der Logdatei für langsame Anfragen gezielt ermittelt werden können. Oftmals sind es aber auch viele einfache, kurze Abfragen, die jedoch zu Hunderten oder Tausenden pro Sekunde auftreten und so die Anwendung viel Zeit kosten und den Datenbank-Server belasten. Viele RDBMS stellen mit dem SQL-Kommando EXPLAIN eine Möglichkeit zur Verfügung, mehr über die innere Arbeitsweise der Datenbank bei einer bestimmten SQL-Abfrage zu erfahren. Durch gezielte Veränderung der SQL-Abfrage oder des Datenschemas kann somit die Bearbeitung der Abfrage optimiert werden.

Die vorliegende Arbeit bezieht sich speziell auf die Optimierung von SQL-Anfragen mittels des SQL-Kommandos EXPLAIN bei dem RDBMS MySQL. Es soll aufgezeigt werden, wie mit dem Query Execution Plan der EXPLAIN-Ausgabe die Arbeitsweise der Datenbank besser verstanden werden kann. Darauf aufbauend wird gezeigt, wie mit zielgerichteten Änderungen des Datenbank-Schemas und der SQL-Anfragen die Geschwindigkeit der Datenbank-Anfragen erheblich gesteigert werden kann.

Grundlage dieser Seminararbeit sind die Fachbücher [1], [2], [3] sowie die MySQL-Referenz-Dokumentation [4], [5], [6]. Nach der Literaturrecherche und Bearbeitung der relevanten Fachliteratur bestand mein Betrag darin, die wesentlichen Punkte kompakt zusammenzufassen und die Vorgehensweise der Optimierung an einigen Beispielen experimentell zu demonstrieren.

2 Theoretische Grundlagen

2.1 Der physische Zugriff auf die Daten

Daten einer DB werden in der Regel auf einer Festplatte (HDD) oder einem Flash-Laufwerk (SSD) gespeichert. Das RDBMS nutzt dazu Funktionen des Betriebssystems auf verschiedenen Ebenen. Dateisystem-Treiber, nimmt Lese und Schreibanforderungen für Datensätze an und gibt rechnet diese in nie durchnumerierten Blöcke des Blockgerätes um. Der Blockgeräte-Treiber liest dann die entsprenden Blöcke von der Platte oder schreibt sie dort hin.

DBMS <—> Dateisystemtreiber <—> Blockgerätetreiber <—> HDD/SSD

Auf den verschiedenen Ebenen findet hierbei Caching statt, um die relativ langsamen Zugriffe auf den Massenspeicher (HDD oder SSD) zu vermeiden oder zumindest zu bündeln. Die Reduzierung von Massenspeicher-Zugriffen ist daher auch eine effektive Methode der DB-Anfrage-Optimierung.

2.2 Speicherstrukturen

Um die Daten einer Datenbank persistent zu speichern, werden diese auf einem Massenspeicher (HDD oder SSD) abgelegt. Auf dieses sog. Blockgerät kann immer nur in Datenblöcken fester Grösse, auch Pages oder Seiten genannt, zugegriffen werden. MySQL speichert jede Tabelle in einer eigenen Datei, für einen Zugriff auf einen bestimmten Datensatz muss daher dessen Position innerhalb der Datei bekannt sein (Record-ID, RID). Andere Datenbanken wie SAP MaxDB oder Oracle speichern alle Tabellen in einer Datei (sog. Tablespace), die vorher mit einer festen Größe angelegt werden muß, um den Speicherplatz für die Daten zu reservieren (vgl. [2], S. 145f). Es ist auch möglich, eine ganze Partition oder ein ganzes physisches Laufwerk für den Tablespace zu verwenden, womit der Overhead des Dateisystems vermieden wird und schneller auf die Datensätze zugegriffen werden kann.

Ein Grundproblem bei Datenbanken ist es, einen bestimmten Datensatz schnell aufzufinden, denn bei grossen Tabellen ist es nicht praktikabel, dafür die gesamte Tabelle zu durchsuchen. Deswegen wurden verschiedene Algorithmen und Speicherstrukturen entwickelt, die den Zugriff auf einen Datensatz mit möglichst wenig Datenträ-

gerzugriffen ermöglichen. Heutige Festplattenlaufwerke erlauben bei Zugriffszeiten von 10ms ca. 100 zufällige Zugriffe pro Sekunden, viele Grössenordnungen langsamer als Zugriffe im RAM. Mit dem Aufkommen erschwinglicher SSDs mit einigen 10.000 bis zu einigen 100.000 Zugriffen pro Sekunde ist das Problem etwas entschärft aber immer noch vorhanden. Im folgenden werden einige Speicherstrukturen vorgestellt, die jedoch nicht alle bei einem RDBMS implementiert sein müssen.

2.2.1 Binärbaum

Der Binärbaum ist einer baumartige Zeigerstruktur, bei der jeder Knoten die Daten selbst (oder einen Zeiger auf die Daten) und zwei Zeiger auf weitere Knoten oder Blätter enthält. Ein Zeiger zeigt auf einen kleinere Datensatz und ein Zeiger zeigt auf einen größeren Datensatz. Ein Knoten hat also prinzipiell folgenden Aufbau:

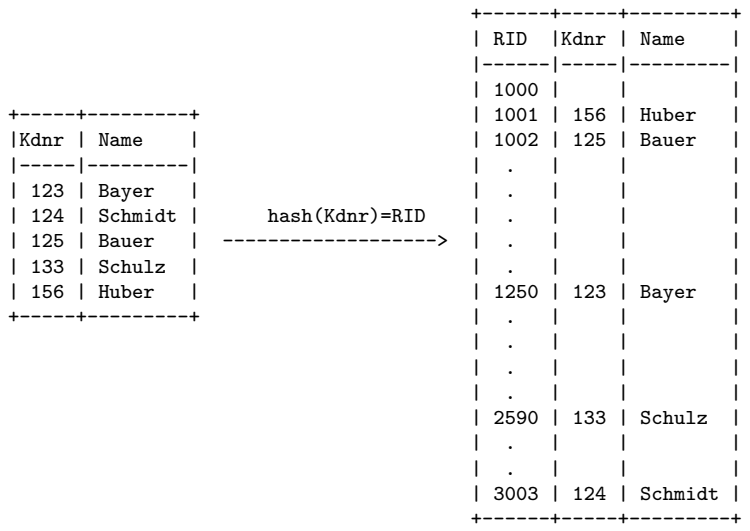
```
           RID:    14
Schlüssel-Wert: München
   kleiner-RID:    34
    größer-RID:    55
```

Wird ein neuer Datensatz eingefügt, so wird dessen Schlüssel-Wert verglichen mit dem Schlüssel-Wert des Wurzel-Knotens und dann rekursiv auf der kleineren oder auf der grösseren Seite des Baumes einsortiert. So ensteht eine geordnete Struktur, welche durch Rekursion in sortierter Reihenfolge gelesen werden kann. Bei häufigen Schreibzugriffen kann es passieren, dass der Binärbaum entartet. Beispielsweise würden beim Einfügen von immer grösseren Werten die rechte Seite zu einer verketteten Liste entarten. Daher muss der Baum regelmässig neu generiert (ausbalanciert) werden, was sehr aufwendig ist. Ein anderer Nachteil ist, daß bei 1000 Datensätzen, bereits 10 Ebenen, also 10 Datenträger-Zugriffe nötig sein, bei 1 Mio. Datensätzen, sind es bereits 20 Datenträgerzugriffe, also ein Zeitaufwand von etwa 0,2 Sekunden zum Auffinden eines Datensatzes (vgl. [2], S. 147f).

2.2.2 B-Baum

2.2.3 Hashing

Eine Methode um schnell auf einenDatensatz zuzugreifen ist das Hash-Verfahren. Hierbei wird der Schlüssel des Datensatzes mit einer Hash-Funktion auf eine RID abgebildet. Eine Hash-Funktion ist jedoch keine eieindeutige Abbildung, daher kann es auch zu Kollisionen kommen, mehrere Datensätze können also auf die gleiche RID abgebildet werden. In einem solchen Fall müssen dann weitere Mechanismen dafür sorgen, dass die Daten an einer anderen Stelle gespeichert und wieder aufgefunden werden können. Z.B. kann die durch nochmalige Anwendung der Hash-Funktion erfolgen oder durch einen zusätzlichen Überlaufbereich, der die Daten einfach sequentiell speichert. Damit es möglichst selten zu Kollisionen kommt und der Zugriff dadurch verlangsamt wird, muss der für die Daten reservierte Speicher gross genug gewählt werden(vgl. [2], S. 153f).



Der Vorteil des Hashing besteht darin, dass die Zugriffszeit nahezu unabhänging von der Tabellengrösse ist (jedoch abhängig vom Füllgrad). In der Regel kann mit einem oder zwei Datenträgerzugriffen der Datensatz gefunden werden. Der grösste Nachteil ist die fehlender Sortierung der Daten. Bei einem Zugriff auf einen Bereich (WHERE Name LIKE 'B

2.2.4 Heap

Bei einem Heap (deutsch: Haufen) werden alle Datensätze einfach der Reihe nach auf dem Datenträger abgelegt. Um auf einen bestimmten Datensatz zuzugreifen, muss daher immer der gesamte Heap gelesen werden. Dies ist bei grossen Tabellen in der Regel ungünstig, kann jedoch eine Möglichkeit für kleinere Tabellen sein, die einen Umfang von einigen wenigen Speicherseiten haben, da diese komplett in den Datencache passen oder ggf. mit nur einem Datenträgerzugriff gelesen werden können.

2.3 Bearbeitung von SQL-Statements

Umsetzung in relationale Algebra

2.4 einfache Optimierungen

frühzeitige Restriktionen, JOINs

3 MySQL-EXPLAIN

In vielen RDBMS steht mit dem SQL-Kommando EXPLAIN ein Werkzeug zur Verfügung, um mehr darüber zu erfahren, wie die Datenbank eine bestimmte Anfrage ausführt, wie also der Query Execution Plan (QEP) ist. Das EXPLAIN-Kommando gehört jedoch nicht zum SQL-Standard und wird bei den verschiedenen RDBMS unterschiedliche Ausgaben erzeugen. Bei MySQL ist EXPLAIN sehr mächtig und gibt umfassend und datailliert Auskunft über den QEP. Hierbei muss jedoch beachtet werden, dass der QEP nicht fix ist, sondern bei jeder Anfrage vom Optimierer erneut erstellt wird (sofern die Anfrage nicht bereits aus dem Query-Cache bedient werden kann). Es gibt daher keine Garantie, dass die Anfrage immer mit dem vorher von EXPLAIN gezeigten QEP ausgeführt wird. Es empfielt sich daher, die untersuchten SQL-Anfragen von Zeit zu Zeit erneut mit EXPLAIN zu prüfen - mit Real-World-Daten.

3.1 Umschreiben von Nicht-SELECT-Anfragen

Vor MySQL 5.6.3 konnte EXPLAIN nur auf SELECT-Anfragen angewendet werden. [4] Einige Nicht-SELECT-Anfragen, wie DELETE, INSERT, REPLACE und UPDATE können jedoch in ein entsprechendes SELECT-Kommando umgeformt werden, um dieses dann mit EXPLAIN zu untersuchen. Zu beachten ist jedoch, daß schreibende Anweisungen generell aufwendiger sind als das entsprechende SELECT-Kommando, da zusätzlich zum Auffinden der Daten noch die Schreiboperation ausgeführt werden muss, ggf. kommen auch noch Aktualisierungen von Indizes hinzu. Beispiel:

```
DELETE user WHERE last_login < '2012-01-01'
```

kann zu folgendem SELECT umgeschrieben werden:

```
SELECT id FROM user WHERE last_login < '2012-01-01'
```

Ab MySQL 5.6.3 kann EXPLAIN auf SELECT, DELETE, INSERT, REPLACE und UPDATE angewendet werden.

3.2 Einfache SELECT-Anfragen mit einer Tabelle

An einem Beispiel einer einfachen SELECT-Anfrage soll gezeigt werden, wie die Ausgabe einer EXPLAIN-Anweisung aufgebaut ist. Hierzu wird zur Demonstation zuerst eine Tabelle mit einigen Daten erzeugt:

```
mysql> CREATE TABLE words (
       id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
       word VARCHAR(60) ) ENGINE=InnoDB;
Query OK, 0 rows affected (0,81 sec)

-- Worte aus Wörterbuch in die Tabelle laden
mysql> LOAD DATA LOCAL INFILE '/usr/share/dict/ngerman'
       INTO TABLE words (word);
Query OK, 339099 rows affected (6,58 sec)
Records: 339099  Deleted: 0  Skipped: 0  Warnings: 0
```

Es wurden also 339099 Zeilen bzw. Wörter in die Tabelle importiert, die id-Spalte hat MySQL dabei automatisch gefüllt und hochgezählt. Nun sollen beispielhaft die fünf alphabetisch letzten Worte abgefragt werden:

```
mysql> SELECT word FROM words ORDER BY word DESC LIMIT 5;
+-----------+
| word      |
+-----------+
| zzgl      |
| Zysten    |
| Zyste     |
| Zypressen |
| Zypresse  |
+-----------+
5 rows in set (0,20 sec)
```

Durch Voranstellen von EXPLAIN lässt sich der QEP der SELECT-Anfrage anzeigen:

```
mysql> EXPLAIN SELECT word FROM words ORDER BY word DESC LIMIT 5;
+----+-------------+-------+------+---------------+------+---------+------+--------+----------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows   | Extra          |
+----+-------------+-------+------+---------------+------+---------+------+--------+----------------+
|  1 | SIMPLE      | words | ALL  | NULL          | NULL | NULL    | NULL | 341202 | Using filesort |
+----+-------------+-------+------+---------------+------+---------+------+--------+----------------+
1 row in set (0,00 sec)
```

Wie man erkennt, wurde das EXPLAIN-Kommando schneller ausgeführt als das eigentliche SELECT. Dies liegt daran, daß EXPLAIN nicht tatsächlich auf die Daten in der Tabelle zugreift, sondern nur aufzeigt, wie die Datenbank die Datensätze auffinden würde.

3.3 Die Spalten der EXPLAIN-Ausgabe

Als Ergebnis einer EXPLAIN-Anfrage liefert MySQL eine Tabelle mit festen Spalten und einer oder mehrerer Zeilen, je nach Komplexität der Anfrage. Die einzelnen Spalten haben folgende Bedeutung (vgl. [5], [3] S. 665-676, [1] Pos. 2696-2906):

id
Diese Zahl identifiziert das SELECT, zu dem die Zeile gehört. Bei einer einfachen SELECT-Abfrage steht in diesem Feld demnach immer nur die Zahl 1.

select_type
Die Spalte gibt an, ob es sich um ein einfaches oder komplexes SELECT handelt. Folgende Werte können hierbei auftreten:

SIMPLE einfaches SELECT, keine Unterabfragen oder UNIONS
PRIMARY äußeres SELECT eines komplexen SELECT
SUBQUERY SELECT in einer Unterabfrage
DERIVED SELECT in einer Unterabfrage in der FROM-Klausel
UNION zweites bzw. nachfolgende SELECT einer UNION
UNION RESULT Ergebnis des UNION, wird aus temporärer Tabelle geholt

table
Die Spalte Table gibt an, auf welche Tabelle zugegriffen wird. Dies kann der tatsächliche Tabellenname sein oder der Alias. Die Spalte ist von oben nach unten zu lesen, um die Join-Reihenfolge zu sehen, die der Optimierer für die Anfrage gewählt hat. Bei komlexeren Anfragen mit abgeleiteten Tabellen und Vereinigungen können hier noch weitere Werte auftreten: <derivedN> wenn es in der FROM-Klausel eine Unterabfrage gibt, wobei N die ID der Unterabfrage ist und in den nachfolgenden Zeilen der Ausgabe zu finden ist. Bei einer Vereinigung mit UNION enthält die UNION RESULT-Zeile in der table-Spalte die IDs der Abfragen, welche vereinigt werden, und beziehen sich daher immer auf vorhergehende Zeilen der Ausgabe, z.B. <union2,4>.

type
Wie ist der Zugriffstyp, wie wird MySQL die Zeilen in der Tabelle auffinden? Folgende Werte können hierbei auftreten (in geordneter Reihenfolge vom langsamsten zum schnellsten Zugriffstyp):

ALL full Tablescan, d.h. Tabelle muss in der Regel von Anfang bis Ende durchlaufen werden, ist ein starker Indiz für weiteren Optimierungsbedarf
index wie Tablescan, aber Scannen der Tabelle erfolgt in Indexreihenfolge, eine extra Sortierung wird hierbei vermieden
range Bereichsscan, d.h. eingeschränkter Indexscan, z.B. bei BETWEEN oder > in der WHERE-Klausel
ref Indexzugriff findet statt, auch Index-Lookup genannt, Zeilen entsprechen einem Wert, nur bei einem nichteindeutigen Index, Index wird hierbei mit einem Referenzwert verglichen. Variante: ref_or_null
eq_ref Index-Lookup mit eindeutigem Treffer, bei Primärschlüssel oder eindeutigem Index

const,system der Datenzugriff konnte von MySQL wegoptimiert oder in eine Konstante umgewandelt werden.

NULL Abfrage kann von MySQL bei der Optimierung aufgelöst werden, kein Zugriff auf Tabelle oder Index, z.B. Minimum einer indizierten Spalte

possible_keys

Diese Spalte gibt an, welche Indizes für die Bearbeitung der Anfrage prinzipiell zur Verfügung stehen. Die hier stehenden Werte werden bereits in einer frühen Phase der Optimierung ermittelt, letzendlich wird in der Regel nur ein Index genutzt. Sind hier viele Indizes aufgeführt deutet das auf ein Problem hin.

key

Die Spalte key gibt an, welcher Index der Optimierer für die Anfrage gewählt hat. Dies kann auch ein abdeckender Index sein, aus dem die Ergebniswerte gelesen werden können ohne dass die eigentliche Tabelle gelesen werden muss. Ein Wert von NULL in der Spalte key bedeutet, dass kein Index genutzt wird und ist ein starkes Indiz für einen Optimierungsbedarf.

key_len

wie viel Byte (Spaltenbreite) eines Index werden benutzt welche Spalten des Index werden genutzt, von links beginnend

ref

-welche Spalten aus früheren Tabellen werden benutzt, um in dem key-Index nachzuschlagen

rows

Die Zahl in der Spalte rows ist eine Schätzung für die Anzahl der Zeilen, die gelesen werden müssen. Bei Abfragen mit mehreren Tabellen bezieht sich diese Angabe pro Schleife im Nested-Loop-Join-Plan. Die Schätzung beruht auf Statistiken kann ungenau sein. Im besten Fall steht hier eine 1.

filtered

Die Spalte ist neu seit MySQL 5.1 und erescheint nur bei EXPLAIN EXTENDED (s.u.). Es ist eine pessimistische Schätzung des Prozentsatzes der Zeilen, die eine Bedingung erfüllen, wie z.B. eine WHERE-Klausel oder ein JOIN mit einer anderen Tabelle.

Extra

Die Extra-Spalte enthält weitere Angaben, die nicht in die anderen Spalten passen:

Using Index abdeckender Index wird genutzt, d.h. die angefragten Daten müssen nicht aus der Tabelle gelesen werden

Using where die Zeilen werden nachträglich gefiltert, d.h. für die WHERE-Bedingung wird nicht der Index genutzt

Using temporary Erstellung einer temporäre Tabelle für Sortierung

Using filesort externe Sortierung, im RAM oder auf den Datenträger

ranke checked for each record (index map: N) kein geeigneter Index vorhanden, N ist ein Bitmap auf die Spalten in possible_keys

3.3.1 EXPLAIN EXTENDED

Wird EXPLAIN EXTENDED anstatt von EXPLAIN verwendet, dann erscheint die zusätzliche Spalte **filtered**. Außerdem werden weitere Informationen generiert, die mit dem nachfolgenden SQL-Kommando SHOW WARNINGS angezeigt werden können. Eine ausführliche Beschreibung findet sich unter [6].

3.3.2 EXPLAIN PARTITIONS

Wird EXPLAIN PARTITIONS verwendet, dann zeigt zusätzlichen Spalte **partitions** die Partitionen, auf welche die Anfrage zugreift, sofern welche verfügbar sind. Diese Option ist erst seit MySQL 5.1 verfügbar. Das Schlüsselwort EXTENDED kann nicht zusammen mit EXPLAIN PARTITIONS verwendet werden.

3.4 Abfragen mit mehreren Tabellen

3.5 Optimierungsmöglichkeiten und Benchmarking

Wichtigsten Spalten key (benutzer Index), rows (Anzahl Zeilen bearbeitet), type

QEP ist nicht fix, wird bei jeder Abfrage neu erstellt. Daher EXPLAIN einer Abfrage mit verschiedenen Beispielwerten. Nicht mit Test-Daten testen, sondern RealWord-Daten, z.B. Backups.

ALTER TABLE users ADD INDEX

Vorsicht beim Anlegen von INDIZES: Kann bei grossen Tabellen sehr lange Dauern und blockiert in dieser Zeit die Tabelle. Vorher Informationen über die Grösse der Tabellen holen, am besten einen Probedurchlauf auf einer Kopie machen.

Wird ein vorhandener Index bei der Anfrage nicht genutzt, obwohl er augenscheinlich genutzt werden sollte, dann kann das an veralteten Tabellen-Statistiken oder einer ungenügenden Stichprobe für die Statistiken liegen. Die Statistiken für eine Tabelle lassen sich dann mit ANALYZE TABLE Tabellenname aktualisieren.

3.6 Visuelles EXPLAIN (graphische Werkzeuge)

4 Fazit und Ausblick

Beschränkungen! Optimierung wichtig Mit Explain möglich nicht immer exakte Angaben Kontrolle der Optimierung mit Benchmarks nötig möglichst bereits in den Entwicklungsprozess integrieren, und nicht erst wenn es brennt

5 Anhang

Beispieldaten und -Scripte
Beispieldaten und Scripte können auf GitHub heruntergeladen werden (ca. 18 MB):
`https://github.com/oliworx/MySQL-EXPLAIN/archive/master.zip`

Setup des Testsystems
Sämtliche Tests wurden auf einem Notebook mit folgender Konfiguration durchgeführt:

Model Lenovo Thinkpad Edge 15 0319-A18
CPU Intel® Pentium® P6200 Prozessor, 2x 2,13 GHz , 3 MB Cache
RAM 4 GB, DDR3 SDRAM , PC3 8500 (1066 MHz)
HDD 320 GB, 2,5“, 7200rpm
OS Ubuntu 14.04 LTS, 64 Bit
DB MySQL 5.6.19,query_cache_type=OFF, Workbench 6.0.8.11354,

Literatur

[1] Bradford, R. (2011) *Effective MySQL: Optimizing SQL Statements*, Oracle Press/McGraw-Hill Osborne Media, 2011

[2] Sauer, H. (1998) *Relationale Datenbanken, Theorie und Praxis*, 4. Auflage, Addison Wesley Longman Verlag, 1998

[3] Schwartz, B., Zaitsev, P., Tkachenko, V., Zawodny, J.D.,Lentz, A., Balling, D.J. (2009) *High Performance MySQL. Optimierung, Datensicherung, Replikation & Lastverteilung*, 2. Auflage, O'Reilly Verlag, 2009

Internetquellen:

[4] ORACLE MySQL Documentation (2014): *Optimizing Queries with EXPLAIN*. URL: `http://dev.mysql.com/doc/refman/5.6/en/using-explain.html`, Abruf am 28.6.2014

[5] ORACLE MySQL Documentation (2014): *EXPLAIN Output Format*. URL: `http://dev.mysql.com/doc/refman/5.6/en/explain-output.html`, Abruf am 28.6.2014

[6] ORACLE MySQL Documentation (2014): *EXPLAIN EXTENDED Output Format*. URL: `http://dev.mysql.com/doc/refman/5.6/en/explain-extended.html`, Abruf am 28.6.2014

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat. Ich erkläre mich damit einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hoch geladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

München, 30. Juni 2014

Oliver Kurmis