**Index**

**Textzusammenfassung**

Bei dem nächsten Thema handelt es sich um Indexierung und die damit verbundene Performance – Optimierungen werden näher erläutert. Zunächst betrachten wir die Grundlagen der Indexierung, daraufhin die verschiedenen Arten von Indexes and am Ende die Auswirkungen im Hinblick auf die Performance.

Indizes (oder auch: Indexes) sind Datenstrukturen, die von Speicher-Engines (engl. storage engines) verwendet werden, um unter anderem Zeilen schneller zu finden. Sie haben einen großen Einfluss auf die Performance der Datenbank und werden umso wichtiger, je größer die Datenbank wird. Deshalb ist es möglich, dass weniger ausgelastete Datenbanken ohne ordnungsgemäße Indizes gut funktionieren, aber die Leistung kann schnell sinken, wenn die Datenmenge wächst. Wenn man ein solches Problem hat, dann ist Index – Optimierung vielleicht der leistungsstärkste Weg, die Abfrageleistung zu verbessern (\cite{buch\_07, mysql}). Um wirklich optimale Indizes zu erstellen, ist es oft erfordert, dass die Abfragen umschrieben werden. Wie genau man Indizes erstellen muss, betrachten wir im späteren Verlauf der Arbeit.

Um die Funktionsweise eines Indizes am einfachsten zu verstehen, schauen wir uns ein Beispiel anhand eines wissenschaftlichen Fachbuches an. Am Ende dieser Bücher gibt es meist ein Stichwortverzeichnis oder Register. Dieses Register besteht aus einer alphabetisch geordneten Liste von Begriffen, Themen und Stichworten. Wenn man ein Begriff nachschlagen will, dann sucht man das Wort in der Liste und sieht zu diesem Wort die Seitenzahlen, auf denen der Begriff vorkommt. In MySQL verwendet die Storage-Engine Indizes auf ähnliche Weise. Sie durchsucht die Datenstruktur des Index nach einem Wert. Wenn ein Treffer gefunden wird, kann die Engine die Zeile finden, die den Treffer enthält. Betrachten wir uns dieses Beispiel an:

* ***SELECT name FROM customer WHERE cust\_id = 7;***

Es gibt einen Index auf der Spalte ***cust\_id***, sodass MYSQL den Index verwendet, um Zeilen zu finden, deren cust\_id gleich 7 ist. Mit anderen Worten wird eine Suche in den Werten des Index durchgeführt und alle Zeilen, die den angegebenen Wert enthalten, werden zurückgegeben.

Ein Index kann dabei Werte aus einer oder aus mehreren Spalten einer Tabelle enthalten. Bei mehreren Spalten ist zudem die Reihenfolge der Spalten im Index sehr wichtig, da MySQL nur effizient auf ein linkes Präfix des Indexes suchen kann. Ein Index auf zwei Spalten ist auch nicht dasselbe wie das Erstellen von zwei einzelnen Spalten – Indizes. Es gibt verschiedene Index-Typen, die jeweils für unterschiedliche Zwecke optimiert sind und die wir uns im nächsten Unterkapitel angucken.

Indizes werden in der Storage – Engine – Ebene implementiert, nicht in der Server – Ebene und daher sind sie nicht standardisiert und unterscheiden sich je nach Engine. Unter anderen werden unterstützen nicht alle Engines alle Indextypen. Eine Storage Engine ist eine Kernkomponente eines Datenbankmanagementsystems (DBMS), die für die Speicherung und Verwaltung der Daten verantwortlich ist. Sie entscheidet, wie Daten physisch auf dem Speichermedium organisiert, gespeichert und abgerufen werden. Verschiedene Storage Engines unterschieden sich, wie oben festgestellt, in der Funktionalität der Indizes, aber auch der Transaktionsunterstützung und Sperrmechanismen.

**B-Baum-Index**

Der erste Indextyp, den wir betrachten, ist der **B-Baum-Index** (engl. *B-Tree Index*), der auf einer speziellen Baum-Datenstruktur basiert. Diese Struktur wird von den meisten MySQL-Storage-Engines unterstützt. Allerdings können die Implementierung und Nutzung des B-Baum-Indexes je nach verwendeter Storage Engine variieren.

Das allgemeine Prinzip eines B-Baums ist, dass alle Werte in einer Reihenfolge gespeichert sind und jede Blatt-Seite den gleichen Abstand zum Wurzelknoten hat. Ein B-Baum-Index beschleunigt den Datenzugriff, weil die Storage-Engine nicht die ganze Tabelle durchlaufen muss, um die gewünschten Daten zu finden.

Stattdessen beginnt die Suche beim Wurzelknoten im binären Baum (auf der Grafik nicht angezeigt). Die Slots im Wurzelknoten enthalten einen Zeiger auf Kind – Knoten und die Storage – Engine folgt diesen Zeigern. Dabei wird der richtige Zeiger gefunden, indem die Werte in den Knoten – Seiten (engl. *node pages*) betrachtet werden, die die oberen und unteren Grenzen der Werte in den Kind – Knoten definieren. Letztendlich stellt die Storage Engine entweder fest, dass der gewünschte Wert nicht existiert, oder sie erreicht erfolgreich eine Blatt – Seite (engl. *leaf page*).

Blattseiten sind besonders, weil sie Zeiger auf die indexierten Daten enthalten, anstelle von Zeigern auf andere Seiten. Zwischen dem Wurzelknoten und den Blattseiten können viele Ebenen von Knoten-Seiten existieren. Die Tiefe des Baumes hängt von der Größe der Tabelle ab.

Außerdem speichern B-Bäume die indexierten Spalten in Reihenfolge, sodass sie für die Suche nach Datenbereichen nützlich sind. Als Beispiel nehmen wir das Durchlaufen eines Baumes mit einem Index auf einem Textfeld (type *varchar*), welches die Werte in alphabetischer Reihenfolge passiert. Eine Suche nach allen Namen, die mit „K“ beginnen ist daher effizient.

Der Index sortiert die Werte gemäß der Reihenfolge der Spalten, die in der CREATE TABLE – Anweisung angegeben wird, z.B. der Primärschlüssel (***last\_name, first\_name, b\_day***). B-Baum-Indizes funktionieren gut für Suchen mit dem vollständigen Schlüsselwert (engl. *full key value*), einem Schlüsselbereich (engl. *key range*) oder einem Schlüsselprefix (engl. *full key prefix*). Für den Schlüsselprefix gilt dies aber nur, wenn die Suche das linkeste Präfix des Indexes verwendet.

Als nächstes betrachten wir die möglichen Abfragen, bei denen B-Baum-Indizes besonders hilfreich sind, um ein Gefühl für den richtige Verwendung zu bekommen. Eine Übereinstimmung mit dem vollständigen Schlüsselwert gibt Werte für alle Spalten im Index an. Eine beispielhafte Abfrage wäre alle Übereinstimmungen für Max Mustermann, geboren am 2000-01-01, wenn der Schlüssel, wie oben beschrieben, aus Nachname, Vorname und Geburtsdatum bestehen würde. Für diesen Index sind auch Abfragen, die nur mit dem linken Präfix übereinstimmen hilfreich, indem bspw. nur die erste Spalte im Index verwendet (Beispiel von oben: Suche nach „Mustermann“). Als nächstes gibt es den Fall einer Übereinstimmung mit einem Spaltenpräfix, d.h. mit dem ersten Teil des Werts einer Spalte. In unserem Beispiel wären dies alle Nachnamen, die mit „M“ beginnen. Auch ist der Index effizient bei Übereinstimmung mit einem Bereich von Werten, z.B. Nachnamen, die zwischen „Mustermann“ und „Müller“. Das vorletzte Anwendungsbeispiel gibt es, wenn es eine Übereinstimmung einer genauen Spalte und eines Bereichs einer anderen Spalte gibt. Im Genaueren bedeutet dies, dass es für unseres Beispiel ein genaues Match mit dem Nachnamen „Mustermann“ geben muss und beispielhaft eine Bereichsfrage auf dem Vornamen, z.b. alle Vornamen, die mit „Ma“ beginnen. Der letzte Anwendungsfall sind Abfragen, die nur den Index verwenden, nicht aber die Zeilenspeicherung. Die ist beispielweise der Fall, wenn alle benötigten Daten im Index enthalten sind.

Ein weiterer Vorteil von B-Baum-Indizes ist, dass er, da der Baumknoten sortiert ist, sowohl Abfragen, also das Finden von Werten, als auch für ORDER BY – Abfragen verwendet werden kann. Das heißt, wenn ein B-Baum dabei helfen kann (siehe Kriterien oben) Zeilen auf eine bestimmte Weise zu finden, dann kann er auch dabei helfen, Zeilen nach denselben Kriterien zu sortieren.

Es gibt aber auch Einschränkungen von B-Baum-Indizes, weshalb wir weitere Indextypen untersuchen, die u.a. für diese Fälle geeignet sind. Die erste Limitation ist, wenn die Suche nicht vom linken Ende der indexierten Spalte beginnt. Beispielweise wollen wir alle Vornamen mit dem Wert Max oder alle Personen, die vor dem Jahr 2000 geboren worden sind. Für diese Fälle ist der Index bestehend aus Nachname, Vorname und Geburtstag nicht nützlich. Man muss nicht nur diese Reihenfolge der Spalten berücksichtigen, sondern man darf auch nicht einzelne Spalten im Index überspringen. Ein Beispiel, wo der Index nicht funktionieren würde, wäre eine Suche nach einem spezifischen Nachnamen und spezifischen Datum. Der Index würde nur dann funktionieren, wenn zusätzlich noch einem spezifischen Vornamen angegeben werden würde. Die Storage-Engine kann den Zugriff nicht optimieren, wenn Spalten rechts von der ersten Range – Bedingung betroffen sind. Beispielweise wenn nach dem Nachnamen „Mustermann“ gesucht wird, alle Vornamen, die mit „M“ beginnen und am die am „1985-02-05“ Geburtstag haben. In diesem Fall wird der Index nur die ersten beiden Spalten (Nachname und Vorname) verwenden, weil der LIKE – Operator eine Bereichsbedingung ist.

Für optimale Leistung sollten Indizes mit denselben Spalten in unterschiedlichen Reihenfolgen, um Ihre Abfragen zu optimieren. Wenn z. B. häufig Abfragen durchführen, bei denen der Nachname zuerst, dann der Vorname und dann das Geburtsdatum abgefragt wird, ist es sinnvoll, einen Index in dieser Reihenfolge zu erstellen. Dabei hilft es, sich einen Überblick von den am häufigsten verwendeten Abfragen zu verschaffen, um dann zu entscheiden, ob es sinnvoll wäre, weitere Indizes einzuführen oder nicht.

**Hash Index**

Den nächsten Indextypen, den wir betrachte, ist der Hash – Index. De Hash – Index baut auf einer Hash – Tabelle auf, weshalb er nur für exakte Suchen, die jede Spalte im Index verwenden, nützlich ist. Dabei geht die Storage – Engine wie folgt vor: Für jede Zeile wird ein Hash – Wert der indexierten Spalte anhand einer Hash – Funktion berechnet. Der Hash – Wert (engl. hash code) ist ein kleiner Wert, der sich wahrscheinlich von den Hash-Werten anderer Zeilen mit unterschiedlichen Schlüsselwerten unterscheidet.

In MySQL unterstützt nur die Memory – Storage – Engine explizite Hash – Indizes. Der eben besprochene Standard – Indextyp für Memory – Tabellen (B – Baum- Index) ist aber ebenfalls möglich. Des weiteren unterstützt die Memory – Engine nicht eindeutige Hash – Indizes. D.h. wenn mehrere Werte denselben Hash – Wert haben, dann speichert der Index ihre Zeiger auf die Zeilen (engl. row pointers) im gleichen Hash – Tabelleneintrag unter Verwendung einer verketteten Liste (bspw. einer LinkedList). Die Alternative zu den nicht eindeutigen Hash – Indizes sind Eindeutige, die sicherstellen, dass für jeden Hash – Wert nur ein einzelner Eintrag existiert. Falls ein Konflikt auftritt, wir ein Mechanismus wie eine *Open* Addressing-Strategie (z. B. Linear Probing oder Quadratic Probing) verwendet, um den Konflikt aufzulösen und den Speicherplatz effizient zu verwalten. Das heißt, dass bei eindeutigen Indizes Wert darauf gelegt wird, Konflikte direkt innerhalb der Hash – Tabelle zu bewältigen, anstatt mit zusätzlichen Datenstrukturen wie verketteten Listen zu arbeiten.

Um das Vorgehen bei der Berechnung der Hashfunktion noch einmal genauer aufzuzeigen, stellen ich den Ablauf nocheinmal mit einem Beispiel dar. Beispielquery: ***SELECT lname FROM testhash WHERE fname='Peter'.*** Zu Allererst berechnet MySQL den Hash-Wert von **'Peter'** und verwendet diesen um den Zeiger im Index nachzuschlagen. Bei der Hashfunktion f('Peter') kommt beispielweise der Wert 7654 heraus. Daraufhin schaut MySQL im Index bei der Position 7654 nach findet einen Zeiger auf Zeile 3. Der letzte Schritt ist, den Wert in Zeile 3 mit **'Peter'** zuvergleichen, um sicherzustellen, dass es die richtige Zahl ist. Die Indizes selbst speichern nur kurz Hash-Werte und daher sind Hash – Indizes sehr kompakt und Suchen sind blitzschnell.

Wie schon bei dem B – Baum – Index, hat auch der Hash – Index einige Einschränkungen, zu denen wir als nächstes gelangen. Da der Index nur die Hash – Werte und Zeiger auf Zeilen (engl. row pointers) enthält und die Werte selbst, kann MySQL die Werte im Index nicht nutzen, um das Lesen der Zeilen zu vermeiden. Da der Zugriff auf die in den Speicher geladenen Zeilen sehr schnell ist, wird die Leistung nicht beeinträchtigt.

Damit ist der erste richtige Nachteil, dass man Hash – Indizes nicht für Sortierungen verwenden kann, da sie nicht in einer richtigen Reihenfolge gespeichert sind. Dies war beim B – Baum – Index möglich, wenn man die Indizes richtig erstellt und auch verwendet. Auch werden anders als noch beim B- Baum – Index keine partiellen Schlüsselübereinstimmungen (engl. partial key matching), weil der Hash – Wert aus dem gesamten indexierten Wert berechnet wird. Wenn man sich als Beispiel einen Index bestehend aus (A,B) anschaut und sich die WHERE – Klausel nur auf A bezieht, wird der Hash – Index nicht helfen. Hash – Indizes sind aber noch weiter eingeschränkt, da sie keine Bereichsabfragen (range queries) beschleunigen. Sie unterstützen nur Gleichheitsvergleiche, zu denen die Operatoren gleich (**'='**), ungleich **'<=>'** und IN() gehören.

Der Zugriff auf Daten ist zwar bei einem Hash – Index eingeschränkter, aber es ist dafür auch performanter. Hash – Indexes verlieren aber an Performance, wenn es zu vielen Hash – Kollisionen kommt, da viele Werte denselben Hash – Wert haben. In diesen Fällen muss die Storage – Engine jeden Zeiger auf die Zeile in der verketteten Liste folgen und deren Werte mit dem gesuchten Wert vergleichen, um die richtige(n) Zeile(n) zu finden.

Einige Index – Wartungsoperationen können dadurch auch langsam sein, wenn es viele Hash – Kollisionen gibt. Zum Beispiel, wenn ein Index auf einer Spalte mit sehr geringer Selektivität erstellt wird und anschließend eine Zeile löscht, kann das Finden des Zeigers im Index sehr teuer sein und damit auch das eigentliche Löschen der Zeile.

All diese Einschränkungen machen Hash-Indizes nur in speziellen Fällen nützlich, aber wenn sie den Bedürfnissen der Anwendung entsprechen, können sie die Leistung dramatisch verbessern. Ein solches Beispiel wäre in einem Data Warehouse (DWH) mit dem Stern – Schema, da es dort sehr viele kleine Tabellen gibt und diese bei Abfragen unter Umständen alle aneinander gejoint werden müssen, würde ein sehr schneller Lockup die teure Operation des Joinens deutlich beschleunigen.

Einige Speicher-Engines (wie InnoDB) können feststellen, dass bestimmte Index-Werte sehr häufig verwendet werden, und bauen automatisch einen Hash-Index für diese Werte im Speicher (memory) auf, der zusätzlich zu den B-Tree-Indizes verwendet wird.

TODO heute: building your own hash, handling hash collsions, spatial indexes, full text indexes, other types of index, benefits of indexes, is an index the best solution