KAPITEL 4

Indizes

Indizes erlauben es MySQL, aus Millionen oder sogar Milliarden von Datensätzen in einer Tabelle sehr schnell eine Gruppe von Records zu finden und abzurufen. Wenn Sie MySQL eine gewisse Zeit genutzt haben, werden Sie wahrscheinlich auch schon Indizes angelegt haben in der Hoffnung, blitzschnelle Antworten auf Ihre Queries zu erhalten. Und vielleicht waren Sie auch überrascht herauszufinden, dass MySQL nicht immer den von Ihnen erwarteten Index verwendet.

Für viele Benutzer stellen Indizes so etwas wie schwarze Magie dar. Manchmal scheinen sie Wunder zu wirken, während sie ein andermal das Einfügen verlangsamen und nur im Weg sind. Und dann kommt es auch vor, dass sie zu Beginn wunderbar funktionieren, um dann langsam in der Wirkung nachzulassen.

In diesem Kapitel sehen wir uns einige der hinter der Indizierung stehenden Konzepte an und stellen die verschiedenen Arten von Indizes vor, die von MySQL unterstützt werden. Danach betrachten wir einige Eigenarten der MySQL-Implementierung von Indizes. Das Kapitel endet mit Empfehlungen für die Wahl der zu indizierenden Spalten sowie der längerfristigen Pflege Ihrer Indizes.

Grundlagen der Indizierung

Um verstehen zu können, wie MySQL Indizes verwendet, sollten Sie zuerst die grundlegende Funktionsweise und die Fähigkeiten von Indizes verstehen. Erst dann können Sie intelligentere Entscheidungen darüber treffen, wie man sie richtig einsetzt.

Indizierungskonzepte

Um zu verstehen, was MySQL dank Indizes machen kann, macht man sich am besten erst mal klar, wie MySQL arbeitet, um eine Query auszuführen. Stellen Sie sich vor, dass `phone_book` eine Tabelle ist, die alle Telefonbücher des Staats Kalifornien umfasst und somit ungefähr 35 Millionen Einträge enthält. Denken Sie außerdem daran, dass die

Datensätze innerhalb der Tabellen nicht unbedingt sortiert sein müssen. Nehmen wir die folgende Query:

```
SELECT * FROM phone_book WHERE last_name = 'Zawodny'
```

Ohne irgendeine Form von Index zu konsultieren, muss MySQL alle Datensätze in der `phone_book`-Tabelle lesen und das `last_name`-Feld auf eine Übereinstimmung mit dem String `'Zawodny'` vergleichen. Natürlich ist das nicht effektiv. Mit einer steigenden Anzahl von Einträgen erhöht sich auch der zum Auffinden eines bestimmten Datensatzes notwendige Aufwand proportional. In der Informatik bezeichnet man das als $O(n)$-Problem.

Bei einem echten Telefonbuch wissen wir aber ganz genau, wie wir jemanden mit dem Namen Zawodny schnell heraussuchen können: Wir springen zum Z am Ende des Buchs und beginnen dort unsere Suche. Weil der zweite Buchstabe ein »a« ist, wissen wir, dass die möglichen Treffer ziemlich am Anfang der Liste aller mit Z beginnenden Namen zu finden sind. Die von uns verwendete Methode basiert auf dem Wissen über die Daten und über deren Sortierung.

Wir schummeln, oder? Überhaupt nicht! Sie können die Zawodnys deshalb so schnell finden, weil sie alphabetisch nach dem Nachnamen einsortiert sind. Darum sind sie einfach zu finden, solange Sie das Alphabet beherrschen.

Die meisten technischen Bücher (wie dieses) stellen am Ende einen Index zur Verfügung. Hier können Sie wichtige Begriffe und Konzepte schnell finden, weil sie (zusammen mit den dazugehörigen Seitennummern) in alphabetischer Reihenfolge sortiert sind. Sie wollen wissen, wo *mysqlhotcopy* diskutiert wird? Schlagen Sie die Seitennummer einfach im Index nach.

Datenbank-Indizes funktionieren ähnlich. Genau wie ein Buchautor oder Verlag einen Index für die wichtigen Konzepte und Begriffe eines Buchs erzeugen kann, können Sie einen Index für eine bestimmte Spalte einer Datenbanktabelle anlegen. In unserem obigen Beispiel könnten Sie einen Index für den Nachnamen anlegen, um Telefonnummern schneller nachschlagen zu können:

```
ALTER TABLE phone_book ADD INDEX (last_name)
```

Damit weisen Sie MySQL an, eine sortierte Liste aller Nachnamen der `phone_book`-Tabelle anzulegen. Zusammen mit den Namen hält es die genaue Position des dazugehörigen Datensatzes fest – genau wie der Index am Ende dieses Buchs die Seitenzahlen für jeden Eintrag angibt.[1]

Aus Sicht des Datenbank-Servers dienen Indizes der schnellen Eliminierung von Datensätzen, die bei der Ausführung einer Query nicht in die Ergebnismenge passen. Ohne Indizes müsste MySQL (wie jeder andere Datenbank-Server) jeden Datensatz einer Tabelle durchgehen. Das ist nicht nur zeitaufwendig, sondern benötigt auch sehr viele I/O-Vorgänge der Festplatten und kann den Platten-Cache stark durcheinander bringen.

1 Das ist nicht ganz richtg. MySQL speichert nicht immer die Position des passenden Datensatzes ab. Warum das so ist, werden Sie gleich sehen.

Im richtigen Leben wird man nur selten dynamische Daten finden, die sortiert sind (und auch sortiert bleiben). Bücher sind ein Sonderfall; sie neigen nicht dazu, sich zu verändern.

Da MySQL eine separate Liste der indizierten Werte pflegt und diese bei der Änderung Ihrer Daten auch aktualisieren muss, wollen Sie mit Sicherheit nicht jede Spalte einer Tabelle indizieren. Indizes sind ein Kompromiss zwischen Platzbedarf und Zeit. Sie opfern etwas zusätzlichen Plattenplatz und ein wenig CPU-Overhead bei jeder `INSERT`-, `UPDATE`- und `DELETE`-Anweisung, um die meisten (wenn nicht alle) Ihrer Queries deutlich schneller zu machen.

Die MySQL-Dokumentation verwendet die Begriffe *Index* und *Schlüssel* (Key) gleichberechtigt. Die Aussage »`last_name` ist ein Schlüssel« in der `phone_book`-Tabelle bedeutet nichts anderes, als dass das `last_name`-Feld der `phone_book`-Tabelle indiziert ist.

Partielle Indizes

Indizes erkaufen sich Performance durch Platz. Manchmal wollen Sie aber nicht den Platz opfern, der für die gewünschte Performance notwendig wäre. Glücklicherweise gibt Ihnen MySQL ein hohes Maß an Kontrolle darüber, wie viel Platz für die Indizes verwendet wird. Vielleicht enthält Ihre `phone_book`-Tabelle ja 2 Milliarden Datensätze. Ein Index für `last_name` würde sehr viel Platz benötigen. Wäre `last_name` durchschnittlich 8 Byte lang, käme man für den Datenbereich des Indiz auf ungefähr 16 GByte. Dazu käme der Datensatz-Zeiger, der (egal was man macht) immer vorhanden ist und weitere 4 bis 8 Byte je Datensatz verbraucht.[2]

Statt den gesamten Nachnamen zu indizieren, könnten Sie sich auch auf die ersten vier Bytes beschränken:

```
ALTER TABLE phone_book ADD INDEX (last_name(4))
```

Auf diese Weise reduzieren Sie den Platzbedarf des Datenbereichs grob um die Hälfte. Der Nachteil ist, dass MySQL bei diesem Index nicht ganz so viele Zeilen eliminieren kann. Eine Query wie

```
SELECT * FROM phone_book WHERE last_name = 'Smith'
```

würde alle mit `Smit` beginnenden Datensätze abrufen, einschließlich solcher Namen wie `Smith`, `Smitty` etc. Die Query muss dann `Smitty` und alle anderen irrelevanten Zeilen aussortieren.

Mehrspaltige Indizes

Wie viele relationale Datenbank-Engines erlaubt auch MySQL das Anlegen aus mehreren Spalten bestehender Indizes:

```
ALTER TABLE phone_book ADD INDEX (last_name, first_name)
```

2 Das ist zugegebenermaßen eine starke Vereinfachung. MySQL besitzt Strategien, die die Größe der Indizes reduzieren, aber sie haben alle ihren Preis.

Solche Indizes können die Geschwindigkeit von Queries erhöhen, wenn Sie häufig alle Spalten in einer WHERE-Klausel abfragen oder wenn eine einzelne Spalte keine ausreichende Vielfalt aufweist. Natürlich können Sie partielle Indizes verwenden, um den notwendigen Raum zu reduzieren:

```
ALTER TABLE phone_book ADD INDEX (last_name(4), first_name(4))
```

In beiden Fällen würde eine Suche nach Josh Woodward schnell ausgeführt sein:

```
SELECT * FROM phone_book
 WHERE last_name = 'Woodward'
   AND first_name = 'Josh'
```

Dass Vorname und Nachname zusammen indiziert werden, bedeutet, dass MySQL Zeilen basierend auf diesen beiden Feldern eliminieren kann, was die Zahl der Datensätze, die berücksichtigt werden müssen, deutlich reduziert. Schließlich gibt es im Telefonbuch wesentlich mehr Menschen, deren Nachname mit »Wood« beginnt, als welche, deren Nachname mit »Wood« und deren Vorname mit »Josh« beginnt.

Bei der Diskussion um mehrspaltige Indizes werden einzeln indizierte Spalten häufig als »Teile des Schlüssels« (engl. *key parts*) bezeichnet. Mehrspaltige Indizes werden auch als zusammengesetzte (compound/composite) Indizes bezeichnet.

Warum erzeugt man also nicht einfach zwei Indizes, einen für last_name und einen für first_name? Nun, Sie können das tun, aber MySQL würde nicht beide gleichzeitig verwenden. Tatsächlich verwendet MySQL nur einen Index pro Tabelle je Query – außer bei UNIONs.[3] Diese Tatsache ist wichtig genug, um sie noch einmal zu wiederholen: *MySQL verwendet nur jeweils einen Index pro Tabelle je Query.*

Bei getrennten Indizes für first_name und last_name wählt MySQL entweder den einen oder den anderen. Hierzu trifft es eine qualifizierte Entscheidung darüber, welcher Index weniger Zeilen zurückliefern wird. Eine qualifizierte Entscheidung deshalb, weil MySQL einige Index-Statistiken mitführt, die Rückschlüsse darüber erlauben, wie die Daten aussehen werden. Diese Statistiken sind aber natürlich vreallgemeinert. Während sie MySQL häufig clevere Entscheidungen treffen lassen, könnte MySQL bei sehr ungünstig verteilten Daten nicht die optimale Entscheidung in Bezug auf den zu verwendenden Index treffen. Wir bezeichnen diese Daten als *unförmig*, wenn der indizierte Schlüssel in manchen Bereichen recht spärlich gefüllt (etwa mit X beginnende Namen) und in anderen stark konzentriert ist (etwa der Name Schmitz im Deutschen). Dies ist ein wichtiges Thema, auf das wir später noch einmal zurückkommen werden.

Index-Sortierung

Wie ordnet MySQL die Werte im Index an? Wenn Sie bereits mit einem anderen RDBMS gearbeitet haben, könnten Sie erwarten, dass MySQL eine spezielle Syntax besitzt, um einen Index in aufsteigender, absteigender oder sonst einer Reihenfolge anzuordnen.

3 Bei einer UNION wird jede logische Query separat ausgeführt und zu einem Ergebnis zusammengeführt.

MySQL gibt Ihnen aber keinerlei Kontrolle über seine interne Sortierung der Indexwerte. Es hat dazu keinen Grund. Seit der Version 4.0 führt es eine recht gute Optimierung für die Fälle durch, die bei anderen Datenbank-Systemen zu einer schlechteren Performance führen.

So könnten einige Datenbank-Produkte die folgende Query sehr schnell ausführen:

```
SELECT * FROM phone_book WHERE last_name = 'Zawodny'
ORDER BY first_name DESC
```

diese hingegen sehr langsam:

```
SELECT * FROM phone_book WHERE last_name = 'Zawodny'
ORDER BY first_name ASC
```

Warum? Weil einige Datenbanken die Indizes in absteigender Reihenfolge speichern und für das Lesen in dieser Reihenfolge optimiert sind. Im ersten Fall verwendet die Datenbank den mehrspaltigen Index, um die passenden Datensätze zu lokalisieren. Weil die Datensätze bereits in absteigender Reihenfolge sortiert sind, müssen sie nicht mehr sortiert werden. Im zweiten Fall findet der Server zwar alle passenden Datensätze, muss sie dann in einem zweiten Durchlauf aber sortieren.

MySQL ist clever genug, um »den Index rückwärts durchzugehen«, wenn es notwendig sein sollte. Es führt beide Queries sehr schnell aus und muss die Records in beiden Fällen nicht neu sortieren.

Indizes als Beschränkung (constraints)

Indizes werden nicht nur genutzt, um die passenden Zeilen einer Query zu ermitteln. Ein *eindeutiger Index* (unique index) legt beispielsweise fest, dass ein bestimmter Wert innerhalb der angegebenen Spalte nur einmal vorkommen darf.[4] Bei unserem Telefonbuch-Beispiel könnten wir etwa einen eindeutigen Index für `phone_number` anlegen, um sicherzustellen, dass jede Telefonnummer nur einmal auftaucht: [5]

```
ALTER TABLE phone_book ADD UNIQUE (phone_number)
```

Der eindeutige Index übernimmt gleich zwei Aufgaben. Er funktioniert genau wie jeder andere Index auch, wenn Sie auf der Telefonnummer basierende Queries ausführen:

```
SELECT * FROM phone_book WHERE phone_number = '555-7271'
```

Gleichzeitig wird der Wert aber auch bei jedem Einfügen bzw. Aktualisieren geprüft, um sicherzustellen, dass der Wert nicht bereits existiert. In dieser Hinsicht arbeitet der eindeutige Index als Beschränkung.

Eindeutige Indizes benötigen genauso viel Platz wie andere Indizes, d.h., es wird der Wert jeder Spalte und die Position des Records festgehalten. Das kann natürlich Platzverschwendung sein, wenn Sie den eindeutigen Index nur als Beschränkung und nicht als

4 Ausser für NULL natürlich. NULL ist immer ein Sonderfall.

5 Im richtigen Leben wäre das allerdings keine gute Taktik, was Ihnen jeder bestätigen wird, der sich einen Telefonanschluss mit seinen Nachbarn teilen musste.

Index nutzen. Anders ausgedrückt, könnten Sie den eindeutigen Index nutzen, um die Eindeutigkeit sicherzustellen, während Sie gleichzeitig nie eine Query verwenden, die diesen eindeutigen Wert verwendet. In diesem Fall gibt es für MySQL eigentlich keinen Grund, die Position eines jeden Datensatzes festzuhalten, da sie ihn ja nie verwenden werden.

Leider gibt es keine Möglichkeit, MySQL Ihre Absicht mitzuteilen. In Zukunft wird es für diesen speziellen Fall sehr wahrscheinlich ein Feature geben. Die MyISAM Storage-Engine unterstützt bereits eindeutige Spalten ohne Index (basierend auf einem Hash-System), allerdings wird dieser Mechanismus noch nicht auf SQL-Ebene bereitgestellt.

Cluster- und Sekundär-Indizes

Bei MyISAM-Tabellen werden Indizes in einer separaten Datei vorgehalten, die eine Liste der Primär- (und eventuell Sekundär-) Schlüssel enthalten, sowie den Byte-Offset des Records. Auf diese Weise wird sichergestellt, dass MySQL den Datensatz in der Datenbank schnell lokalisieren und sich dann dorthin bewegen kann. MySQL muss die Indizes auf diese Weise ablegen, weil die Datensätze grundsätzlich in einer zufälligen Reihenfolge vorliegen.

Bei *geclusterten Indizes* werden der Primärschlüssel und der Datensatz selbst »geclustert«, d.h. zusammengefasst, und die Datensätze werden in der Reihenfolge des Primärschlüssels gespeichert. InnoDB verwendet geclusterte Indizes. In der Oracle-Welt werden Cluster-Indizes als »Index-organisierte Tabellen« bezeichnet. Diese Bezeichnung macht die Beziehung zwischen dem Primärschlüssel und der Anordnung der Datensätze leicht verständlich.

Werden die Daten meist über den Primärschlüssel abgesucht, können Abfragen durch geclusterte Schlüssel unglaublich schnell sein. Bei einem normalen MyISAM-Index gibt es zwei Abfragen, einen für den Index und einen zweiten (über die im Index festgehaltene Position) für die Tabelle selbst. Bei geclusterten Indizes verweist ein einzelner Lookup direkt auf den fraglichen Datensatz.

Bei einigen Operationen sind geclusterte Indizes nicht ganz so effizient. Nehmen wir zum Beispiel an, dass ein sekundärer Index verwendet wird. In unserem Telefonbuch-Beispiel könnte etwa `last_name` als primärer Index verwendet werden und `phone_number` als sekundärer, und Sie führen die folgende Query aus:

```
SELECT * FROM phone_book WHERE phone_number = '555-7271'
```

MySQL sucht den `phone_number`-Index nach `555-7271` ab, dessen Primärschlüssel `Zawodny` enthält, weil der Nachname bei `phone_book` der primäre Index ist. MySQL positioniert dann auf den relevanten Eintrag in der Datenbank.

Mit anderen Worten: Auf dem Primärschlüssel basierende Lookups werden sehr schnell ausgeführt, während auf sekundären Indizes basierende Lookups mit der gleichen Geschwindigkeit ausgeführt werden wie normale MyISAM-Index-Lookups.

Aber unter günstigen (bzw. ungünstigen) Bedingungen kann ein geclusterter Index der Performance sogar schaden. Wird er zusammen mit einem sekundären Index verwendet, müssen Sie die Auswirkungen auf den Speicherplatz berücksichtigen. Sekundäre Indizes verweisen auf den Primärschlüssel und nicht auf den Datensatz. Wenn Sie also einen großen Wert indizieren und verschiedene sekundäre Indizes verwenden, haben Sie es letztendlich mit vielen Kopien des primären Index zu tun: einmal als geclusterten Index zusammen mit den Daten selbst und dann noch einmal für jeden sekundären Index, der auf den geclusterten Index verweist. Hat der Primärschlüssel einen kurzen Wert, ist das nicht so schlimm, aber bei einem potenziell langen Wert (etwa einer URL), kann diese wiederholte Speicherung des Primärschlüssels auf der Platte zu Speicherproblemen führen.

Eine weniger häufig vorkommende, aber ebenso problematische Bedingung tritt ein, wenn Daten so verändert werden, dass sich dabei auch der Primärschlüssel ändert. Das ist die teuerste Operation eines geclusterten Index. Es müssen dann nämlich einige Dinge ausgeführt werden, die so richtig auf die Performance gehen:

- Ändern des fraglichen Records entsprechend der ausgeführten Query.
- Den neuen Primärschlüssel basierend auf den neuen Daten des Datensatzes bestimmen.
- Die gespeicherten Records neu so anordnen, dass der fragliche Datensatz an die richtige Stelle im Tablespace verschoben wird.
- Aktualisierung jeglicher sekundärer Indizes, die auf diesen Primärschlüssel verweisen.

Wie Sie sich vorstellen können, dauert ein `UPDATE`-Befehl zur Aktualisierung des Primärschlüssels für eine Reihe von Records eine gewisse Zeit, besonders bei größeren Tabellen. Wählen Sie die Primärschlüssel also mit Bedacht aus. Verwenden Sie Werte, die sich nicht verändern, etwa die Sozialversicherungsnummer an Stelle des Nachnamens oder die Seriennummer an Stelle des Produktnamens etc.

Eindeutige Indizes versus Primärschlüssel

Wenn Sie bereits mit anderen relationalen Datenbanken gearbeitet haben, fragen Sie sich vielleicht, worin bei MySQL der Unterschied zwischen einem Primärschlüssel und einem eindeutigen Index besteht. Das hängt wie immer davon ab. Bei MyISAM-Tabellen gibt es praktisch keinen Unterschied. Das einzig Besondere an einem Primärschlüssel ist die Tatsache, dass er keine NULL-Werte enthalten darf. Der Primärschlüssel ist einfach ein `NOT NULL UNIQUE INDEX` namens `PRIMARY`. Bei MyISAM-Tabellen müssen Sie keinen Primärschlüssel deklarieren.

InnoDB- und BDB-Tabellen verlangen Primärschlüssel für jede Tabelle. Allerdings müssen Sie selbst keinen festlegen, d.h., die Storage-Engine fügt automatisch einen internen Primärschlüssel ein, wenn Sie keinen angeben. In beiden Fällen sind Primärschlüssel einfach inkrementierte numerische Werte, ähnlich einer `AUTO-INCREMENT`-Spalte. Wenn Sie

sich zu einem späteren Zeitpunkt entscheiden, einen eigenen Primärschlüssel einzufügen, verwenden Sie einfach `ALTER TABLE`, um einen solchen hinzuzufügen. Beide Storage-Engines verwerfen dann die intern generierten Schlüssel zu Gunsten Ihres Schlüssels. Heap-Tabellen benötigen keinen Primärschlüssel, erzeugen aber einen für Sie. Tatsächlich können Heap-Tabellen ohne jegliche Indizes angelegt werden.

Indizierte NULL-Attribute

Man vergisst häufig, dass SQL bei der Durchführung logischer Operationen eine mit drei Zuständen arbeitende Logik (tristate) verwendet. Solange eine Spalte nicht mit `NOT NULL` deklariert ist, kann ein logischer Vergleich drei Ergebnisse zurückliefern. Der Vergleich kann wahr sein, wenn die Werte übereinstimmen, oder falsch, wenn sie nicht übereinstimmen, oder es gibt keine Übereinstimmung, weil einer der Werte NULL ist. Wenn einer der Werte NULL ist, dann ist auch das Ergebnis NULL.

Für Programmierer steht NULL häufig für undefiniert oder unbekannt. Auf diese Weise kann man dem Datenbank-Server mitteilen, dass ein unbekannter Wert vorliegt. Wie wirken sich NULL-Werte nun auf Indizes aus?

NULL-Werte können in normalen (nicht eindeutigen) Indizes verwendet werden. Das gilt für alle Datenbank-Server. Allerdings erlaubt MySQL im Gegensatz zu vielen anderen Datenbank-Servern die Verwendung von NULL-Werten in eindeutigen Indizes.[6] Sie können in einem solchen Index so viele NULL-Werte ablegen, wie Sie wollen. Das sieht vielleicht wenig intuitiv aus, ist aber die Natur von NULL. Weil NULL einen undefinierten Wert darstellt, muss MySQL davon ausgehen, dass alle NULL-Werte gleich sind, wenn es nur einen einzelnen Wert in einem eindeutigen Index erlaubt.

Um die Sache noch etwas interessanter zu machen, darf ein NULL-Wert nur einmal als Primärschlüssel auftauchen. Warum? Weil der SQL-Standard dieses Verhalten festlegt. Das ist eines der wenigen Dinge, in denen sich Primärschlüssel von eindeutigen Indizes bei MySQL unterscheiden. Und, falls Sie sich das fragen sollten, NULL-Werte im Index wirken sich nicht auf die Performance aus.

Index-Strukturen

Nachdem wir einige grundlegende Ideen erläutert haben, die sich hinter der Indizierung verbergen, wollen wir uns die verschiedenen Typen (oder Strukturen) von Indizes bei MySQL ansehen. Keiner der vorgestellten Indextypen ist MySQL-spezifisch. Sie finden vergleichbare Indizes bei PostgreSQL, DB2, Oracle etc.

Statt sich zu sehr mit den Details der Implementierung zu beschäftigen,[7] wollen wir uns ansehen, für welche Art von Daten bzw. Anwendungen jeder Typ entworfen wurde.

6 MySQL-Version 3.23 und ältere erlauben das nicht, die Versionen 4.0 und höher schon.

7 Wie bei vielen Produkten ändern sich die Implementierungsdetails mit der Zeit. Wenn Sie versuchen, diese Interna für Ihre Zwecke zu nutzen, öffnen Sie Problemen im Fall von Änderungen Tür und Tor.

Dabei wollen wir Antworten auf solche Fragen liefern wie: Welche Indextypen sind die schnellsten, die flexibelsten, und welche benötigen den meisten bzw. geringsten Platz?

Wäre dies eine allgemeine Einführung in die Informatik, würden wir wohl tiefer in die jeweiligen Datenstrukturen und Algorithmen eintauchen, die hier hinter den Kulissen arbeiten. Da das aber nicht der Fall ist, wollen wir uns auf die Praxis beschränken. Wenn Sie wirklich einen Blick hinter die Kulissen werfen wollen, sei auf die vielen ausgezeichneten Bücher verwiesen, die sich mit diesem Thema beschäftigen.

B-Tree-Indizes

Der B-Tree (»balanced tree«), also der ausbalancierte Baum, ist der am weitesten verbreitete Indextyp. Nahezu alle Datenbank-Server und Datenbank-Bibliotheken bieten B-Tree-Indizes an, häufig als den standardmäßig verwendeten Indextyp. Die einmalige Kombination aus Flexibilität und Größe sowie die insgesamt gute Performance machen diesen Typ üblicherweise zur ersten Wahl.

Wie es der Name andeutet, ist ein B-Tree eine Baumstruktur. Die Knoten werden (basierend auf den Schlüsselwerten) sortiert abgelegt. Ein B-Tree wird als »balanciert« bezeichnet, weil er nie in »Schieflage« gerät, wenn Knoten hinzugefügt oder entfernt werden. Der Hauptvorteil dieser Gleichmäßigket besteht darin, dass die Performance eines B-Trees selbst im schlimmsten Fall recht gut ist. B-Trees bieten eine $O(\log n)$-Performance bei Lookups einzelner Records. Im Gegensatz zu binären Bäumen, bei denen jeder Knoten maximal zwei Unterelemente enthält, können B-Trees viele Schlüssel pro Knoten enthalten und wachsen nicht so schnell in die Breite oder Tiefe wie binäre Bäume.

B-Tree-Indizes bieten sehr viel Flexibilität, wenn es um die Auflösung von Queries geht. Bereichsbasierte Queries wie die folgende können sehr schnell beantwortet werden:

```
SELECT * FROM phone_book WHERE last_name
BETWEEN 'Marten' and 'Mason'
```

Der Server findet einfach den ersten Datensatz mit »Marten« und den letzten mit »Mason« und weiß, dass alles Dazwischenliegende auch einen Treffer darstellt. Das Gleiche gilt auch für nahezu jede Query, bei der es um Wertebereiche geht, einschließlich `MIN( )` und `MAX( )`, aber auch für nach oben offene Queries wie:

```
SELECT COUNT(*) FROM phone_book WHERE last_name > 'Zawodny'
```

MySQL sucht sich einfach den letzten Zawodny heraus und zählt dann alle Records, die im Indexbaum dahinter liegen.

Hash-Indizes

Der zweite weit verbreitete Indextyp basiert auf dem Hash. Diese *Hash-Indizes* ähneln eher einer Hash-Tabelle denn einem Baum. Verglichen damit ist ihre Struktur sehr flach. Statt Datensätze basierend auf einem Vergleich der Schlüsselwerte mit anderen Schlüsseln anzuordnen, werden Hash-Indizes basierend auf dem Ergebnis einer *Hash-Funktion*

angeordnet, die jeder Schlüssel durchlaufen muss. Die Aufgabe der Hash-Funktion besteht darin, einen halb eindeutigen Hash-Wert (üblicherweise numerisch) für einen gegebenen Schlüssel zu erzeugen. Dieser Wert wird dann genutzt, um zu bestimmen, in welchem Fach (Bucket) der Schlüssel abgelegt wird.

Nehmen wir eine gängige Hash-Funktion wie `MD5()`. Für einander recht ähnliche Strings werden deutlich unterschiedliche Ergebnisse generiert:

```
mysql> SELECT MD5('Smith');
+----------------------------------+
| MD5('Smith')                     |
+----------------------------------+
| e95f770ac4fb91ac2e4873e4b2dfc0e6 |
+----------------------------------+
1 row in set (0.46 sec)

mysql> SELECT MD5('Smitty');
+----------------------------------+
| MD5('Smitty')                    |
+----------------------------------+
| 6d6f09a116b2eded33b9c871e6797a47 |
+----------------------------------+
1 row in set (0.00 sec)
```

Nun erzeugt der MD5-Algorithmus 128-Bit-Werte (die standardmäßig in Base-64 dargestellt werden), was bedeutet, dass es über $3{,}4 \times 10^{38}$ mögliche Werte gibt. Weil die meisten Computer nicht annähernd so viel Plattenplatz (geschweige denn Arbeitsspeicher) besitzen, sind Hash-Tabellen immer auf den verfügbaren Speicherplatz beschränkt.

Eine gängige Technik, den möglichen Schlüsselplatz für die Hash-Tabelle zu beschränken, besteht darin, eine feste Anzahl von Speicherplätzen (Buckets) zu allozieren (meist eine relativ große Primzahl wie 35149). Dann dividieren Sie das Ergebnis der Hash-Funktion durch die Primzahl und nutzen den Rest, um festzulegen, in welchen Bucket der Wert eingetragen wird.

So weit die Theorie. Die Details der Implementierung können etwas komplizierter sein, und sie zu kennen hilft einem nicht weiter. Das Endergebnis ist, dass der Hash-Index sehr schnelle Lookups ermöglicht, generell O(1), solange die Hash-Funktion die Werte für Ihre Daten gut verteilt.

Obwohl Hash-basierte Indizes generell einen der schnellsten Schlüssel-Lookups bieten, sind sie weniger flexibel und nicht so gut vorhersehbar wie andere Indizes. Sie sind weniger flexibel, weil bereichsbasierte Queries den Index nicht nutzen können. Gute Hash-Funktionen erzeugen sehr unterschiedliche Werte für ähnliche Daten, so dass der Server keine Annahmen über die Reihenfolge der Daten innerhalb der Indexstruktur treffen kann. In einer Hash-Tabelle nahe beieinander liegende Werte sind einander selten ähnlich. Hash-Indizes sind weniger vorhersehbar, weil eine schlechte Kombination aus Daten und Hash-Funktion dazu führen kann, dass die meisten Datensätze auf nur wenige Buckets verteilt sind. Wenn das passiert, leidet die Performance ein wenig. Statt

sich durch eine relativ kleine Liste von Schlüsseln mit dem gleichen Hash-Wert zu arbeiten, muss der Computer eine größere Liste untersuchen.

Hash-Indizes funktionieren bei den meisten textbasierten und numerischen Datentypen recht gut. Weil Hash-Funktionen beliebig große Schlüssel recht effizient auf kleine Hash-Werte reduzieren, verbrauchen sie tendenziell nicht so viel Speicher wie baumbasierte Indizes.

R-Tree-Indizes

R-Tree-Indizes werden für räumliche oder N-dimensionale Daten genutzt. Sie sind in kartografischen und geowissenschaftlichen Anwendungen recht populär, funktionieren aber in allen Situationen, bei denen Daten basierend auf zwei Achsen oder Dimensionen abgefragt werden: Länge und Breite, Höhe und Gewicht etc.

R-Tree-Indizes wurden in der Version 4.1 eingeführt, sind also relativ neu bei MySQL. Die MySQL-Implementierung basiert auf der OpenGIS-Spezifikation, die online über *http://www.opengis.org* zur Verfügung steht. Die Unterstützung für räumliche Daten bei anderen populären Datenbank-Servern basiert häufig auf den OpenGIS-Spezifikationen, so dass Sie mit der Syntax vertraut sein sollten, wenn Sie schon mit ähnlichen Produkten gearbeitet haben.

N-dimensionale Indizes könnten auch langjährigen MySQL-Benutzern fremd sein, deshalb wollen wir uns hier ein einfaches Beispiel ansehen. Wir werden eine Tabelle mit räumlichen Daten anlegen, mehrere Punkte in x-/y-Koordinaten angeben und MySQL dann fragen, welche Punkte innerhalb der Grenzen eines Polygons liegen.

Zuerst legen wir eine Tabelle mit einem kleinen `BLOB`-Feld an, das die räumlichen Daten enthält:

```
mysql> create table map_test
    -> (
    ->   name varchar(100) not null primary key,
    ->   loc  geometry,
    ->   spatial index(loc)
    -> );
Query OK, 0 rows affected (0.00 sec)
```

Dann fügen wir einige Punkte ein:

```
mysql> insert into map_test values ('One Two', point(1,2));
Query OK, 1 row affected (0.00 sec)

mysql> insert into map_test values ('Two Two', point(2,2));
Query OK, 1 row affected (0.00 sec)

mysql> insert into map_test values ('Two One', point(2,1));
Query OK, 1 row affected (0.00 sec)
```

Sehen wir uns nun an, ob die Tabelle in Ordnung ist:

```
mysql> select name, AsText(loc) from map_test;
```

```
+---------+------------+
| name    | AsText(loc) |
+---------+------------+
| One Two | POINT(1 2)  |
| Two Two | POINT(2 2)  |
| Two One | POINT(2 1)  |
+---------+------------+
3 rows in set (0.00 sec)
```

Abschließend fragen wir MySQL, welche Punkte innerhalb des Polygons liegen:

```
mysql> SELECT name FROM map_test WHERE
    -> Contains(GeomFromText('POLYGON((0 0, 0 3, 3 3, 3 0, 0 0))'), loc);
+---------+
| name    |
+---------+
| One Two |
| Two Two |
| Two One |
+---------+
3 rows in set (0.00 sec)
```

Abbildung 4-1 zeigt die Punkte und das Polygon in einem Diagramm.

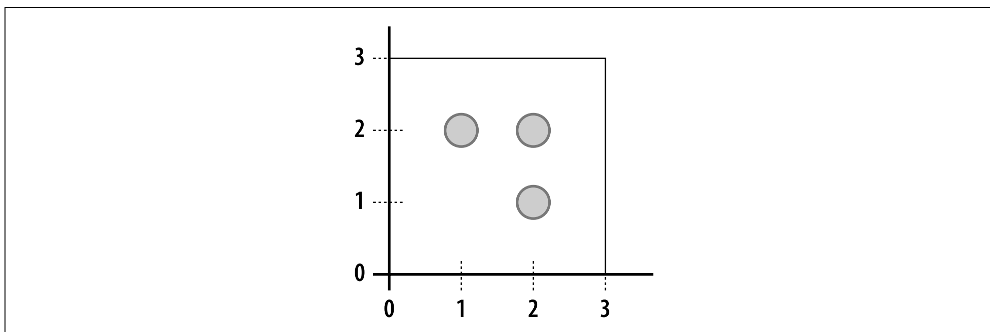

Abbildung 4-1: 2-D-Punkte und ein sie enthaltendes Polygon

MySQL indiziert die verschiedenen Formen, die mit Hilfe des MBR (Minimum Bounding Rectangle, also des minimal umschließenden Rechtecks) repräsentiert werden kann (Punkte, Linien, Polygone). Zu diesem Zweck berechnet es das kleinste Rechteck, das Sie zeichnen müssen, um die Form vollständig zu umschließen. MySQL speichert die Koordinaten dieses Rechtecks ab und nutzt es, wenn es versucht, die verschiedenen Formen für einen bestimmten Bereich zu ermitteln.

Indizes und Tabellentypen

Nachdem wir die gängigen Indextypen, die Terminologie und Anwendungen bislang relativ allgemein behandelt haben, wollen wir uns nun die Indizes ansehen, die in den jeweiligen MySQL Storage-Engines implementiert sind. Jede Engine implementiert eine

Untermenge der drei von uns betrachteten Indextypen. Sie bieten darüber hinaus verschiedene Optimierungen an, deren Sie sich bewusst sein sollten.

MyISAM-Tabellen

Der von MySQL standardmäßig verwendete Tabellentyp stellt B-Tree-Indizes zur Verfügung und seit der Version 4.1.0 auch R-Tree-Indizes für räumliche Daten. Neben den üblichen Vorteilen einer guten B-Tree-Implementierung nutzt MyISAM zwei weitere wichtige, aber relativ unbekannte Features: Präfix-Komprimierung und gepackte Schlüssel.

Die Präfix-Komprimierung wird genutzt, um gängige Präfixe in String-Schlüsseln zu eliminieren. Bei einer URLs speichernden Tabelle wäre es für MySQL Platzverschwendung, das »http://« in jedem Knoten des B-Trees abzuspeichern. Da es bei einer großen Anzahl von Schlüsseln vorkommt, wird dieses gängige Präfix komprimiert, so dass deutlich weniger Platz benötigt wird.

Gepackte Schlüssel kann man sich am besten als Präfix-Komprimierung für Integer-Schlüssel vorstellen. Weil Integer-Schlüssel mit den höherwertigen Bytes zuerst abgespeichert werden, ist für eine große Gruppe von Schlüsseln ein gemeinsames Präfix durchaus üblich, weil sich die höchsten Bits der Zahl deutlich seltener ändern. Um gepackte Schlüssel zu aktivieren, hängen Sie einfach

```
PACK_KEYS = 1
```

an die `CREATE TABLE`-Anweisung an.

MySQL speichert die Indizes einer Tabelle in der *.MYI*-Datei der Tabelle.

Verzögertes Schreiben von Schlüsseln

Ein die Performance steigerndes Feature von MyISAM-Tabellen ist die Fähigkeit, das Schreiben der Indexdaten auf die Festplatte zu verzögern. Normalerweise schreibt MySQL modifizierte Schlüsselblöcke sofort auf die Platte, sobald Änderungen vorgenommen wurden, aber Sie können dieses Verhalten global oder tabellenbezogen ändern. Das führt zu einer deutlichen Leistungssteigerung bei starken `INSERT`-, `UPDATE`- und `DELETE`-Operationen.

Die MySQL-Option `delay_key_write` steuert dieses Verhalten über drei mögliche Einstellungen. Die Voreinstellung, `ON`, bedeutet, dass MySQL die `DELAY_KEY_WRITE`-Option in `CREATE TABLE` berücksichtigt. Setzen Sie es auf `OFF`, wird das Schreiben von Schlüsseln durch MySQL nie verzögert. Die Einstellung `ALL` weist MySQL an, das Schreiben der Schlüssel bei allen MyISAM-Tabellen zu verzögern, und zwar unabhängig davon, welcher `DELAY_KEY_WRITE`-Wert beim Anlegen der Tabelle verwendet wurde.

Der Nachteil des verzögerten Schreibens von Schlüsseln besteht darin, dass die Indizes nach einem Absturz von MySQL nicht mit den Daten übereinstimmen müssen, wenn ungeschriebene Daten im Puffer lagen. Um dieses Problem zu beheben, ist ein `REPAIR TABLE` notwendig, das alle Indizes wieder aufbaut und möglicherweise sehr viel Zeit benötigt.

Heap-Tabellen

MySQLs einziger speicherbasierter Tabellentyp unterstützte ursprünglich nur Hash-Indizes. Seit der Version 4.1.0 können Sie bei Heap-Tabellen aber zwischen B-Tree- und Hash-Indizes wählen. Standardmäßig wird immer noch ein Hash-Index verwendet, aber einen B-Tree festzulegen ist einfach:

```
mysql> create table heap_test (
    -> name varchar(50) not null,
    -> index using btree (name)
    -> ) type = HEAP;
Query OK, 0 rows affected (0.00 sec)
```

Um zu prüfen, ob der Index korrekt angelegt wurde, verwenden Sie den Befehl `SHOW KEYS`:

```
mysql> show keys from heap_test \G
*************************** 1. row ***************************
       Table: heap_test
  Non_unique: 1
    Key_name: name
Seq_in_index: 1
 Column_name: name
   Collation: A
 Cardinality: NULL
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
1 row in set (0.00 sec)
```

Durch die Kombination der Flexibilität von B-Tree-Indizes mit der hohen Geschwindigkeit einer In-Memory-basierten Tabelle ist die Query-Performance dieser temporären Tabellen nicht zu schlagen. Wenn Sie nur schnelle Lookups einzelner Schlüssel benötigen, reichen Standard-Hash-Indizes bei Heap-Tabellen aber möglicherweise aus. Sie sind äußerst schnell und gehen sehr effizient mit Platz und Speicher um.

Die Indexdaten für Heap-Tabellen werden immer im Speicher abgelegt – genau wie die Daten.

BDB-Tabellen

MySQLs Berkeley DB-(BDB-)Tabellen stellen nur B-Tree-Indizes zur Verfügung. Das mag langjährige BDB-Nutzer überraschen, die mit den zu Grunde liegenden Hash-basierten Indizes vertraut sind. Die Indizes werden in der gleichen Datei abgelegt wie die Daten selbst.

BDB-Indizes bieten (genau wie MyISAM) eine Präfix-Komprimierung an. Wie InnoDB verwendet BDB auch geclusterte Indizes, und BDB-Tabellen benötigen einen Primärschlüssel. Wenn Sie keinen definieren, legt MySQL selbst einen verdeckten Primärschlüssel an, den es intern zur Lokalisierung von Zeilen verwendet. Das ist notwendig, weil

BDB immer den Primärschlüssel zur Lokalisierung von Datensätzen verwendet. Indexeinträge verweisen auf die jeweiligen Zeilen immer über den Primärschlüssel und nicht über die physikalische Lage des Datensatzes. Das bedeutet, dass Lookups über sekundäre Indizes etwas langsamer sind als Lookups über den Primärschlüssel.

InnoDB-Tabellen

InnoDB-Tabellen arbeiten mit B-Tree-Indizes. Das Packen oder eine Präfix-Komprimierung ist für die Indizes nicht möglich. Außerdem verlangt InnoDB ebenfalls einen Primärschlüssel für jede Tabelle. Wenn Sie keinen Primärschlüssel angeben, stellt MySQL wie bei BDB einen 64-Bit-Wert für Sie bereit.

Die Indizes werden genau wie die Daten und das Data Dictionary (Tabellendefinition etc.) im InnoDB-Tablespace gespeichert. InnoDB verwendet auch geclusterte Indizes. Das bedeutet, dass der Wert des Primärschlüssels direkt die physikalische Lage der Zeile und den dazugehörigen Indexknoten beeinflusst. Aus diesem Grund sind auf dem Primärschlüssel basierende Lookups bei InnoDB sehr schnell. Sobald ein Indexknoten gefunden wurde, liegen die relevanten Records sehr wahrscheinlich schon im InnoDB-Puffer-Pool vor.

Volltextindizes

Ein Volltextindex ist ein spezieller Indextyp, der sehr schnell die Lage jedes unterschiedlichen Worts in einem Feld zurückliefert. MySQL unterstützt die Volltextindizierung in MyISAM-Tabellen. Volltextindizes werden über ein oder mehrere Textfelder (`VARCHAR`, `TEXT` etc.) einer Tabelle aufgebaut.

Der Volltextindex wird ebenfalls in der *.MYI*-Datei abgelegt. Er ist als normaler zweiteiliger MyISAM-B-Tree-Index implementiert, bei dem das erste Feld ein `VARCHAR` ist und das zweite ein `FLOAT`. Das erste Feld enthält das indizierte Wort, das `FLOAT`-Feld die Gewichtung innerhalb der Zeile.

Da sie generell einen Datensatz für jedes Wort jedes indizierten Felds enthalten, können Volltextindizes schnell sehr groß werden. Glücklicherweise sind die B-Tree-Indizes bei MySQL recht effektiv, so dass der durch einen Volltext verbrauchte Platz den Performance-Schub durchaus rechtfertigt.

Es ist für eine Query wie

```
select * from articles where body = "%database%"
```

nicht ungewöhnlich, tausende Male schneller zu laufen, wenn ein Volltextindex hinzugefügt und die Query wie folgt umgeschrieben wird:

```
select * from articles (body) match against ('database')
```

Wie bei allen Indextypen erkauft man sich Geschwindigkeit durch Platz.

Einschränkungen von Indizes

Es gibt viele Fälle, in denen MySQL nicht einfach einen Index nutzen kann, um eine Query zu beantworten. Um Ihnen zu helfen, diese Einschränkungen zu erkennen (und hoffentlich zu vermeiden), wollen wir uns die vier wichtigsten Hindernisse bei der Verwendung eines Index ansehen.

Wildcard-Matching

Eine Query, die alle das Wort »buffy« enthaltenden Records ausfindig machen soll:

```
select * from pages where page_text like "%buffy%"
```

ist garantiert langsam. Sie verlangt von MySQL die Verarbeitung jeder Zeile der Tabelle. Und sie muss nicht einmal alle Vorkommen erkennen, weil auf »buffy« irgendein Interpunktionszeichen folgen kann. Die Lösung besteht natürlich darin, einen Volltextindex für das `page_text`-Feld aufzubauen und eine Query mit MySQLs `MATCH AGAINST`-Syntax zu nutzen.

Wenn Sie mit Wortteilen arbeiten, werden die Dinge aber sehr schnell langsam. Stellen Sie sich vor, Sie möchten die Telefonnummer aller Teilnehmer ermitteln, deren Nachname den String »son« (etwa Johnson, Ansona oder Bronson) enthält. Die Query würde etwa so aussehen:

```
select phone_number from phone_book where last_name like "%son%"
```

Das sieht unserem »buffy«-Beispiel verdächtig ähnlich, und das ist es auch. Da Sie eine Wildcard-Suche im Feld vornehmen, muss MySQL jede Zeile verarbeiten, aber auch ein Wechsel auf einen Volltextindex würde in diesem Fall nichts nutzen. Volltextindizes arbeiten mit vollständigen Wörtern, d.h., sie können uns in dieser Situation nicht weiterhelfen.

Falls Sie das überrascht, denken Sie daran, auf welche Weise Sie diese Namen in einem normalen Telefonbuch ausfindig machen müssten. Finden Sie einen effizienten Ansatz? Es gibt wirklich keine praktikable Möglichkeit, die diese Art von Anfrage in einem gedruckten Telefonbuch zu vereinfachen.

Reguläre Ausdrücke

Die Verwendung regulärer Ausdrücke führt zu ähnlichen Problemen. Nehmen wir an, Sie möchten alle Nachnamen ermitteln, die auf »ith« (wie bei Smith) oder »son« (wie bei Johnson) enden. Jeder Perl-Hacker wird Ihnen sagen, dass das ganz leicht ist. Verwenden Sie einfach den regulären Ausdruck `(son|ith)$`.

Wenn man das auf MySQL anwendet, sieht die Query so aus:

```
select last_name from phone_book where last_name rlike "(son|ith)$"
```

Allerdings wird diese Query sehr langsam ausgeführt, und zwar aus dem gleichen Grund wie beim Suchen mit Hilfe eines Wildcard. Es gibt einfach keine effektive Lösung, einen

Index aufzubauen, der die Ausführung beliebiger Wildcard- und Regex-Suchen unterstützt.

In diesem speziellen Fall können Sie die Beschränkung umgehen, indem Sie die Nachnamen in umgekehrter Reihenfolge in einem zweiten Feld festhalten. Sie können die Suche dann umkehren und eine Query wie die folgende verwenden:

```
select last_name from phone_book where rev_last_name like "thi%"
union
select last_name from phone_book where rev_last_name like "nos%"
```

Das ist aber nur effektiv, weil Sie am Anfang des Strings beginnen, der eigentlich das Ende des Strings ist. Wie gesagt, es gibt keine allgemein gültige Lösung für dieses Problem.

Beachten Sie, dass ein regulärer Ausdruck in diesem Fall immer noch nicht effektiv ist. Sie könnten versucht sein, die folgende Query zu verwenden:

```
select last_name from phone_book where rev_last_name rlike "^(thi|nos)"
```

Allerdings wären Sie von der Performance sehr enttäuscht. Der MySQL-Optimizer unternimmt keinen Versuch, Regex-basierte Queries zu optimieren.

Fehlerhafte oder beschädigte Statistiken

Wenn die MySQL-internen Indexstatistiken beschädigt werden oder aus anderen Gründen falsch sind (etwa nach einem Absturz oder dem versehentlichen Herunterfahren des Servers), kann MySQL ein sehr merkwürdiges Verhalten an den Tag legen. Sind die Statistiken einfach falsch, könnten Sie feststellen, dass für Ihre Queries kein Index mehr verwendet wird. Oder der Index könnte nur zeitweise verwendet werden.

MySQL glaubt sehr wahrscheinlich, dass die Anzahl der auf Ihre Query passenden Zeilen so hoch ist, dass es effektiver ist, die Tabelle vollständig durchzugehen. Da solche Tabellen-Scans hauptsächlich aus sequenziellen Leseoperationen bestehen, sind diese effektiver, als einen Großteil der Records über einen Index abzurufen, der weitaus mehr Suchoperationen auf der Festplatte verlangt.

Wenn das passiert (oder wenn Sie den Verdacht haben, dass es so ist), sollten Sie die Befehle zur Reparatur und Analyse von Indizes verwenden, die im folgenden Abschnitt »Indexpflege« diskutiert werden.

Zu viele passende Datensätze

Die Performance kann auch dann in den Keller gehen, wenn eine Query sehr viele Treffer zurückliefert. Da stellt sich dann die Frage, wie viele Treffer für MySQL zu viele Treffer sind. Nun, das kommt darauf an, aber eine gute Faustformel besagt, dass mit einem Tabellen-Scan an Stelle eines Indexes gearbeitet wird, wenn etwa 30 % aller Zeilen als Treffer identifiziert werden. Es gibt einige wenige Ausnahmen von dieser Regel. Eine detaillierte Betrachtung dieses Problems finden Sie in Kapitel 5.

Indexpflege

Nachdem Sie Indizes angelegt und gelöscht haben und Ihre Anwendung zufrieden stellend läuft, werden Sie sich um fortlaufende Wartungs- und Administrationsaufgaben Gedanken machen. Die gute Nachricht lautet, dass Sie sich um nichts Besonderes kümmern müssen, es gibt aber eine Reihe von Dingen, die von Zeit zu Zeit erledigt werden sollten.

Indexinformationen abrufen

Wenn Sie schon jemals eine langsame Query oder ein Indizierungsproblem bei einer Tabelle (oder einer Gruppe von Tabellen) untersuchen mussten, mit der Sie sich schon längere Zeit nicht mehr befasst haben, benötigen Sie zuerst einige grundlegende Informationen. Welche Spalten sind indiziert? Wie viele Werte gibt es? Wie groß ist der Index?

Glücklicherweise ist es bei MySQL relativ einfach, diese Informationen abzurufen. Mit `SHOW CREATE TABLE` können Sie den kompletten SQL-Code abrufen, der zur Generierung (bzw. Regenerierung) der Tabelle notwendig ist. Sind Sie hingegen nur an den Indizes interessiert, liefert `SHOW INDEXES FROM` weitaus mehr Informationen zurück.

```
mysql> SHOW INDEXES FROM access_jeremy_zawodny_com \G
*************************** 1. row ***************************
       Table: access_jeremy_zawodny_com
  Non_unique: 1
    Key_name: time_stamp
Seq_in_index: 1
 Column_name: time_stamp
   Collation: A
 Cardinality: 9434851
    Sub_part: NULL
      Packed: NULL
        Null: YES
  Index_type: BTREE
     Comment:

1 rows in set (0.00 sec)
```

Sie können in der Query auch `INDEXES` durch `KEYS` ersetzen.

Die Tabelle im Beispiel verwendet einen einzelnen Index namens `time_stamp`. Es handelt sich um einen B-Tree-Index mit nur einer Komponente, nämlich der `time_stamp`-Spalte (d.h., es handelt sich nicht um einen mehrspaltigen Index). Der Index ist nicht gepackt und darf NULL-Werte enthalten. Außerdem ist er nicht eindeutig, d.h., Duplikate sind erlaubt.

Indexstatistiken aktualisieren

Wird eine Tabelle häufig geändert, schleichen sich mit der Zeit einige Ineffizienzen in die Indizes ein. Fragmentierung durch die Verschiebung von Blöcken auf der Platte und feh-

lerhafte Indexstatistiken sind die beiden Hauptprobleme, denen Sie am ehesten begegnen werden. Glücklicherweise ist es bei MySQL einfach, die Indexdaten für MyISAM-Tabellen zu optimieren.

Sie können den Befehl `OPTIMIZE TABLE` verwenden, um eine Tabelle neu zu indizieren. In diesem Fall liest MySQL alle Records der Tabelle ein und baut alle Indizes neu auf. Das Ergebnis sind eng gepackte Indizes mit vernünftigen statistischen Werten.

Denken Sie daran, dass die Reindizierung einer Tabelle recht lange dauern kann, wenn die Tabelle sehr groß ist. Während dieser Zeit ist die Tabelle durch MySQL schreibgeschützt, d.h., sie kann nicht aktualisiert werden.

Mit Hilfe des Kommandozeilen-Tools *myisamchk* können Sie diese Analyse offline vornehmen:

```
$ cd datenbank-name
$ myisamchk tabellen-name
```

Stellen Sie nur sicher, dass MySQL nicht läuft, während Sie es ausführen, weil Sie sonst eine Beschädigung Ihrer Indizes riskieren.

Bei BDB- und InnoDB-Tabellen ist die Notwendigkeit solcher Arbeiten recht gering. Das ist auch gut so, weil die Möglichkeiten der Reindizierung etwas zeitaufwendiger sind. Sie können alle Indizes manuell löschen und neu anlegen, oder Sie müssen die Tabelle mit einem Dump sichern und neu laden. Durch die Verwendung von `ANALYZE TABLE` für eine InnoDB-Tabelle werden die Daten aber neu einlesen, um bessere Statistiken erzeugen zu können.