**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.

**Eigene Hash-Indexes erstellen**

Man kann den Hash – Index auch selbst nachstellen, um Zugang zu einigen der gewünschten Eigenschaften von Hash-Indizes zu erlangen. Beispielsweise eine sehr kleine Indexgröße für sehr lange Schlüssel. Dabei muss man nach dem folgenden Schema arbeiten. Man erstellt einen Pseudo-Hash-Index auf einem Standard B-Tree-Index. Der B-Tree-Index wird weiterhin für Suchen (lockups) verwendet, aber es werden die Hash-Werte der Schlüssel (hash-values) anstell der Schlüssel selbst verwendet. Dazu wird die Hash-Funktion manuell in der WHERE-Klausel angegeben. Diese Verwendung funktioniert gut für URL-Suchen, weil der MySQL-Abfrageoptimierer (query optimizer) bemerkt, dass es einen kleinen, hochselektiven Index auf der ur\_crc – Spalte gibt. Dies ist schneller als die Alternative, die vollständige URL als Zeichenkette zu indexieren. Dafür besteht die Notwendigkeit, die Hash-Werte zu pflegen, weshalb man neben der Erstellung der Tabelle mit der url- und der url\_crc-Spalte auch einen Trigger für Insert oder Update führen muss. Der hauptsächliche Vorteil bei der eigenen Erstellung ist, dass man die Hashfunktion selbst bestimmen kann. Um möglichst wenig Speicherplatz zu verschwenden, bieten es sich an keine komplexen Hash-Funktionen mit langen Zeichenketten als Rückgabe zu verwenden. Einfache Hash-Funktionen bieten akzeptablere Kollisionsraten mit besseren Leistungen.

**TODO(Daniel): Erklärung der Grafiken + Analyse**

**Umgang mit Hash-Kollisionen**

Wenn man nach einem Wert durch seinen Hash suchst, musst du auch den literalen Wert in der WHERE-Klausel angeben. Andernfalls würde es mehr als einen Wert zurückgeben, wenn es zu einer Hashkollision kommt. Dabei wachsen Hash-Kollisionen schneller als man zunächst annehmen würde. Zur Veranschaulichung kann man das sogenannte Geburtstagsparadoxon. Wenn sich in einem Raum 23 Personen befinden, dann ist die Wahrscheinlichkeit dafür, dass zwei am gleichen Tag (ohne Berücksichtigung des Jahres) Geburtstag haben bei über 50%. Bei 30 Personen liegt die Wahrscheinlichkeit schon bei über 70%, bei 40 Personen bei über 89% und bei 50 Personen bei 97%. Die Hashfunktion CRC32() gibt einen 32-Bit-Wert zurück, sodass die Wahrscheinlichkeit einer Kollision bei bereits 93.000 Werten bei 1% liegt. Wenn Kollisionen kein Problem darstellen, z.B. aufgrund von statistischen Abfragen, kann man es weiter vereinfachen und etwas Effizienz gewinnen, indem du weniger Bit-große Ganzzahlen verwendest.

**Volltext-Indizes**

Dies ist ein spezieller Indextyp, der Schlüsselwörter im Text findet, anstatt Werte direkt mit den Werten im Index zu vergleichen. Volltextsuche (full-text-searching) ist völlig anders als andere Arten von Übereinstimmungen und hat viele Feinheiten, wie Stoppwörter, Stemming und Pluralformen, sowie Boolesche Suche. Samit ist es viel mehr analog zu dem, was eine Suchmaschine (search engine) tut, als zu einer einfacher WHERE-Parameterübereinstimmung. Einen Volltext-Index auf einer Spalte zu haben, beseitigt nicht den Wert eines B-Tree-Indexes auf der gleichen Spalte. Volltext-Indizes sind für MATCH AGAINST-Operationen, nicht für gewöhnliche WHERE-Klausel-Operationen

**Weitere Indizes**

Spatial Index und Clustered Indexes und Covering Indexes

**Vorteile von Indizes / Zusammenfassung**

Indizes ermöglichen es dem Server, schnell zu einer gewünschten Position in der Tabelle zu navigieren. Zudem haben sie mehrere Vorteile, basierend auf den Eigenschaften der Datenstrukturen, die zu ihrer Erstellung verwendet werden. Für B-Tree-Indizes kann MySQL die sortierten Daten für Abfragen mit Klauseln wie ORDER BY oder GROUP BY ausnutzen. Da die Daten vorsortiert sind, speichert ein B-Tree-Index auch verwandte Werte nah beieinander. Der Index speichert tatsächlich eine Kopie der Werte, sodass einige Abfragen direkt aus dem Index befriedigt werden können. Dies hat 3 Hauptvorteile: Indizes reduzieren die Menge der Daten, die der Server untersuchen muss. Außerdem helfen Indizes dem Server, Sortierungen und temporäre Tabellen zu vermeiden. Indizes verwandeln zufällige I/O in sequenzielle E/A.

**Index die beste Lösung**

Ein Index ist nicht immer das richtige Werkzeug. Index sind am effektivsten, wenn sie bei der Speichermaschine helfen, Zeilen zu finden, ohne mehr Arbeit hinzuzufügen, als sie zu vermeiden. Für kleine Tabellen ist es oft effektiver, einfach alle Zeilen in der Tabelle zu lesen. Für mittlere bis große Tabellen können Indizes sehr effektive sein. Für enorm große Tabellen kann der Overhead der Indizierung sowie die Arbeit, die erfordert ist, um die Indizes tatsächlich zu verwenden, anfangen, sich zu summieren. In diesem Fall muss eine Technik gewählt werden, die Gruppen von Zeilen identifiziert (Partitionierung). Für viele Tabellen kann es sinnvoll sein, eine Metadaten-Tabelle zu erstellen, um einige Merkmale von Interesse für deine Tabelle zu speichern (Partitionierung in mehrere Tabellen).

**Indexierungsstrategien für hohe Leistungen**

MySQL kann in der Regel keine Indizes auf Spalten verwenden, es sei denn, die Spalten sind in der Abfrage isoliert. Deshalb ist es eine Strategie die „Indexierung der Spalte“. Was bedeutet, dass die Spalte nicht Teil eines Ausdrucks oder einer Funktion in der Abfrage sein sollte, nicht wie folgt: WHERE actor\_id + 1= 5. Für den Menschen ist diese Anweisung leicht zu vereinfachen, aber MySQL kann es nicht. Deshalb sollte man die WHERE-Klauseln so vereinfachen, sodass die indexierte Spalte allein auf einer Seite des Vergleichsoperators steht. Manchmal müssen sehr lange Zeichenketten-Spalten indexiert wurden, wodurch die Indizes groß und langsam werden. Eine Strategie bietet der Hash-Index, wie bereits erwähnt. Oft kann aber auch Speicherplatz gespart werden und eine gute Leistung erzielt werden, indem nur die ersten Zeichen statt des gesamten Wertes zu indexieren. Dafür wird dann weniger Speicherplatz gebraucht, aber dafür verliert man Selektivität. Die Indexselektivität beschriebt das Verhältnis der Anzahl der verschiedenen indexierten Werten (die Kardinalität) zur Gesamtzahl der Zeilen in der Tabelle (#T) und reicht von 1/#T bis 1. Ein hoher selektiver Index ist gut, weil er MySQL ermöglicht, mehr Zeilen herauszufiltern, wenn es nach Übereinstimmungen sucht. Ein einzigartiger Index hat eine Selektivität von 1. Ein Präfix der Spalten ist oft ausreichend selektiv, um eine gute Leistung zu erzielen. Für BLOB, TEXT oder sehr lange VARCHAR-Spalten musst du Präfix-Indizes definieren. Wähle ein Präfix, das lang genug ist, um eine gute Selektivität zu bieten, aber kurz genug ist, um Speicherplatz zu sparen. Das Ziel dabei ist es, dass die Kardinalität des Präfixes nahe der Kardinalität der gesamten Spalte ist. MySQL kann Präfix – Indizes nicht für ORDER BY oder GROUP BY – Abfragen verwenden und auch nicht als Covering – Indizes. Lange hexadezimale Bezeichner profitieren besonders von Präfix-Indizes. Manchmal machen auch Suffix-Indexes Sinn (z.B. für das Finden aller E-Mail-Adressen aus einer bestimmten Domain). MySQL unterstützt sie nicht nativ, aber du kannst einen umgekehrten String speichern und ein Präfix davon indexieren und zusätzlich noch ein Trigger verwenden, um den Index zu pflegen.

**Multicoloumn Indexes**

Das Buch von Lahdenmaki und Leach führt ein Drei-Sterne-System ein, um die Eignung eines Indexes für eine Abfrage zu bewerten. Den ersten Stern erhält ein Index, wenn er relevante Zeilen nebeneinander platzt. Einen zweiten Stern wird dafür vergeben, wenn alle Zeilen in der Reihenfolge sortiert sind, wie sie die Abfrage benötigt. Und der letzte Stern wird dafür vergeben, wenn alle Spalten im Index enthalten sind, die für die Abfrage benötigt werden. Häufige Fehler, die dabei auftreten, sind viele oder alle Spalten separat zu indexieren oder Spalten in der falschen Reihenfolge zu indexieren. Zu viele separat indexierte Spalten führen höchstens zu Ein-Sterne-Indizes. Unter anderem kann es auch vorkommen, dass man keinen Drei-Sterne-Index entwerfen kann. In diesen Fällen ist es besser, die WHERE-Klausel zu ignorieren und auf die optimale Reihenfolge der Zeilen zu achten (oder stattdessen einen Covering-Index zu erstellen => TODO(Daniel): Recherche dazu). Wenn man Tabellen schlecht indiziert, dann kann man ein solches Problem ein wenig umgehen. MySQL verwendet eine Strategie namens Index-Merge, die es einer Abfrage ermöglicht, dass mehrere Indizes aus einer Tabelle kombinieren werden, um die gewünschten Zeilen zu finden.

Als nächstes betrachten wir dafür ein Beispiel. Wir haben eine Tabelle film\_actor mit einem Index auf film\_id und einen auf actor\_id. Bei einer Abfrage, bei der beide Attribute in der WHERE-Klausel auftreten, wirken die Indizes nicht. In älteren Versionen von MySQL führt dies entweder zu einem Tabellen-Scan führen, bei dem beide Indexe nicht genutzt werden oder man umschreibt die Anfrage als UNION zweier Abfragen, die beide jeweils die Indexe benutzten. Seit MySQL 5.0 kann die Abfrage beide Indizes nutzen, indem sie gleichzeitig scannt und die Ergebnisse zusammenführt. Mit EXPLAIN PLAN kann man den genauen Ausführungsplan anzeigen lassen. Wenn die Indizes UNION verwenden, dann sieht man dies unter dem Attribut „extra“. Wenn MySQL die Queries optimiert, bedeutet das normalerweise, dass man anstelle von einzelnen Indizes für jeweils nur eine Spalte, einen einzelnen Index, bzw. weniger Indizes mit allen für die meisten Abfragen relevanten Spalten benötigt. Außerdem kann diese Merge-Strategie Auswirkungen auf die Performance haben, wie z.B. das Puffer-, Sortier- und Zusammenführungsoperationen des Algorithmus viele CPU -und Speicherressourcen belasten. Der Query Optimizer berücksichtigt diese Kosten nicht, weshalb es dazu führen kann, dass die Abfrage schneller eingeschätzt wird als ein Full-Table-Scan oder eine umschriebene UNION-Abfrage. Es gibt auch Möglichkeiten die Index-Merges zu deaktivieren. Am besten ist es allerdings, wenn man die Anwendungszwecke analysiert und die konkreten Indizes anpasst, wodurch die Index-Merge-Strategie nicht zugreifen muss, da der Index schon korrekt benutzt werden kann.

**Bestimmung einer guten Spaltenreihenfolge**

Die Reihenfolge der Spalten in den Indizes ist eine der häufigsten Verwirrungsursachen. Dabei ist die korrekte Reihenfolge abhängig von den Abfragen, die den Index verwenden. Deshalb muss die Indexreihenfolge so gewählt werden, dass die Zeilen in einer Weise sortiert und gruppiert werden, die der Abfrage zugutekommt. Dies gilt nur für B-Tree-Indizes, da Hash- und andere Indexarten ihre Daten nicht in einer sortierten Reihenfolge speichern. Ein Multicolumn-B-Tree-Index bedeutet, dass der Index zuerst nach der linksstehenden Spalte sortiert wird. Der Index kann entweder vorwärts oder rückwärts gescannt werden, um Abfragen wie ORDER BY zu erfüllen. Die Spaltenreihenfolge ermöglicht oder verhindert, dass der Index „Sterne“ im Drei-Sterne-System von Lahdenmaki und Leach erhält. Um die möglichst optimale Spaltenreihenfolge zu erhalten, muss man die selektivste Spalte zuerst im Index sein. Dies ist aber nur der Fall, wenn keine Sortierung oder Gruppierung berücksichtigt werden soll und der Zweck nur die Optimierung von WHERE-Lockups ist. Hängt auch von den tatsächlichen Werten ab, mit denen die Zeilen gesucht werden, weshalb die Spaltenreihenfolge möglichst so gewählt sein sollte, dass sie für die Abfragen, die am häufigsten ausgeführt werden, so selektiv wie möglich ist.

**Clustered Index P.168\_07**

Geclusterte Indizes sind kein separater Indextyp, sondern eine Methode der Datenspeicherung. InnoDB’s geclusterte Indizes speichern einen B-Tree-Index und die Zeilen zusammen in derselben Struktur. Die Bezeichnung Geclusterte bezieht dabei darauf, dass Zielen mit benachbarten Schlüsselwerten nahe beieinander gespeichert werden. Wenn eine Tabelle einen geclusterten Index hat, werden die Zeilen in den Blattseiten des Index gespeichert. Damit kann jede Tabelle maximal nur einen geclusterten Index haben, da keine Zeilen an zwei Orten gleichzeitig gespeichert werden können. Covering-Indizes ermöglichen es, mehrere geclusterte Indizes zu emulieren. Dies wird aber nicht von allen Storage Engines unterstützt. Die Blattseiten enthalten vollständige Zeilen, während die Knoten-Seiten nur die indexierten Spalten erhalten. Einige Datenbank-Server ermöglichen die Auswahl des Indexes, nach dem die Tabelle geclustert wird und andere verwenden einfach den Primärschlüssel.

Das Clustern von Primärschlüsseln kann die Performance verbessern, es kann jedoch auch zu ernsthaften Performance-Problemen führen. Die Vorteile des Clusterns von Daten ist, dass verwandte Daten zusammengehalten werden und der schnellere Datenzugriff im Vergleich zu nicht geclusterten Indizes. Außerdem können Abfragen, die Covering-Indexes verwenden, die PK-Werte verwenden, die im Blattknoten vorhanden sind.

Als nächstes kommen wir zu den Nachteilen. Clustering bietet die größten Veränderungen für I/O-gebundene Workloads, wenn aber die Daten in den Speicher passen, dann spielt die Reihenfolge keine Rolle mehr und damit gibt es auch keine Vorteile. An sich hängt die Einfügegeschwindigkeit stark von der Einfüge-Reihenfolge ab. Der schnellste Weg für das Einfügen von Zeilen ist in PK-Reihenfolge. Es könnte auch eine gute Idee sein, die Tabelle nach dem Laden großer Datenmengen mit OPTIMIZE TABLE neu zu organisieren, wenn die Zeilen nicht in PK-Reihenfolge geladen wurden. Es darf dabei aber nicht vergessen werden, dass Aktualisierungen des geclusterten Index sehr teuer sind, da sie dazu zwingen, jede aktualisierte Zeile an einen neuen Speicherort zu verschieben. Tabellen sind anfällig für sogenannte Page Splits, wenn neue Zeilen eingefügt werden. In solchen Fällen muss eine Seite in zwei Teile aufgeteilt werden, um die neue Zeile unterzubringen. Dies kann dazu führen, dass die Tabelle mehr Speicherplatz auf der Festplatte benötigt. Clustered-Tabellen können bei Full-Table-Scans langsamer sein, insbesondere dann, wenn die Zeilen aufgrund von Page Splits weniger dicht gepackt oder nicht mehr in sequenzieller Reihenfolge gespeichert sind. Sekundäre, also non-clustered, Indizes können größer sein als erwartet, da ihre Blattknoten die Primärschlüsselspalten der referenzierten Zeilen enthalten. Der Zugriff auf sekundäre Indizes erfordert zudem zwei Index-Lookups anstelle von einem. Das liegt daran, dass ein Blattknoten keinen direkten Zeilen-Pointer auf den physischen Standort der referenzierten Zeile speichert, sondern stattdessen die Primärschlüsselwerte der Zeile enthält.

**Vergleich des Layouts von InnoDB (clustered) und MyISAM (nonclustered)**

Angenommen, die Tabelle ist mit Primärschlüsselwerten von 1 bis 10.000 populiert. Die Werte wurden dabei zufällig eingefügt und dann mit OPTIMIZE TABLE optimiert. Als Resultat sind die Daten optimal auf der Festplatte arrangiert, aber die Zeilen könnten in zufälliger Reihenfolge vorliegen und die Werte für col2 werden zufällig zwischen 1 und 100 zugewiesen (mit vielen Duplikaten).

MyISAM speichert die Zeilen auf der Festplatte in der Reihenfolge, in der sie eingefügt wurden. Die Zeilen sind dabei in fester Größe und MyISAM kann jede Zeile finden, indem es die erforderliche Anzahl an Bytes vom Anfang der Tabelle aus ansteuert. Jedes Blatt im Index kann einfach die Zeilennummer der Layout-Tabelle enthalten. Col2 ist einfach ein Index wie jeder andere und es gibt keinen strukturelleren Unterschied zwischen einem Primärschlüssel oder einem anderen Index. Der PK ist einfach ein einzigartiger, nicht nullbarer Index mit dem Namen PRIMARY.

InnoDB speichert dieselben Daten aufgrund seiner clustered Organization sehr unterschiedlich. Der clustered index „ist“ die Tabelle in InnoDB und es gibt keine separate Zeilenspeicherung wie bei MyISAM. Jeder Blattknoten im clustered index enthält den Primärschlüsselwert, die Transaktions-ID und einen Rollback-Pointer, den InnoDB für transaktionale und MVCC-Zwecke verwendet, sowie die restlichen Spalten (in diesem Fall col2). Anstatt „Zeilen-Pointer“ zu speichern, enthalten die Blattknoten des sekundären Index in InnoDB die Primärschlüsselwerte, die als „Pointer“ zu den Zeilen dienen. Dies reduziert den Arbeitsaufwand zur Pflege sekundärer Indizes, aber macht den Index größer. InnoDB’s non-leaf B-Tree-Knoten enthalten die indexierten Spalten sowie einen Pointer zum nächsten tieferen Knoten (der entweder ein weiterer non-leaf Knoten oder ein Blattknoten sein kann)

**Wenn die Reihenfolge des Primärschlüssels problematisch ist**

Bei hohe Parallelitätslasten (high-concurrency workloads) kann das Einfügen von Daten in Primärschlüsselreihenfolge zu Konfliktpunkten in InnoDB führen. Das obere Ende des Primärschlüssels wird zu einem Hotspot, weil alle Einfügungen dort stattfinden, was dazu führt, dass gleichzeitige Einfügungen um den next-key Lock konkurrieren. Ein weiteres Problem ist der AUTO\_INCREMENT-Sperrmechanismus, der die Konkurrenz bei Multi-Thread-Szenarien weiter verstärken kann. Lösungen dieser Probleme könnten entweder das Neu-Design der Tabelle oder Anwendung sein, um Einfügungen gleichmäßiger zu verteilen, oder die Konfiguration der *innodb\_autoinc\_lock\_mode*-Einstellung, die steuert, wie AUTO\_INCREMENT-Werte während des Einfügens gesperrt werden.

**Redundante und doppelte Indizes**

MySQL erlaubt das Erstellen mehrerer Indizes auf demselben Satz von Spalten, was jedoch ineffizient sein kann, denn MySQL müsste jeden doppelten Index separat verwalten, was zu zusätzlichem Overhead führen kann. Der Abfrageoptimierer (query optimizer) berücksichtigt jeden Index unabhängig bei der Abfrageoptimierung, was zu möglichen Leistungseinbußen führen kann. Das bedeutet, dass doppelte Indizes nicht automatisch vor Fehlern schützen und die Leistung nicht verbessern, sondern unnötige Arbeit verursachen. Deshalb gilt es als Standard, dass man das Erstellen doppelter Indizes vermeidet und falls doppelte Indizes bereits existieren, sollten diese entfernt werden, um die Leistung zu verbessern und unnötigen Overhead zu reduzieren.

**Zusammenfassung weiterer Themen**

**Auswahl von Identifikatoren (Choosing identifiers)**

Die Wahl eines geeigneten Datentyps für eine Identifikatorspalte ist entscheidend, da diese Spalten häufig für Vergleiche, Lookups oder als Fremdschlüssel in anderen Tabellen genutzt werden. Dabei sollten nicht nur der Speichertyp, sondern auch die Art und Weise, wie MySQL Berechnungen und Vergleiche durchführt, berücksichtigt werden. Beispielsweise werden ENUM- und SET-Typen intern als Ganzzahlen gespeichert, jedoch in Zeichenketten umgewandelt, wenn sie im String-Kontext verglichen werden.

Es ist wichtig, dass in allen zugehörigen Tabellen identische Datentypen verwendet werden, einschließlich Eigenschaften wie UNSIGNED, um Leistungsprobleme oder schwer zu findende Fehler zu vermeiden. Generell sollte der kleinste Typ gewählt werden, der den benötigten Wertebereich abdeckt, mit ausreichend Spielraum für zukünftiges Wachstum. INTEGER ist meist die beste Wahl für Identifikatoren, da er schnell ist und gut mit AUTO\_INCREMENT arbeitet. ENUM-, SET- und String-Typen sollten wegen ihrer Performance-Nachteile und ihres höheren Speicherbedarfs vermieden werden.

Schlecht geschriebene Schema-Migrationsprogramme und automatisch generierte Schemata können erhebliche Leistungsprobleme verursachen. Anwendungen, die Komplexität vor Entwicklern verbergen, skalieren oft schlecht. Daher sollte die Abwägung zwischen Leistung und Entwicklerproduktivität sorgfältig erfolgen, begleitet von Tests mit realistisch großen Datensätzen, um spätere Probleme zu vermeiden.

Zu den spezifischen MySQL-Problemen zählen eine zu hohe Anzahl von Spalten oder Joins – als Faustregel gilt, dass eine Abfrage nicht mehr als ein Dutzend Tabellen umfassen sollte, um hohe Parallelität und Geschwindigkeit zu gewährleisten. Auch die falsche Verwendung von ENUM-Typen oder die Erfindung von Ersatzmethoden für NULL (z. B. '0000-00-00 00:00:00') sollte vermieden werden. Nutzen Sie NULL, um unbekannte Werte korrekt darzustellen.

**Normalisierung und Denormalisierung**

Es gibt in der Regel viele unterschiedliche Ansätze, um Daten in einer Datenbank zu repräsentieren. Diese reichen von vollständig normalisierten bis hin zu vollständig denormalisierten Strukturen und zahlreichen Mischformen dazwischen. In einer normalisierten Datenbank wird jede Tatsache genau einmal gespeichert, wodurch Redundanzen vermieden werden. Im Gegensatz dazu speichern denormalisierte Datenbanken Informationen mehrfach oder an verschiedenen Orten. Besonders bei schreibintensiven Workloads wird empfohlen, Datenbanken zu normalisieren, da dies oft Leistungsprobleme löst.

Die Normalisierung bietet mehrere Vorteile. Normalisierte Updates sind in der Regel schneller, da weniger redundante Daten geändert werden müssen. Zudem sind normalisierte Tabellen häufig kompakter, wodurch sie besser in den Speicher passen und so die Gesamtleistung erhöhen. Ein weiterer Vorteil ist, dass das Fehlen redundanter Daten den Bedarf an komplexen Abfragen wie DISTINCT oder GROUP BY reduziert, wenn eindeutige Wertelisten abgerufen werden. Ein Beispiel hierfür ist die Erstellung einer Liste von Abteilungen: Während dies in einem denormalisierten Schema nur mit DISTINCT oder GROUP BY möglich ist, ist es in einem normalisierten Schema, in dem Abteilungen in einer separaten Tabelle gespeichert sind, eine triviale Abfrage. Allerdings erfordern viele nicht triviale Abfragen in einem normalisierten Schema mindestens einen Join, oft sogar mehrere. Diese Joins können nicht nur teuer in der Ausführung sein, sondern manchmal auch die Nutzung bestimmter Index-Strategien verhindern, da Normalisierung Spalten in verschiedene Tabellen aufteilt.

Denormalisierte Tabellen hingegen können die Leistung verbessern, da sie weniger Lookups und Joins benötigen. Vollständig normalisierte oder denormalisierte Schemata sind jedoch selten realitätsnah und gleichen eher theoretischen Modellen. In der Praxis ist oft ein Mischansatz erforderlich, der Elemente von Normalisierung und Denormalisierung kombiniert, beispielsweise durch den Einsatz von Cache- oder Zusammenfassungstabellen.

Eine Möglichkeit, die Leistung zu verbessern, besteht darin, redundante Daten direkt in der Tabelle zu speichern, aus der sie abgeleitet wurden. Alternativ können Cache-Tabellen verwendet werden. Diese Tabellen enthalten Daten, die aus dem Hauptschema zwar leicht, aber langsamer abgerufen werden können, und dienen dazu, Such- und Abrufabfragen zu optimieren. Cache-Tabellen können jedoch zahlreiche Index-Kombinationen erfordern, um verschiedene Abfragtypen zu beschleunigen. In manchen Fällen enthalten sie nur eine Teilmenge der Spalten der Haupttabelle, um Konflikte zwischen Anforderungen zu lösen.

Zusammenfassungstabellen bieten eine weitere Lösung. Sie speichern aggregierte Daten, die typischerweise aus GROUP BY-Abfragen stammen. Ein Beispiel hierfür ist die Zählung der in den letzten 24 Stunden geposteten Nachrichten. Eine Zusammenfassungstabelle, die stündlich aktualisiert wird, kann solche Informationen nahezu in Echtzeit bereitstellen, ohne dass teure Berechnungen erforderlich sind. Da Zusammenfassungs- und Cache-Tabellen oft während des Neuaufbaus weiterhin Daten verfügbar halten müssen, bietet sich die Nutzung von sogenannten Schatten-Tabellen an. Diese werden im Hintergrund aktualisiert und ersetzen anschließend die Haupttabelle, um Unterbrechungen zu vermeiden.

Ein gut ausbalanciertes Schema, das Normalisierung und Denormalisierung kombiniert und durch Cache- sowie Zusammenfassungstabellen ergänzt wird, ermöglicht es, sowohl Datenintegrität als auch Leistung zu optimieren.

**Materialisierte Sichten (Materialized Views)**

Viele Datenbankmanagementsysteme bieten die Funktionalität von materialisierten Sichten, jedoch unterstützt MySQL diese nicht direkt. Materialisierte Sichten sind spezielle Sichten, die nicht nur definiert werden, sondern tatsächlich vorab berechnet und als Tabellen auf der Festplatte gespeichert werden. Diese Sichten können regelmäßig aktualisiert oder aufgefrischt werden, je nach den gewählten Strategien. Eine besonders effiziente Implementierung dieser Funktion ist FlexViews, eine spezielle Lösung eines Entwicklers, die Deltas verwendet. Deltas repräsentieren die Unterschiede zwischen den vorherigen und den aktuellen Daten, was deutlich effizienter ist, als jedes Mal die Daten aus der Quelltabelle neu zu lesen.

**Zähltabellen (Counter Tables)**

Eine Anwendung, die Zählwerte in einer Tabelle speichert, kann bei der Aktualisierung der Zähler auf Nebenläufigkeitsprobleme stoßen. Um diese zu vermeiden, kann die Verwendung einer separaten Tabelle helfen, Abfrage-Cache-Invalideierungen zu verhindern und den Einsatz fortgeschrittenerer Techniken zu ermöglichen. Eine bewährte Methode zur Verbesserung der Nebenläufigkeit ist die Verwendung einer zufälligen Slot-Variablen, die zusammen mit dem Tag als Primärschlüssel genutzt wird, um den Zähler zu aktualisieren, auch bei der Verwendung von „ON DUPLICATE KEY UPDATE“. Zusätzlich kann ein periodischer Job geschrieben werden, der alle Ergebnisse im Slot 0 zusammenführt und alle anderen Slots löscht, um die Daten zu konsolidieren und die Effizienz zu verbessern.

**Schnellere Leseoperationen, langsamere Schreiboperationen (Faster Reads, Slower Writes)**

Oft sind zusätzliche Indizes, redundante Felder oder sogar Cache- und Zusammenfassungstabellen erforderlich, um Leseabfragen zu beschleunigen. Allerdings erhöhen diese Maßnahmen auch die Arbeitsbelastung für Schreibabfragen und Wartungsaufgaben. Um schnellere Leseabfragen zu ermöglichen, steigt die Entwicklungs-Komplexität sowohl für Lese- als auch für Schreiboperationen. Trotz dieser zusätzlichen Herausforderungen ist diese Technik jedoch häufig notwendig, wenn eine Hochleistungs-Datenbank entworfen wird.

**Beschleunigung von ALTER TABLE (Speeding Up ALTER TABLE)**

Die Leistung von ALTER TABLE kann bei sehr großen Tabellen problematisch sein. MySQL führt Änderungen in der Regel wie folgt aus: Es wird eine neue leere Tabelle mit der gewünschten Struktur erstellt, anschließend werden alle Daten aus der alten Tabelle in die neue kopiert und schließlich wird die alte Tabelle gelöscht. Dieser Prozess kann besonders zeitaufwendig sein, wenn der Arbeitsspeicher begrenzt ist, die Tabelle groß ist oder viele Indizes enthält. In einigen Fällen kann dies mehrere Tage dauern. Ab MySQL 5.1 gibt es Unterstützung für einige „Online“-Operationen, bei denen die Tabelle während des Vorgangs nicht gesperrt wird. Für den allgemeinen Fall empfiehlt es sich, operationelle Tricks zu verwenden, wie das Umstellen von Servern oder das Ausführen von ALTER TABLE auf einem Nicht-Produktionsserver. Eine andere Technik ist die „Schattenkopie“-Methode: Dabei wird eine neue Tabelle mit der gewünschten Struktur neben der bestehenden erstellt, und anschließend wird durch Umbenennen und Löschen die alte Tabelle durch die neue ersetzt. Eine der teuersten Methoden, um den Standardwert einer Spalte zu ändern, ist das Kommando MODIFY COLUMN rental\_duration TINYINT(3) NOT NULL DEFAULT 5. Eine kürzere und einfachere Methode ist ALTER COLUMN rental\_duration SET DEFAULT 5, da der Standardwert für die Spalte in der .frm-Datei der Tabelle gespeichert wird, sodass er geändert werden kann, ohne die gesamte Tabelle zu verändern. Weitere mögliche Operationen ohne Tabellen-Neuaufbau sind das Entfernen (aber nicht Hinzufügen) des AUTO\_INCREMENT-Attributs einer Spalte sowie das Hinzufügen, Entfernen oder Ändern von ENUM- und SET-Konstanten. Eine Technik zur Umsetzung lautet wie folgt: Erstellen Sie eine leere Tabelle mit der gleichen Struktur, jedoch mit der gewünschten Änderung, und führen Sie dann FLUSH TABLES WITH READ LOCK aus, um alle Tabellen zu schließen und den Zugriff zu blockieren. Danach tauschen Sie die .frm-Dateien aus und führen UNLOCK TABLES aus, um die Lesesperre zu lösen.