**Index**

Ein Index für ein bestimmtes Attribut A einer Tabelle ist eine Datenstruktur, die die Suche nach Tupeln mit einem bestimmen Wert für Attribut A effizienter macht.

Wenn Tabellen immer mehr Datensätze enthalten, wird es umso schwieriger alle Tupel zu scannen, um (nur wenige) Tupel zu finden, die zu einer Bedingung passen. Besonders nützlich sind Indizes bei Abfragen, die Joins zwischen mehreren Tabellen enthalten, da sie ermöglichen, die Anzahl der zu prüfenden Tupel erheblich zu reduzieren, wenn eine einschränkende Bedingung vorliegt.

Datenbankmanagementsysteme ermöglichen das Erstellen von Indizes über mehrere Attribute. Die Reihenfolge der Attribute ist hierbei entscheidend, da der Index nur effizient genutzt werden kann, wenn das erste Attribut der Reihenfolge in der Abfrage angegeben wird. Gibt man nur das zweite Attribut an, ohne das erste zu referenzieren, kann der Index nicht direkt verwendet werden.

Das Auswählen von Indizes erfordert von den Entwicklern eine Tradeoff abzuwägen. Es gibt dabei zwei Faktoren, die die Entscheidung beeinflussen. Zum einen kann ein Index auf einem Attribut Abfragen mit diesem Attribut erheblich beschleunigen. Zum anderen erschwert jeder Index Einfügungen, Löschungen und Aktualisierungen, da diese mehr Zeit und Aufwand erfordern.

Um zu verstehen, wie man Indizes für eine Datenbank auswählt, muss man wissen, wo bei der Bearbeitung einer Abfrage die meiste Zeit verbraucht wird. Die Tupel einer Relation sind üblicherweise auf viele Seiten einer Festplatte verteilt, wobei eine Seite mehrere Tausend Bytes umfasst und viele Tupel speichert. Um ein Tupel zu prüfen, muss die gesamte Seite, bzw. Block, in den Hauptspeicher geladen werden, wobei es kaum mehr Zeit kostet, alle Tupel einer Seite statt nur ein einzelnes zu prüfen.

Meistens ist der sinnvollste Index, den wir in einer Tabelle definieren können, der Schlüssel. Das liegt zum einen daran, dass Abfragen auf diese Spalte sehr häufig vorkommen und zum anderen gibt es beim Schlüssel keine doppelten Werte, weshalb der Index entweder nichts oder genau eine Referenz auf ein Tupel zurückgibt.

Es gibt aber auch zwei Situationen, bei denen es hilfreich sein kann, einen Index auf Nichtschlüsselattribute zu definieren. Zum einen wenn das Attribut ähnlich wie ein Schlüssel. Damit ist gemeint, dass es für einen bestimmten Wert nicht genau ein Tupel gibt, wie bei einem Schlüssel, aber nur eine geringe Anzahl an Tupeln und damit werden auch weniger Festplattenzugriff für einen bestimmten Wert des Attributs benötigt wird. Selbst wenn die Werte auf unterschiedlichen Blöcken gespeichert sind, sparen wir uns immer noch Zeit mit Index, wenn durch ihn nicht alle Blöcke geladen werden müssen. Der andere Fall, bei dem ein Index sinnvoll sein könnte, wäre wenn die Tupel nach dem Attribut geclustert sind. Das bedeutet, dass aufeinanderfolgende Ausprägungen des Attributs bei einem erstellten Index dazu führen, dass weniger Datenblöcke auf die Festplatte geladen werden müssen.

Jede Modifikation an einer Relation R zwingt uns dazu, jeden Index auf einem oder mehreren der modifizierten Attribute von R zu ändern. Daher müssen wir nicht nur die Seiten von R lesen und schreiben, die modifiziert wurden, sondern auch bestimmte Seiten, die den Index enthalten, lesen und schreiben. Aber selbst wenn Modifikationen die häufigste Form von Datenbankaktionen sind, kann es einen Effizienzgewinn bedeuten, einen Index auf ein häufig verwendetes Attribut zu erstellen. Da tatsächlich einige Modifikationsbefehle Abfragen in der Datenbank beinhalten, muss man vorsichtig sein, wie man die relative Häufigkeit von Modifikationen und Abfragen schätzt.

Da eine Relation auf viele Festplattenblöcke verteilt ist, entstehen die Hauptkosten einer Abfrage oder Modifikation durch das Laden der Seiten in den Hauptspeicher. Indizes können hierbei Zeit sparen, da sie den Zugriff auf Tupel ohne vollständige Durchsuchung der Relation ermöglichen. Allerdings müssen auch Indizes auf der Festplatte gespeichert werden, was zusätzliche Festplattenzugriffe verursacht. Modifikationen sind daher etwa doppelt so teuer wie der Zugriff auf den Index oder die Daten in einer Abfrage.

Um den neuen Wert eines Index zu berechnen, müssen wir Annahmen darüber treffen, welche Abfragen und Modifikationen mit hoher Wahrscheinlichkeit auf der Datenbank durchgeführt werden. Manchmal haben wir eine Historie von Abfragen, die wir nutzen können, um gute Informationen zu erhalten, unter der Annahme, dass die Zukunft wie die Vergangenheit sein wird.

**Beispiel:**

StarsIn(movieTitle, movieYear, starName)

Q1: SELECT movieTitle, movieYear FROM StarsIn WHERE starName = s;

Q2: SELECT starName FROM StarsIn WHERE movieTitle = t AND movieYear = y;

Ein Bild, das Text, Schrift, Reihe, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.I: INSERT INTO StarsIn VALUES(i, *y,* s) ;

**Voraussetzungen**

1. StarsIn belegt 10 Seiten, daher die Kosten bei gesamter Untersuchung bei 10
2. Im Durchschnitt ist ein Star in 3 Filmen aufgetreten und ein Film hat 3 Stars
3. Selbst mit Index einen Durchschnitt von 3 Tupel für einen Star oder Film, ohne 10
4. Wenn eine Indexseite modifiziert/eingefügt werden muss, ist ein weiterer Festplattenzugriff erforderlich, um die modifizierte Seite zurückzuschreiben
5. Für eine Einfügung ohne Index sind auch zwei Festplattenzugriffe nötig

Wenn es nur einen Index auf Stars gibt, erfordert Q2 immer noch einen Scan der gesamten Relation (Kosten 10). Allerdings kann Q1 beantwortet werden, indem eine Indexseite gelesen wird, um die drei Tupel für einen bestimmten Star zu finden, und anschließend drei weitere Zugriffe erfolgen, um diese Tupel zu finden. Einfügen I erfordert, dass sowohl eine Seite des Index als auch eine Seite der Daten gelesen und geschrieben wird, was insgesamt 4 Festplattenzugriffe bedeutet.

Die letzte Zeile in Abb. 8.3 gibt die durchschnittlichen Kosten einer Aktion an, unter der Annahme, dass der Anteil der Zeit, in der wir Qi ausführen, p1 beträgt und der Anteil der Zeit, in der wir Q2 ausführen, p2 ist. Daher beträgt der Anteil der Zeit, in der wir I ausführen, 1 — p1 — p2.

Je nach pi und p2 kann jede der vier Optionen (Index/kein Index) die besten durchschnittlichen Kosten für die drei Aktionen ergeben. Zum Beispiel, wenn pi = p2 = 0,1, dann ist der Ausdruck 2 + 8pi + 8p2 am kleinsten, sodass wir es bevorzugen würden, keine Indizes zu erstellen.

Das heißt, wenn wir überwiegend Einfügungen durchführen und nur sehr wenige Abfragen, dann wollen wir keinen Index. Intuitiv, wenn wir viele Abfragen durchführen und die Anzahl der Abfragen, die Filme und Stars angeben, ungefähr gleich häufig sind, dann sind beide Indizes erwünscht. Wenn wir nur ein Typ von Query häufig verwenden, dann sollten wir nur den Index definieren, der uns bei dieser hilft.

Das "Tuning" einer Datenbank umfasst nicht nur die Auswahl von Indizes, sondern auch die Festlegung vieler verschiedener Parameter. Es gibt zahlreiche Tools, die entwickelt wurden, um diese Verantwortung vom Datenbankdesigner zu übernehmen, indem das System sich selbst optimiert oder dem Designer zumindest Empfehlungen für sinnvolle Entscheidungen gibt. Dabei ermittelt das Tool zuerst die Abfragelast, dann können Einschränkungen festgelegt werden. Anschließend werde Kandidaten-Indizes generiert und bewertet und zu guter Letzt wählt man die Index-Sets aus. Ein komplexeres Problem tritt beim dritten Schritt auf, wenn bereits gewählte Indizes existieren, da diese beeinflussen können, wie viel Nutzen ein zusätzlicher Index bringt. Ein bewährter Ansatz zur Auswahl von Indizes ist das sogenannte „greedy“ - Verfahren, bei dem zunächst ohne ausgewählte Indizes der Nutzen jedes Kandidaten-Index bewertet wird. Wenn es einen Index mit positivem Nutzen gibt, wird dieser ausgewählt und anschließend wird eine Neubewertung ausgeführt, wobei davon ausgegangen wird, dass der zuvor ausgewählte Index bereits verfügbar ist. Dieser Prozess wird so lange wiederholt, bis es keinen Kandidaten-Index mit positivem Nutzen mehr gibt.

**Materialisierte Sichten (Materialized Views)**

* Viele Datenbankmanagementsysteme bieten diese Funktionalität an, MySQL jedoch nicht
* Materialisierte Sichten sind Sichten, die tatsächlich vorab berechnet und als Tabellen auf der Festplatte gespeichert werden. Sie können durch verschiedene Strategien aktualisiert oder aufgefrischt werden
* FlexViews (eine spezielle Implementierung eines Entwicklers) verwenden Deltas (Unterschiede zwischen den vorherigen und den aktuellen Daten), was wesentlich effizienter ist, als die Daten aus der Quelltabelle zu lesen

**Zähltabellen (Counter Tables)**

* Eine Anwendung, die Zählwerte in einer Tabelle speichert, kann bei der Aktualisierung der Zähler auf Nebenläufigkeitsprobleme stoßen
* Die Verwendung einer separaten Tabelle kann helfen, Abfrage-Cache-Invalideierungen zu vermeiden und ermöglicht den Einsatz fortgeschrittenerer Techniken
* Verwenden Sie eine zufällige Slot-Variable für bessere Nebenläufigkeit (und verwenden Sie sie zusammen mit dem Tag als Primärschlüssel), um den Zähler zu aktualisieren, auch mit ON DUPLICATE KEY UPDATE
* Schreiben Sie einen periodischen Job, der alle Ergebnisse in Slot 0 zusammenführt und alle anderen Slots löscht

**Views**

Relationen, bzw. Tabellen, die durch das CREATE TABLE – Statement definiert werden, existieren physisch in der Datenbank. Damit sind sie persistent, was bedeutet, dass sie dauerhaft existieren und sich nicht ändern, es sei denn, sie werden explizit durch eine SQL-Änderungsanweisung dazu aufgefordert (eine Eigenschaft, die Teil der ACID-Prinzipien ist, https://www.bigdata-insider.de/was-ist-acid-a-776182/). Es gibt jedoch eine weitere Klasse von SQL-Relationen, die nicht wie Tabellen tatsächlich physisch gespeichert werden und als virtuelle Sichten bezeichnet wird. Virtuelle Sichten werden durch einen Ausdruck definiert, der einer Abfrage ähnelt und können wiederum abgefragt werden, als ob sie physische existieren. In einigen Fällen können die Datensätze der Sicht sogar geändert werden.

Die einfachste Form der View - Deklarierung sieht so aus: CREATE VIEW <view-name> AS <view-definition>;

**TODO: add an example from the lua-script**

Bei der Abfrage geben wir den Namen der virtuellen Sicht in einer FROM – Klausel an und verlassen uns darauf, dass das DBMS die benötigten Tupel erzielt, indem es auf den Relationen operiert, die die virtuelle Sicht definieren. Eine Möglichkeit, um sich die Funktionsweise dieser Sicht besser vorzustellen, kann man jede Sicht in einer FROM-Klausel durch eine Unterfrage ersetzen, die mit der Sichtdefinition identisch ist. Diese Unterabfrage müssen wir noch mit einer Tupelvariablen ergänzen, damit wir auch Bezug auf die Tupel nehmen können. Man kann den Attributen einer Sicht auch eigene Namen vergeben, indem wir sie in Klammern hinter dem Namen der Sicht aus der CREATE VIEW-Anweisung auflisten.

Die Definition einer Sicht kann man mit DROP VIEW <view-name> löschen, sodass auf dieser Sicht keine Abfragen oder Änderungsbefehle ausgeführt werden können. Das Löschen der Sicht hat jedoch keine Auswirkungen auf die Tupel der zugrundeliegenden Tabelle(n). Im Gegensatz dazu würde DROP TABLE <table-name> die Tabelle entfernen und die View unbrauchbar machen.

Unter bestimmten Bedingungen erlaubt SQL Änderungen an Sichten. Dafür muss die Sicht durch eine einfache Abfrage aus nur einer einzigen Relation definiert sein und außerdem muss die SELECT-Klausel ausreichend Attribute umfassen, sodass fehlende Werte bei Einfügungen mit NULL oder Standardwerten ergänzt werden können. Die Änderungen werden dann direkt auf die Basistabelle angewendet, wobei nur die in der Sicht definierten Attribute berücksichtigt werden.

**Bsp.: INSERT INTO ParamountMovies VALUES(’Star Trek’, 1979);**

Der Befehl wird so ausgeführt wie dieser hier:

**INSERT INTO Movies(title, year) VALUES('Star Trek', 1979);**

Für die vier nicht definierten Attribute hat das eingefügte Tupel den Wert NULL. Das damit auch das Attribut studioName den Wert NULL hat, wird die Auswahlbedingung für die View nicht erfüllt und das eingefügte Tupel hat keine Wirkung auf die Sicht. Um dieses Problem zu lösen, könnten wir studioName in die SELECT-Klausel der Sicht aufnehmen und dann bei dem Insert-Befehl auch definieren. Auch bei Löschungen und Aktualisierungen wird die Änderung auf die zugrundeliegende Relation R übertragen. Die WHERE-Bedingung der View wird zu den Bedingungen der Änderung im WHERE-Block hinzugefügt.

**DELETE FROM ParamountMovies WHERE title LIKE ’‘/.Trek\*/.’ ;**

Ist äquivalent zu:

**DELETE FROM Movies WHERE title LIKE ’‘/.Trek’/,’ AND studioName = ’Paramount’ ;**

Anstelle der Einfügungen an Views, wie bisher beschrieben, können wir auch Trigger benutzen, die BEFORE oder AFTER einem Event ausgelöst werden. Bei Triggern auf Sichten können auch INSTEAD-OF-Trigger verwendet werden, der Versuche unterbricht, die Sicht zu ändern und stattdessen eine Aktion ausführt, die sich frei definieren lässt. Wenn wir beispielsweise eine View haben, die 2 Tabellen joint, dann haben wir keine andere Möglichkeit als Änderungen durch INSTEAD-OF-Trigger durchzuführen.

**CREATE TRIGGER ParamountInsert**

**INSTEAD OF INSERT ON ParamountMovies**

**REFERENCING NEW ROW AS NewRow**

**FOR EACH ROW**

**INSERT INTO Movies(title, year, studioName)**

**VALUES(NewRow.title, NewRow.year, ’Paramount’);**

Der Trigger ersetzt die Einfügungsversuche auf die Sicht durch eine Einfügung der zugrundeliegenden Tabelle. Dabei stammen die bekannten Attribute title und year aus dem eingefügten Tupel, während studioName fest auf „Paramount“ gesetzt wird. Damit sind die Ergebnisse der Einfügungen auf der virtuellen Sicht auch tatsächlich in dieser sichtbar.

**Materialized Views**

Generell beschreibt eine Sicht, wie eine neue Relation aus Basistabellen durch Ausführen einer Abfrage auf diesen Tabellen erstellt werden kann. Bisher haben wir Sichten nur als logische Beschreibungen von Relationen betrachtet, aber wenn diese häufig verwendet wird, kann es effizient sein, sie zu materialisieren, also den Wert physisch zu speichern. Die Pflege der materialisierten Sicht ist, wie auch schon bei Indizes, mit zusätzlichem Aufwand verbunden, da Teile der Sicht jedes Mal neu berechnet werden müssen, wenn sich die zugrundeliegende Basistabelle ändert. Die Anzahl an Neuberechnungen kann aber auf ein Minimum begrenzt werden und auch der Arbeitsaufwand pro Aktualisierung kann reduziert werden.

Bsp:

**CREATE MATERIALIZED VIEW MovieProd AS SELECT title, year, name  
FROM Movies, MovieExec  
WHERE producerC# = cert#**

Das DBMS muss Änderungen, die nicht in der Abfrage enthalten sind, sowie Relationen, die nicht betroffen sind, nicht berücksichtigen. Es gibt jedoch noch weitere Vereinfachungen, die es uns ermöglichen, Änderungen an **Movies** oder **MovieExec** effizienter zu behandeln als durch eine erneute Ausführung der Abfrage.

Wenn wir einen neuen Film in die **Movie** - Tabelle einfügen, müssen wir nur den Namen zu der Spalte **cert#** aus **MovieExec** bekommen und da **cert#** der Schlüssel der Tabelle ist, gibt es höchstens einen Eintrag dazu. Der Name von diesem Eintrag kann direkt dazu genutzt werden, um den Datensatz in die Relation einzutragen. Da die materialisierte Sicht physisch in der Datenbank gespeichert wird, ist dies möglich und man muss nicht die komplette Tabelle neu aufbauen. Darin sieht man, dass alle Änderungen inkrementell sind und das bedeutet, dass die gesamte Sicht niemals von Grund auf neu rekonstruiert werden muss. Stattdessen können Einfügungen, Löschungen und Aktualisierungen an einer Basistabelle durch eine kleine Anzahl von Abfragen auf die Basistabellen und anschließende Änderungsanweisungen an der materialisierten Sicht umgesetzt werden. Zudem betreffen diese Änderungen nicht alle Tupel der Sicht, sondern nur die, die mindestens ein Attribut mit einem bestimmten konstanten Wert enthalten.

MySQL bietet leider keine Möglichkeit materialisierte Sichten nativ zu erstellen und damit gibt es auch keine Möglichkeit der inkrementellen Auffrischung (https://www.datacamp.com/de/tutorial/sql-materialized-view?dc\_referrer=https%3A%2F%2Fwww.google.com%2F).

Ein anderer Anwendungsfall ist die Nutzung von aggregierten Daten in einer materialisierten Sicht für Analysezwecke. Wenn beispielswiese Analysten eines Motorradbetriebes für den Zukunft den Einkauf planen wollen, dann müssen sie vergangene Daten häufig in einer aggregierten Form betrachten. Diese materialisierte Sicht wird damit eher selten abgefragt, wohin hingegen Änderungen, d.h. den Lagerbestand von Motorrädern, sehr häufig passieren. Wenn man die Sicht bei jeder Änderung aktualisieren würde, wäre dies sehr kostspielig. Daher ist es sinnvoll, einen Kompromiss einzugehen, bei dem die Daten nur einmal täglich aktualisiert werden, idealerweise als Cron-Job nachts, wenn die Systemlast gering ist. In diesem Fall haben die Analysten zwar nicht die aktuellen Daten, sondern nur den Stand des letzten Tages.

Für Analysten stellt diese Verzögerung in der Regel kein großes Problem dar, da sie in der Regel mit historischen Daten arbeiten. Wenn jedoch eine schnelle Lieferung an den Kunden versprochen wird, ist es für den Verkäufer, beispielsweise in einem Online-Shop, entscheidend, aktuelle Daten zu haben. Andernfalls riskieren wir, dem Kunden falsche Versprechungen zu machen, etwa indem wir Produkte als verfügbar anzeigen, die in Wirklichkeit nicht mehr auf Lager sind.

Eine materialisierte Sicht kann in der FROM-Klausel einer Abfrage verwendet werden, genauso wie eine virtuelle Sicht. Da jedoch eine materialisierte Sicht in der Datenbank gespeichert wird, ist es möglich, eine Abfrage so umzuschreiben, auch wenn die Sicht nicht in der ursprünglich geschriebenen Abfrage erwähnt wurde. Eine solche Umformulierung kann die Ausführung erheblich beschleunigen, da Teil der Berechnungen schon gemacht wurden, als die materialisierte Sicht erstellt worden ist.

View mit: **SELECT Lv FROM Rv WHERE Cv**

Query with: **SELECT Lq FROM Rq WHERE Cq**

Dann müssen die Relationen in der Liste Rv alle in der Liste Rq vorkommen. Die Bedingung Cq ist äquivalent zu Cv und C für eine Bedingung C. Als Sonderfall könnte Cq auch äquivalent zu Cv sein. Wenn C erforderlich ist, dann müssen die Attribute der Relationen in der Liste Rv, auf die C Bezug nimmt, auch in der Liste Lv der Attribute vorhanden sein. Attribute in der Liste Lq, die aus Relationen in der Liste Rv stammen, müssen ebenfalls in der Liste Lv vorhanden sein. Wenn alle Bedingungen erfüllt sind, dann kann die Abfrage Q umschrieben werden, um die Sicht V zu verwenden. Zuerst ersetze die Liste Rq durch V und die Relationen, die in der Liste Rq sind, aber nicht in Rv. Zweitens ersetze Cq durch C. Falls C nicht erforderlich ist (d. h. Cv = Cq), entfällt die WHERE-Klausel.

**Example**

Das bedeutet, dass wir in der FROM-Klausel die Tabellen Movies und MovieExec durch die materialisierte Sicht MovieProd ersetzt haben. Außerdem haben wir die Bedingung der Sicht aus der WHERE-Klausel entfernt und nur die Bedingung C beibehalten. Da die umgeschriebene Abfrage nur den Join von zwei Relationen statt von drei enthält, erwarten wir, dass die umgeschriebene Abfrage weniger Zeit zur Ausführung benötigt als die Ursprüngliche.

Die Auswahl von materialisierten Sichten ist deutlich komplexer als die von Indizes, da potenziell jede Abfrage eine Sicht definieren könnte. Daher sollten nur Sichten erstellt werden, die mindestens eine Abfrage der erwarteten Workload verbessern, wobei Kriterien wie Relationen, Bedingungen und Attribute berücksichtigt werden. Zudem muss der Nutzen einer Sicht nicht nur anhand der Laufzeitverbesserung, sondern auch im Verhältnis zu ihrem Speicherbedarf bewertet werden, da materialisierte Sichten oft nicht nur erheblich mehr Speicherplatz beanspruchen können, sondern untereinander sich von der Größe deutlich unterscheiden.