Benchmarks dienen dazu, praktisch und effektiv zu untersuchen, wie sich ein System unter Last verhält. Die wichtigste Erkenntnis, die man aus Benchmarks gewinnen kann, sind die Probleme und Fehler, die man systematisch dokumentieren und nach Priorität abarbeiten sollte. Das Ziel von Benchmarks ist die Reduzierung und Bewertung von unerwünschtem Verhalten sowie die Analyse, wie sich das System derzeit und unter simulierten, zukünftigen, anspruchsvolleren Bedingungen verhalten könnte.

Es gibt zwei verschiedene Techniken für Benchmarks. Die erste zielt darauf ab, die Applikation als Ganzes zu testen (full-stack). Dabei wird nicht nur die Datenbank getestet, sondern die gesamte Applikation, einschließlich des Webservers, des Netzwerks und des Applikationscodes. Der Ansatz dahinter ist, dass ein Nutzer genauso lange auf eine Abfrage warten muss, wie das gesamte System benötigt. Daher sollte diese Wartezeit so gering wie möglich sein. Es kann dabei vorkommen, dass MySQL nicht immer das Bottleneck ist. (Fußnote hoch: Gemeint ist ein Engpass beim Transport von Daten oder Waren, der maßgeblichen Einfluss auf die Abarbeitungsgeschwindigkeit hat. Optimierungsversuche an anderer Stelle führen oft nur zu geringen oder gar keinen messbaren Verbesserungen der Gesamtsituation. (<https://martinvogel.de/lexikon/bottleneck.html)>). Full-Stack-Benchmarks haben jedoch auch Nachteile. Sie sind schwieriger zu erstellen und insbesondere schwieriger korrekt einzurichten. Wenn man lediglich verschiedene Schemas und Abfragen in MySQL auf ihre Performance testen möchte, gibt es sogenannte Single-Component-Benchmarks. Diese analysieren ein spezifisches Problem in der Applikation und sind deutlich einfacher zu erstellen. Ein weiterer Vorteil besteht darin, dass nur ein Teil des gesamten Systems getestet wird, wodurch die Antwortzeiten kürzer sind und man schneller Ergebnisse erhält.

Wenn bei Benchmarks schlechte Designentscheidungen getroffen werden, kann dies zu einer falschen Interpretation des Systems führen, da die Ergebnisse nicht die Realität widerspiegeln. Die Größe des Datensatzes und des Workloads muss realistisch sein. Idealerweise verwendet man einen Snapshot (Fußnote) des tatsächlichen produktiven Datensatzes. Gibt es keine Produktionsdaten, sollten die Daten und der Workload simuliert werden, da realistische Benchmarks komplex und zeitaufwendig sein können.

Häufige Fehler beim Durchführen von Benchmarks sind unter anderem, dass nur ein kleiner Teil der tatsächlichen Datensatzgröße verwendet wird und die Datensätze unkorrekt gleichmäßig verteilt sind. In der Realität können Hotspots auftreten. Bei zufällig generierten Werten kommt es hingegen häufig zu unrealistisch gleichmäßig verteilten Datensätzen. Ein weiterer Fehler besteht darin, dass man beim Testen einer Anwendung nicht das tatsächliche Benutzerverhalten nachstellt. Wenn gleiche Abfragen in einer Schleife ausgeführt werden, muss man außerdem auf das Caching achten, da sonst falsche Annahmen über die Performance getroffen werden können. Zudem wird oft die Warmmachphase des Systems vollständig ignoriert. Kurze Benchmarks können schnell zu falschen Annahmen über die Performance des Systems führen.

Um verlässliche Ergebnisse zu erhalten, sollte ein Benchmark ausreichend lange laufen, um den stabilen Zustand des Systems zu beobachten, insbesondere bei Servern mit großen Datenmengen und viel Speicher. Dabei ist es wichtig, so viele Informationen wie möglich zu erfassen und sicherzustellen, dass der Benchmark wiederholbar ist, da unzureichende oder fehlerhafte Tests wertlos sind. Außerdem ist es wichtig, die Ergebnisse in einem Diagramm darzustellen, da auftretende Phänomene sonst anhand einer tabellarischen Darstellung nicht erkannt werden können.

**Measures**

* **Durchsatz (Throughput):**Der Durchsatz ist die Anzahl an Transaktionen pro Zeiteinheit. Er ist standardisiert, und Datenbankanbieter versuchen, diesen zu optimieren. Meistens werden Transaktionen pro Sekunde (oder manchmal pro Minute) als Einheit verwendet.
* **Antwortzeiten (Latenz):** Die Antwortzeit misst die gesamte Zeit, die für eine Abfrage benötigt wird. Diese kann, abhängig von der Applikation, in Mikrosekunden (µs), Millisekunden (ms), Sekunden oder Minuten angegeben werden. Von dieser Zeit können aggregierte Antwortzeiten wie Durchschnitt, Maximum, Minimum und Perzentile abgeleitet werden. Das Maximum ist oft eine weniger sinnvolle Metrik, da es sich nicht gut wiederholen lässt. Daher nutzt man eher Perzentile bei den Antwortzeiten. Wenn beispielsweise das 95. Perzentil der Antwortzeit bei 5 ms liegt, bedeutet dies, dass mit einer Wahrscheinlichkeit von 95 % die Abfrage in weniger als 5 ms abgeschlossen ist.
* **Nebenläufigkeit (Concurrency):** Die Nebenläufigkeit auf dem Webserver lässt sich nicht zwangsläufig auf den Datenbankserver übertragen. Eine genauere Messung der Gleichzeitigkeit auf dem Webserver besteht darin, zu bestimmen, wie viele gleichzeitige Anfragen zu einem bestimmten Zeitpunkt ausgeführt werden. Es kann auch geprüft werden, ob der Durchsatz sinkt oder die Antwortzeiten steigen, wenn die Gleichzeitigkeit zunimmt. Beispielsweise benötigt eine Website mit „50.000 Benutzern gleichzeitig“ vielleicht nur 10 oder 15 gleichzeitig laufende Abfragen.
* **Skalierbarkeit (Scalability):** Skalierbarkeit ist wichtig für Systeme, die ihre Performance unter unterschiedlich starken Workloads beibehalten müssen. Ein ideales System würde doppelt so viele Abfragen beantworten (Throughput), wenn doppelt so viele „Arbeiter“ versuchen, die Aufgaben zu erfüllen. Die meisten Systeme sind jedoch nicht linear skalierbar und zeigen Leistungseinbußen, wenn die Parameter variieren.

**Tools** (<https://www.webhosterwissen.de/know-how/server/mysql-benchmark-mittels-sysbench/>, <https://github.com/akopytov/sysbench>)

Als Haupttool, um Benchmarktests durchzuführen, habe ich mich für Sysbench entschieden. Sysbench ist ein Open-Source-Tool, das ein skriptfähiges, multi-threaded Benchmark-Tool, das auf LuaJIT basiert. Es wird auch hauptsächlich für Datenbankbenchmarks verwendet, kann jedoch auch dazu eingesetzt werden, beliebig komplexe Arbeitslasten zu erstellen, die keinen Datenbankserver erfordern. Dabei werden Tests auf verschiedenen Systemressourcen, wie CPU, Speicher, I/O und Datenbanken wie MySQL verwendet.

Im Zuge der Recherchearbeit habe ich mir auch andere Benchmarking – Tools betrachtet, wie z.B. Benchbase (<https://github.com/cmu-db/benchbase>) oder mybench (<https://github.com/Shopify/mybench>). Die größten Vorteile von Sysbench habe ich in der Skriptfähigkeit und Flexibilität gesehen. D.h. dass ich benutzerdefinierte Benchmarks schneller und unkomplizierten erstellen kann. Außerdem hat sich Sysbench als de facto standard im Bereich der Datenbankbenchmarks etabliert (<https://shopify.github.io/mybench/introduction.html)>., wodurch eine breite Nutzerbasis und viele verfügbare Ressourcen zur Verfügung stehen. Im Vergleich zu den anderen Tools bietet allerdings Sysbench eine weniger präzise Steuerung der Ergebnisrate und der Transaktionen. Außerdem haben Tools, wie mybench, die Möglichkeit in Echtzeit umfassende Visualisierung darzustellen. Damit können Metriken live in einem Diagramm angezeigt werden (<https://shopify.github.io/mybench/detailed-design-doc.html#live-monitoring-user-interface>). Dieses Feature ist sicherlich hilfreich, im meinem Fall, habe ich aber abgewogen und bin zu dem Entschluss gekommen, dass die einfachere Bedienung für mich der ausschlaggebende Grund, neben dem Fakt, dass Sysbench der de facto standard ist.

Da die Graphen aber trotzdem eine entscheidende Rolle bei der Analyse darstellen, werde ich das Tool Gnuplot (<https://github.com/gnuplot/gnuplot>) dafür benutzen, um die Werte nach der Durchführung des Benchmarks die Ergebnisse zu visualisieren.

Kurze Einführung in die Tools

Zunächst müssen die beiden Tools installiert werden. Auf einem Mac erfolgt dies über ***brew install sysbench*** und ***brew install gnuplot*** mit der Hilfe von Homebrew. Desweiten muss der MySQL - Server korrekt gestartet sein und eine Database erstellt werden mit ***CREATE DATABASE sbtest***. Um einfach Testdaten in die Datenbank einzufügen kann dieser Befehl verwendet werden: sysbench --db-driver=mysql --mysql-host=localhost --mysql-user= YOUR\_USER --mysql-password= YOUR\_PASSWORD --mysql-db=sbtest oltp\_insert --tables=10 --table-size=100000 prepare. YOUR\_USER und YOUR\_PASSWORD müssen entsprechend ersetzt werden.

sysbench --db-driver=mysql --mysql-host=localhost --mysql-user=root --mysql-password=password --mysql-db=sbtest oltp\_insert --tables=10 --table-size=100000 prepare

sysbench 1.0.20 (using system LuaJIT 2.1.1725453128)

**Beispiel**

* Kunden mit Name als ID und einmal extra ID als PK
* Bestellung mit Kunde als FK => References one time varchar and one time number

**Include in Bachelorarbeit**

* Benchmarks techniques and basics
* Using incorrectly distributed data (in real world may be “hot spots”) => randomly generated data is almost always unrealistically distributed => check if I need to implement this part

**Portugal – Neue Beispiele => TODO: Umsetzen in Benchmarks**

* Einfluss der Auswahl von Datentypen (Ganzzahlen, Gleitzahlen, String Typen, BLOB und TEXT Typen, **ENUM, ZEIT – und Datumstypen**)
* Richtigen Schlüssel auswählen
* Unterschied zwischen denormalisiertem und normalisiertem Schema aufzeigen
* Effekt von Zähler und Summentabellen veranschaulichen
* Indexing für bessere Performance

**Einfluss der Auswahl von Datentypen**

* Ganzzahlen
* Gleitzahlen
* String Typen
* BLOB und TEXT Typen
* ENUM
* ZEIT – und Datumstypen

**Richtigen Schlüssel auswählen**

**(De-) Normalisierung**

**Summen – und Zählertabellen**

* **Alter Table beschleunigen**

**Indexing für bessere Performance**

* Basics
* B-Tree Inde
* Hash Index
* Limitations
* (Eigenen Index bauen)
* Spatial Index (Geometrie)
* Full Text Index
* Vorteile
* Beste Lösung?
* Index Strategies
* Mulitcolumn Index
* Column Order auswählen
* Clustered Index

**Einfluss der Auswahl von Datentypen**

**Allgemein:**

* **Kleiner** ist besser
  + Wähle den kleinstmöglichen Datentypen, den du speichern kannst und der deine vorhandenen Daten entsprechend repräsentieren kann
  + Meistens schneller, da weniger Speicherplatz (In-Memory und CPU – Cache) => auch weniger CPU-Zyklen zu verarbeiten
  + Spätere Typveränderungen gehen schnell (wenn man es falsch einschätzt)
* **Einfach** ist gut
  + Weniger CPU-Zyklen benötigt, um Operationen auf einfachen Datentypen zu verarbeiten
  + Integer ist einfacher zu verarbeiten als Character, da Character Sets und Sortierregeln den Character – Vergleich erschweren (**TODO:** what means collations?)
* **NULL** vermeiden, wenn möglich
  + Viele Tabellen enthalten NULLABLE Spalten, selbst wenn die Anwendung keine NULL (das Fehlen eines Wertes) speichern muss => dies ist die Standardeinstellung
  + Es ist am besten, Spalten als NOT NULL (NOT NULL = keine NULL-Werte erlaubt) zu definieren => außer NULL – Werte sollen gespeichert werden
  + Es ist schwieriger für MYSQL, Abfragen zu optimieren, die auf nullable Spalten verweisen => durch Indexe, Indexstatistiken (index statistics) und Wertvergleiche (value comparison) komplizierter, benötigen mehr Speicherplatz und erfordern spezielle Verarbeitung innerhalb von MYSQL
  + Indizierte nullable Spalte benötigt ein zusätzliches Byte pro Eintrag und kann dazu führen, dass ein Index mit fester Größe (fixed-size index) in einen variablen Index (variable-sized) umgewandelt wird
  + Die Leistungsverbesserung (performance improvement), die durch die Änderung von NULL-Spalten in NOT NULL erzielt wird, ist in der Regel gering => bei Verwendung von Indizes sollten aber nullable Spalten vermieden werden
* **Schritte** zur Auswahl von Datentypen (data types)

1. **Bestimmen** Sie, welche allgemeine **Klasse von Typen** (class of types) geeignet ist: numerisch (numeric), Zeichenketten (string), zeitbezogen (temporal), usw

2. **Wählen** Sie den **spezifischen Typ** (specific type): Viele können dieselbe Art von Daten speichern, unterscheiden sich jedoch im **Bereich der Werte** (range of values), die sie speichern können, der **Genauigkeit** (precision), die sie erlauben, oder dem **physischen Speicherplatz** den sie benötigen (auf der Festplatte (on disk) und im Arbeitsspeicher (in memory)). Einige haben auch spezielle Verhaltensweisen oder Eigenschaften

* **DATETIME** und **TIMESTAMP** können dieselbe Art von Daten speichern mit einer Genauigkeit von einer Sekunde. TIMESTAMP benötigt jedoch nur halb so viel Speicherplatz, ist zeitzonenbewusst (time-zone aware) und verfügt über spezielle Auto-Update-Funktionen (auto updating capabilities). Allerdings hat TIMESTAMP einen viel kleineren Bereich erlaubter Werte (range of allowable values), und manchmal können seine speziellen Fähigkeiten ein Nachteil sein
* MYSQL unterstützt viele **Aliase** (aliases) (z. B. INTEGER, BOOL, NUMERIC) => das kann verwirrend sein, beeinflusst jedoch nicht die Leistung (performance)
  + Wenn Sie eine Tabelle mit einem aliasierten Datentyp (aliased data type) erstellen und dann SHOW CREATE TABLE untersuchen, wird MYSQL den Basistyp (base type) anzeigen

**Ganze Zahlen (whole numbers)**

* Es gibt zwei Arten von Zahlen: ganze Zahlen (whole numbers) und reelle Zahlen (real numbers, Zahlen mit einem Bruchteil)
* Zum Speichern von ganzen Zahlen verwenden Sie einen der **Integer-Typen**:
  + TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT
  + Benötigen 8, 16, 24, 32 bzw. 64 Bits Speicherplatz
  + Können Werte von -2(N-1) bis 2(N-1)-1 speichern, wobei N die Anzahl der Speicherbits ist
* Integer-Typen können optional das **UNSIGNED**-Attribut haben, das negative Werte ausschließt und die obere Grenze der speicherbaren positiven Werte ungefähr verdoppelt => bei TINYINT [0; 255] statt [-128; 127]
* Signed- und unsigned-Typen verwenden denselben Speicherplatz und haben dieselbe Leistung (performance)
* Integer-Berechnungen verwenden in der Regel 64-Bit-BIGINT-Integer, selbst auf 32-Bit-Architekturen (außer Aggregatfunktionen, die DECIMAL oder DOUBLE verwenden)
* MYSQL erlaubt die Angabe der "Breite" (width) für integer-Typen, z. B. INT(11) => dies schränkt den zulässigen Wertebereich nicht ein, vereinfacht jedoch die Anzeige. Für Speicher- und Berechnungszwecke ist INT(1) identisch mit INT(20)

**Reelle Zahlen (real numbers)**

* Reelle Zahlen sind Zahlen mit einem Bruchteil
* Mit DECIMAL können auch ganze Zahlen gespeichert werden, die so groß sind, dass sie nicht in BIGINT passen
* FLOAT- und DOUBLE-Typen unterstützen ungefähre Berechnungen mit standardmäßiger Gleitkomma-Arithmetik (floating-point math)
* DECIMAL-Typen werden verwendet, um exakte Bruchzahlen zu speichern
* Sowohl Gleitkomma- (floating-point) als auch DECIMAL-Typen erlauben die Angabe einer Genauigkeit (precision) => bei DECIMAL kann die maximale Anzahl der Ziffern vor und nach dem Dezimalpunkt angegeben werden
* Gleitkomma-Arithmetik (floating-point math) ist erheblich schneller, da die CPU die Berechnungen nativ ausführt, allerdings gibt es einen Präzisionsverlust
* MYSQL packt die Ziffern in eine binäre Zeichenkette (binary string, neun Ziffern pro vier Bytes)
* DECIMAL(18, 9) speichert neun Ziffern auf jeder Seite des Dezimalpunkts und benötigt insgesamt 9 Bytes: vier für die Ziffern vor dem Dezimalpunkt, eine für den Dezimalpunkt selbst und vier für die Ziffern nach dem Dezimalpunkt
* DECIMAL kann bis zu 65 Ziffern haben
* Gleitkomma-Typen (floating-point types) benötigen in der Regel weniger Speicherplatz als DECIMAL, um denselben Wertebereich zu speichern:
  + FLOAT-Spalten (FLOAT column) benötigen vier Bytes Speicherplatz
  + DOUBLE benötigt acht Bytes und bietet höhere Präzision und einen größeren Wertebereich als FLOAT
* Wie bei ganzen Zahlen (integers) wählen Sie nur den Speicher-Typ (storage type)
  + MYSQL verwendet DOUBLE für interne Berechnungen bei Gleitkomma-Typen
* DECIMAL sollte nur für die Speicherung von Finanzdaten verwendet werden => hier könnte auch BIGINT genutzt werden, um sowohl die Ungenauigkeit von Gleitkomma-Speicherungen als auch die Kosten der präzisen DECIMAL-Arithmetik zu vermeiden

**String-Typen (string types)**

* Seit MySQL 4.1 kann jede String-Spalte (string column) ihre eigene Zeichencodierung (character set) und ein eigenes Sortierungsregelset (collation) für diese Zeichencodierung haben
* Die beiden Haupttypen von Strings sind VARCHAR und CHAR, die Zeichenwerte speichern
* Es ist schwierig zu erklären, wie diese Werte auf der Festplatte gespeichert werden => die Implementierung hängt von der Speicher-Engine (storage engine) ab
* VARCHAR speichert Zeichenfolgen mit variabler Länge (variable-length character strings) und ist der am häufigsten verwendete String-Datentyp
  + Benötigt weniger Speicherplatz als Typen mit fester Länge (fixed-length types), da nur so viel Platz verwendet wird, wie tatsächlich benötigt wird (kürzere Werte benötigen weniger Platz)
  + VARCHAR verwendet 1 oder 2 zusätzliche Bytes, um die Länge des Wertes aufzuzeichnen (1 Byte für Werte < 255 Bytes)
  + Da die Zeilen jedoch variable Länge haben, können sie beim Aktualisieren wachsen, was zusätzlichen Aufwand verursachen kann => das Verhalten, wenn der Wert nicht mehr passt, ist von der Engine abhängig
  + Es lohnt sich, VARCHAR zu verwenden, wenn die maximale Spaltenlänge deutlich größer als die durchschnittliche Länge ist
* CHAR ist festlängig (fixed-length) und MySQL reserviert immer genügend Platz für die angegebene Anzahl von Zeichen
  + Nützlich, wenn Sie sehr kurze Strings speichern oder wenn alle Werte fast die gleiche Länge haben
  + CHAR ist besser als VARCHAR für Daten, die häufig geändert werden, da eine feste Zeilenlänge nicht zu Fragmentierung neigt
  + Für sehr kurze Spalten ist CHAR effizienter als VARCHAR => CHAR(1) benötigt nur ein Byte, aber VARCHAR(1) benötigt 2 Bytes wegen des Längen-Bytes
* Die verwandten Typen (sibling types) von CHAR und VARCHAR sind BINARY und VARBINARY, die binäre Strings speichern
  + Binäre Strings (binary strings) sind den herkömmlichen Strings sehr ähnlich, speichern jedoch Bytes anstelle von Zeichen, und die Auffüllung (padding) ist anders (MySQL füllt BINARY-Werte mit \0 (dem Null-Byte) auf, anstelle von Leerzeichen, und entfernt die Auffüllwerte beim Abruf nicht)
  + Nützlich, wenn Sie Binärdaten (binary data) speichern und die Werte byteweise anstelle von zeichenweise vergleichen möchten
    - Der Vorteil des byteweisen Vergleichs (byte-wise comparisons) liegt nicht nur in der Fallunempfindlichkeit (case insensitivity)
    - Binäre Vergleiche (binary comparisons) können viel einfacher und schneller sein als Zeichenvergleiche
* Das Speichern des Wertes 'hello' benötigt in einer VARCHAR(5)- und einer VARCHAR(200)-Spalte gleich viel Speicherplatz, ist jedoch schlecht für Sortierungen oder Operationen, die temporäre Tabellen im Speicher verwenden => reservieren Sie nur so viel Platz, wie Sie benötigen
* Vorsicht mit dem BINARY-Typ, wenn der Wert nach dem Abruf unverändert bleiben muss. MySQL füllt ihn auf die erforderliche Länge mit \0s auf

**BLOB- und TEXT-Typen (BLOB and TEXT types)**

* Sind String-Datentypen, die entwickelt wurden, um große Datenmengen als binäre Strings (binary strings) bzw. Zeichen-Strings (character strings) zu speichern
* Es gibt jeweils Familien von Datentypen:
  + Zeichen-Typen (character types) sind TINYTEXT, SMALLTEXT, TEXT, MEDIUMTEXT und LONGTEXT => BLOB ist ein Synonym für SMALLBLOB
  + Binäre Typen (binary types) sind TINYBLOB, SMALLBLOB, BLOB, MEDIUMBLOB und LONGBLOB => TEXT ist ein Synonym für SMALLTEXT
* Im Gegensatz zu allen anderen Datentypen behandelt MySQL jeden BLOB- und TEXT-Wert als ein Objekt mit eigener Identität
* Der einzige Unterschied zwischen den beiden Typen besteht darin, dass BLOB-Typen binäre Daten ohne Sortierung (collation) oder Zeichencodierung (character set) speichern, während TEXT-Typen eine Zeichencodierung und Sortierung haben
* Anstelle die gesamte Länge eines Strings zu sortieren, sortiert MySQL nur die ersten max\_sort\_lengthZeichen solcher Spalten:
  + Wenn Sie nur nach den ersten Zeichen sortieren möchten, können Sie die Servervariable max\_sort\_length verringern oder ORDER BY SUBSTRING(column, length) verwenden
* Abfragen, die BLOB- oder TEXT-Spalten verwenden und eine temporäre Tabelle benötigen => können zu einem erheblichen Leistungsaufwand führen
  + Die beste Lösung ist, die BLOB- und TEXT-Typen zu vermeiden, es sei denn, sie sind wirklich notwendig
  + Wenn sie unvermeidlich sind, könnten Sie überall, wo eine BLOB-Spalte erwähnt wird, den Trick mit SUBSTRING(column, length) verwenden => konvertiert die Werte in Zeichen-Strings, wodurch temporäre Tabellen im Speicher genutzt werden können
  + Stellen Sie sicher, dass das Substring kurz genug ist, damit die temporäre Tabelle nicht größer als ***max\_heap\_table\_size*** oder ***tmp\_table\_size*** wird

**ENUM**

* Manchmal können ENUM-Typen anstelle von herkömmlichen String-Typen verwendet werden
* Eine ENUM-Spalte kann eine vordefinierte Menge von eindeutigen Werten speichern
* MYSQL speichert sie sehr kompakt, gepackt in ein oder zwei Bytes, je nach Anzahl der Werte in der Liste
* Speichert jeden Wert intern als eine Ganzzahl (integer), die seine Position in der Felddefinitionsliste repräsentiert
* Die "Nachschlagetabelle" (lookup table), die die Zuordnung von Zahlen zu Strings definiert, wird in der .frm-Datei der Tabelle gespeichert
* Diese Dualität kann sehr verwirrend sein, wenn Sie Zahlen für Ihre ENUM-Konstanten angeben => z. B. ENUM('1', '2', '3') => Wir empfehlen, dies nicht zu tun
* ENUM-Felder sortieren nach den internen Integer-Werten, nicht nach den Strings selbst
  + Geben Sie ENUM-Mitglieder in der Reihenfolge an, in der Sie sie sortieren möchten
  + Oder verwenden Sie FIELD(), um eine Sortierreihenfolge explizit in Ihren Abfragen anzugeben, was jedoch verhindert, dass MySQL den Index für die Sortierung verwendet
* Der größte Nachteil von ENUM ist, dass die Liste der Strings fest ist, und das Hinzufügen oder Entfernen von Strings erfordert die Verwendung von ALTER TABLE
* Es kann langsamer sein, eine CHAR- oder VARCHAR-Spalte mit einer ENUM-Spalte zu verbinden, als mit einer anderen CHAR- oder VARCHAR-Spalte
* Beispielanalyse mit 2 Spalten, die von VARCHAR in ENUM konvertiert wurden:
  + Das Join ist nach der Konvertierung der Spalten in ENUM schneller, aber das Verbinden der ENUM-Spalten mit VARCHAR-Spalten ist langsamer
  + Sieht nach einer guten Idee aus, solange sie nicht mit VARCHAR-Spalten verbunden werden müssen
  + Es ist eine gängige Designpraxis, "Nachschlagetabellen" (lookup tables) mit ganzzahligen Primärschlüsseln zu verwenden, um Zeichenbasierte Werte in Joins zu vermeiden
  + Die Konvertierung dieser zwei Spalten in ENUM machte die Tabelle etwa ein Drittel kleiner laut SHOW TABLE STATUS => in einigen Fällen vorteilhaft, selbst wenn die ENUM-Spalten mit VARCHAR-Spalten verbunden werden müssen
  + Der Primärschlüssel selbst ist nach der Konvertierung etwa halb so groß => auch vorteilhaft für die Indizes

**Datums- und Zeit-Typen (date and time types)**

* MySQL bietet viele Typen, z. B. YEAR und DATE
* Die feinste Granularität der Zeit, die MySQL speichern kann, beträgt eine Sekunde
* Es kann jedoch temporale Berechnungen (temporal computations) mit Mikrosekundengenauigkeit durchführen
* Die meisten temporalen Typen haben keine Alternativen, daher stellt sich nicht die Frage, welcher der beste Typ ist => die einzige Frage ist, wie man Datum und Uhrzeit zusammen speichert: DATE, TIME, TIMESTAMP
* **DATETIME**
  + Kann einen großen Wertebereich von 1001 bis 9999 speichern, mit einer Genauigkeit von einer Sekunde
  + Speichert Datum und Zeit als Integer im Format YYYYMMDDHHMMSS, unabhängig von der Zeitzone
  + Benötigt 8 Byte Speicherplatz
  + MySQL zeigt DATETIME-Werte standardmäßig in einem sortierbaren, eindeutigen Format an, z. B. 2008-01-16 22:37:08
* **TIMESTAMP**
  + Speichert die Anzahl der Sekunden, die seit Mitternacht, dem 1. Januar 1970 (GMT), vergangen sind
  + Verwendet nur 4 Byte Speicherplatz, hat daher einen kleineren Wertebereich von 1970 bis etwa Mitte 2038
  + Der angezeigte Wert eines TIMESTAMP hängt auch von der Zeitzone ab
  + Besitzt spezielle Eigenschaften, die DATETIME nicht hat => diese ändern sich mit jeder MySQL-Version:
    - MySQL setzt standardmäßig die erste TIMESTAMP-Spalte auf die aktuelle Zeit, wenn Sie eine neue Zeile einfügen => gilt auch für Updates
    - Das Verhalten bei Einfügen und Aktualisieren kann für jede TIMESTAMP-Spalte konfiguriert werden
  + TIMESTAMP-Spalten sind standardmäßig NOT NULL
* Wenn Sie TIMESTAMP verwenden können, sollten Sie dies tun, da es speichereffizienter ist
* Für die Speicherung von Mikrosekunden müssen Sie BIGINT oder DOUBLE verwenden

**BIT-Typ**

* Der BIT-Typ sollte vermieden werden, um einen oder mehrere Wahr/Falsch-Werte (true/false values) in einer einzigen Spalte zu speichern
* Wenn Sie viele Wahr/Falsch-Werte speichern müssen, sollten Sie erwägen, mehrere Spalten in einer Spalte mit MySQLs nativem SET-Datentyp (SET data type) zu kombinieren => Änderung der Spaltendefinition ist nur mit ALTER TABLE möglich (sehr aufwendig und Indizes können nicht verwendet werden)
* Bitweise Operationen (bitwise operations) auf Integer-Spalten:
  + Es ist besser, die „Aufzählung“ (enumeration) zu ändern, erfordert jedoch ein gutes Verständnis von bitweisen Manipulationen (bitwise manipulations)

**Auswahl von Identifikatoren (Choosing identifiers)**

* Die Wahl eines geeigneten Datentyps für eine Identifikatorspalte ist sehr wichtig, da diese Spalten häufiger mit anderen Werten verglichen, für Lookups verwendet oder als Fremdschlüssel (foreign keys) in anderen Tabellen genutzt werden
* Dabei müssen Sie nicht nur den Speichertyp, sondern auch die Art und Weise berücksichtigen, wie MySQL Berechnungen und Vergleiche mit diesem Typ durchführt
* Beispielsweise sind ENUM- und SET-Typen intern Ganzzahlen, werden jedoch in Zeichenketten (strings) umgewandelt, wenn sie in einem String-Kontext verglichen werden
* Wenn Sie einen Typ auswählen, stellen Sie sicher, dass derselbe Typ in allen zugehörigen Tabellen verwendet wird => Typen sollten exakt übereinstimmen, einschließlich Eigenschaften wie UNSIGNED
  + Andernfalls können Leistungsprobleme auftreten oder schwer auffindbare Fehler entstehen, selbst wenn es funktioniert
* Nochmals: Wählen Sie die kleinste Größe, die Ihren erforderlichen Wertebereich abdecken kann, und lassen Sie Platz für zukünftiges Wachstum, falls nötig
* INTEGER ist normalerweise die beste Wahl für Identifikatoren, da er schnell ist und mit AUTO\_INCREMENT funktioniert
* Vermeiden Sie ENUM- oder SET-Typen sowie String-Typen aufgrund von Leistungsproblemen und Speicherplatzbedarf (bei Strings)
* Schlecht geschriebene Schema-Migrationsprogramme (schema migration programs) und Programme, die Schemata automatisch generieren, können erhebliche Leistungsprobleme verursachen
* Anwendungen, die „Komplexität vor Entwicklern verbergen“, skalieren normalerweise schlecht => überlegen Sie sorgfältig, bevor Sie Leistung gegen Produktivität der Entwickler eintauschen, und testen Sie mit realistisch großen Datensätzen, damit Sie Leistungsprobleme nicht zu spät entdecken
* MySQL-spezifische Probleme:
  + Zu viele Spalten, zu viele Joins (joins): Als grobe Faustregel ist es besser, pro Abfrage ein Dutzend oder weniger Tabellen zu haben, wenn Sie sehr schnelle Abfragen mit hoher Parallelität benötigen
  + Falsche Verwendung von ENUM
  + Verwenden Sie NULL, wenn Sie nicht wissen, wie Sie einen unbekannten Wert darstellen sollen, anstatt verschiedene Methoden zu erfinden, um NULL zu repräsentieren (wie z. B. '0000-00-00 00:00:00')

**Normalisierung und Denormalisierung**

* Es gibt normalerweise viele verschiedene Möglichkeiten, gegebene Daten darzustellen, von vollständig normalisiert bis vollständig denormalisiert und alles dazwischen
* In einer normalisierten Datenbank wird jede Tatsache einmal und nur einmal dargestellt
* In einer denormalisierten Datenbank werden Informationen dupliziert oder an mehreren Stellen gespeichert
* Menschen mit Leistungsproblemen wird empfohlen, ihre Schemata zu normalisieren, insbesondere bei schreibintensiven Workloads

Positive Aspekte der Normalisierung:

* Normalisierte Updates sind in der Regel schneller als denormalisierte Updates
* Wenn die Daten gut normalisiert sind, gibt es weniger Daten zu ändern
* Normalisierte Tabellen sind in der Regel kleiner, sodass sie besser in den Speicher passen und eine bessere Leistung bieten
* Das Fehlen redundanter Daten bedeutet, dass weniger DISTINCT- oder GROUP-BY-Abfragen benötigt werden, um Wertelisten abzurufen
* Beispiel: Es ist unmöglich, eine eindeutige Liste von Abteilungen aus dem denormalisierten Schema ohne DISTINCT oder GROUP BY zu erhalten, aber wenn DEPARTMENT eine separate Tabelle ist, ist es eine triviale Abfrage
* Jede nicht triviale Abfrage in einem gut normalisierten Schema erfordert wahrscheinlich mindestens einen Join, möglicherweise mehrere
  + Diese sind nicht nur teuer, sondern können auch einige Index-Strategien unmöglich machen => Normalisierung kann Spalten in verschiedene Tabellen verschieben, die von demselben Index profitieren würden

Denormalisierte Tabellen:

* Können Leistungsverbesserungen bringen, da weniger Lookups und Joins erforderlich sind
* Vollständig (de-)normalisierte Schemata sind wie Laborratten => haben wenig mit der realen Welt zu tun
* Häufig müssen Sie die Ansätze mischen, möglicherweise ein teilweise normalisiertes Schema, Cache-Tabellen oder andere Techniken verwenden

Mögliche Lösungen:

* Manchmal ist die beste Möglichkeit, die Leistung zu verbessern, redundante Daten in derselben Tabelle wie die Daten zu halten, aus denen sie abgeleitet wurden
* In anderen Fällen müssen Sie vollständig separate Zusammenfassungs- (summary) oder Cache-Tabellen erstellen, die speziell für Ihre Abrufanforderungen optimiert sind

„Cache-Tabellen“ (cache tables):

* Beziehen sich auf Tabellen, die Daten enthalten, die aus dem Schema leicht, aber langsamer abgerufen werden können (d. h. logisch redundante Daten)
* Sind nützlich zur Optimierung von Such- und Abrufabfragen
* Erfordern möglicherweise viele verschiedene Index-Kombinationen, um verschiedene Abfragtypen zu beschleunigen
* Konfliktanforderungen erfordern manchmal das Erstellen einer Cache-Tabelle, die nur einige der Spalten aus der Haupttabelle enthält

„Zusammenfassungstabellen“ (summary tables):

* Halten aggregierte Daten aus GROUP-BY-Abfragen
* Beispiel: Eine Zählung (count) der Anzahl von Nachrichten, die in den letzten 24 Stunden gepostet wurden. Eine Zusammenfassungstabelle, die jede Stunde neu generiert wird, wäre nützlich => Zählungen sind nicht zu 100 % genau, aber nahezu, da es teuer ist, diese Statistiken in Echtzeit zu berechnen
* Beim Neuaufbau von Zusammenfassungs- und Cache-Tabellen müssen deren Daten oft während der Operation verfügbar bleiben => erreichbarer durch Verwendung einer „Schatten-Tabelle“ (shadow table), die „hinter“ der echten Tabelle aufgebaut wird

**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

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

* Oft sind zusätzliche Indizes, redundante Felder oder sogar Cache- und Zusammenfassungstabellen erforderlich, um Leseabfragen zu beschleunigen
* Gleichzeitig erhöhen sie die Arbeitsbelastung für Schreibabfragen und Wartungsaufgaben
* Für schnellere Leseabfragen erhöhen Sie auch die Entwicklungs-Komplexität sowohl für Lese- als auch für Schreiboperationen
  + Trotzdem ist dies eine Technik, die häufig verwendet wird, wenn Sie eine Hochleistungs-Datenbank entwerfen

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

* Die Leistung von ALTER TABLE kann bei sehr großen Tabellen ein Problem darstellen
* MySQL führt die meisten Änderungen wie folgt aus:
  + Erstellt eine neue leere Tabelle mit der gewünschten Struktur
  + Fügt alle Daten aus der alten Tabelle in die neue ein
  + Löscht die alte Tabelle
* Dies kann viel Zeit in Anspruch nehmen, besonders wenn der Arbeitsspeicher knapp ist, die Tabelle groß ist und viele Indizes enthält → in einigen Fällen kann dies Tage dauern
* Ab MySQL 5.1 gibt es Unterstützung für einige Arten von „Online“-Operationen, die die Tabelle während der gesamten Operation nicht sperren
* Für den allgemeinen Fall: Verwenden Sie entweder operationelle Tricks (wie das Umstellen von Servern und das Ausführen von ALTER TABLE auf einem Nicht-Produktionsserver) oder die „Schattenkopie“-Methode:
  + Erstellen Sie eine neue Tabelle mit der gewünschten Struktur neben der bestehenden und führen Sie dann ein Umbenennen und Löschen durch, um die beiden Tabellen zu tauschen
* Teuerste Methode, um den Standardwert einer Spalte zu ändern: MODIFY COLUMN rental\_duration TINYINT(3) NOT NULL DEFAULT 5;
* Eine kürzere und einfachere Methode: ALTER COLUMN rental\_duration SET DEFAULT 5;
  + Der Standardwert für die Spalte wird in der .frm-Datei der Tabelle gespeichert, sodass Sie ihn ändern können, ohne die Tabelle selbst zu berühren
* Mögliche Operationen ohne Tabellen-Neuaufbau:
  + Entfernen (aber nicht Hinzufügen) des AUTO\_INCREMENT-Attributs einer Spalte
  + Hinzufügen, Entfernen oder Ändern von ENUM- und SET-Konstanten
* Folgende Technik:
  1. Erstellen Sie eine leere Tabelle mit exakt derselben Struktur, abgesehen von der gewünschten Änderung (z. B. hinzugefügene ENUM-Konstanten)
  2. Führen Sie FLUSH TABLES WITH READ LOCK aus. Dadurch werden alle verwendeten Tabellen geschlossen und verhindert, dass andere Tabellen geöffnet werden
  3. Tauschen Sie die .frm-Dateien aus
  4. Führen Sie UNLOCK TABLES aus, um die Lesesperre zu lösen
* Es gibt auch ein Beispiel, wie genau diese Technik funktioniert (siehe Seite 143)

**Indexierung für hohe Leistung**

* Indizes = Datenstrukturen, die von Speicher-Engines verwendet werden, um Zeilen schnell zu finden
* Sie sind entscheidend für gute Leistung und werden umso wichtiger, je größer Ihre Datenbank wird
* Weniger ausgelastete Datenbanken können oft auch ohne ordnungsgemäße Indizes gut funktionieren, aber die Leistung kann schnell sinken, wenn die Datenmenge wächst
* „Index-Optimierung ist vielleicht der leistungsstärkste Weg, die Abfrageleistung zu verbessern.“ – eine sehr starke Aussage (kann im BA diskutiert werden)
* Das Erstellen wirklich optimaler Indizes erfordert oft, dass Sie die Abfragen umschreiben

**Grundlagen der Indexierung**

* Der einfachste Weg, das zu verstehen, ist, an den Index in einem Buch zu denken. Um zu finden, wo ein bestimmtes Thema im Buch besprochen wird, schauen Sie im Index nach, und dieser zeigt Ihnen die Seitenzahl(en), auf denen der Begriff vorkommt
* In MySQL verwendet eine Speicher-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
* Beispiel: SELECT first\_name FROM sakila.actor WHERE actor\_id = 5;
  + Es gibt einen Index auf der Spalte actor\_id, sodass MySQL den Index verwendet, um Zeilen zu finden, deren actor\_id gleich 5 ist – mit anderen Worten: Es wird eine Suche in den Werten des Index durchgeführt und alle Zeilen, die den angegebenen Wert enthalten, werden zurückgegeben
* Ein Index enthält Werte aus einer oder mehreren Spalten einer Tabelle
* Bei mehreren Spalten ist 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 nicht dasselbe wie das Erstellen von zwei einzelnen Spalten-Indizes
* Wenn Sie ein objekt-relationales Mapping (ORM) Tool verwenden, müssen Sie dennoch etwas über Indizes lernen
* Es gibt viele verschiedene Index-Typen, die jeweils für unterschiedliche Zwecke optimiert sind
* Indizes werden in der Speicher-Engine-Ebene implementiert, nicht in der Server-Ebene – daher sind sie nicht standardisiert und unterscheiden sich je nach Engine (nicht alle Engines unterstützen alle Index-Typen)

**B-Baum-Indizes (B-Tree Indexes)**

* Verwendet eine B-Baum-Datenstruktur, um seine Daten zu speichern
* Die meisten MySQL-Speicher-Engines unterstützen diesen Indextyp
* Der Name „B-Baum“ kommt von der Datenstruktur, die MySQL in CREATE TABLE und anderen Anweisungen verwendet
* Die Implementierung und Nutzung von B-Baum-Indizes variiert je nach Speicher-Engine
* 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 Speicher-Engine nicht die ganze Tabelle durchsuchen muss, um die gewünschten Daten zu finden
  + Stattdessen beginnt sie beim Wurzelknoten (der hier nicht gezeigt wird)
  + Die Slots im Wurzelknoten enthalten Zeiger auf Kind-Knoten, und die Speicher-Engine folgt diesen Zeigern
  + Der richtige Zeiger wird gefunden, indem die Werte in den Knoten-Seiten betrachtet werden, die die oberen und unteren Grenzen der Werte in den Kind-Knoten definieren
  + Ein Bild, das Text, Diagramm, Reihe, Screenshot enthält.

    Automatisch generierte BeschreibungSchließlich stellt die Speicher-Engine entweder fest, dass der gewünschte Wert nicht existiert, oder sie erreicht erfolgreich eine Blatt-Seite
  + **Blattseiten sind besonders**, weil sie Zeiger auf die indexierten Daten enthalten, anstatt Zeiger auf andere Seiten
  + Zwischen dem Wurzelknoten und den Blattseiten können viele Ebenen von Knoten-Seiten existieren. Die Tiefe des Baums hängt von der Größe der Tabelle ab
* **B-Bäume speichern die indexierten Spalten in Reihenfolge**, sodass sie für die Suche nach Datenbereichen nützlich sind
  + Beispiel: Das Durchlaufen des Baums für einen Index auf einem Textfeld passiert die Werte in alphabetischer Reihenfolge. 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 sind, z. B. der Primärschlüssel (last\_name, first\_name, dob)
* B-Baum-Indizes funktionieren gut für Suchen mit dem vollständigen Schlüsselswert, einem Schlüsselsbereich oder einem Schlüsselprefix. Sie sind jedoch nur dann nützlich, wenn die Suche das **linkeste Präfix** des Indexes verwendet

**Arten von Abfragen, bei denen B-Baum-Indizes hilfreich sind:**

* **Vollständige Übereinstimmung mit dem Wert**: Eine Übereinstimmung mit dem vollständigen Schlüsselswert gibt Werte für alle Spalten im Index an. Beispiel: Cuba Allen, geboren am 1960-01-01
* **Übereinstimmung mit einem linken Präfix**: Verwendet nur die erste Spalte im Index. Beispiel: Allen
* **Übereinstimmung mit einem Spaltenpräfix**: Der erste Teil des Werts einer Spalte, z. B. Nachnamen, die mit „M“ beginnen
* **Übereinstimmung mit einem Bereich von Werten**: Beispiel: Finden von Personen, deren Nachnamen zwischen „Allen“ und „Barrymore“ liegen
* **Übereinstimmung einer genauen Spalte und eines Bereichs auf einer anderen Spalte**: Genaues Match mit dem Nachnamen und eine Bereichsabfrage auf dem Vornamen. Beispiel: Nachname ist „Allen“ und Vorname beginnt mit „K“
* **Index-Only-Abfragen**: Abfragen, die nur den Index verwenden, nicht aber die Zeilenspeicherung (z. B. wenn alle benötigten Daten im Index enthalten sind)

**Weitere Vorteile von B-Baum-Indizes:**

* Da Baumknoten sortiert sind, können sie sowohl für **Abfragen (Werte finden)** als auch für **ORDER BY-Abfragen (Werte in sortierter Reihenfolge finden)** verwendet werden
* Wenn ein B-Baum Ihnen hilft, eine Zeile auf eine bestimmte Weise zu finden, kann er Ihnen auch helfen, Zeilen nach denselben Kriterien zu sortieren

**Einschränkungen von B-Baum-Indizes:**

* **Nicht nützlich**, wenn die Suche **nicht vom linken Ende der indexierten Spalten beginnt**. Beispiel: Wenn nur der Vorname oder das Datum abgefragt wird, ohne den Nachnamen zu verwenden
* Sie können keine Spalten im Index überspringen. Ein **spezifischer Nachname und spezifisches Datum** funktionieren nicht, es sei denn, Sie geben einen spezifischen **Vornamen** an
* Die Speicher-Engine kann den Zugriff nicht optimieren, wenn **Spalten rechts von der ersten Range-Bedingung**betroffen sind
  + Beispiel: WHERE last\_name="Smith" AND first\_name LIKE 'J%' AND dob='1976-12-23'. Der Index wird nur die ersten beiden Spalten (Nachname und Vorname) verwenden, weil der LIKE-Operator eine Bereichsbedingung ist
* Für optimale Leistung erstellen Sie Indizes mit denselben Spalten in unterschiedlichen Reihenfolgen, um Ihre Abfragen zu optimieren
  + Wenn Sie 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

**Hash-Indizes**

* Ein Hash-Index wird auf einer Hash-Tabelle (hash table) aufgebaut und ist nur für exakte Suchen nützlich, die jede Spalte im Index verwenden
  + Für jede Zeile berechnet die Speicher-Engine einen Hash-Wert (hash code) der indexierten Spalten
  + Hash-Wert = Ein kleiner Wert, der sich wahrscheinlich von den Hash-Werten anderer Zeilen mit unterschiedlichen Schlüsselwerten unterscheidet
* In MySQL unterstützt nur die Memory-Speicher-Engine explizite Hash-Indizes
* Der Standard-Indextyp für Memory-Tabellen, aber B-Tree ist ebenfalls möglich
* Die Memory-Engine unterstützt nicht-eindeutige Hash-Indizes
  + Wenn mehrere Werte denselben Hash-Wert haben, speichert der Index ihre Zeiger auf die Zeilen (row pointers) im gleichen Hash-Tabelleneintrag (hash table entry) unter Verwendung einer verketteten Liste(linked list)
* Beispiel: ***SELECT lname FROM testhash WHERE fname='Peter'***
  + MySQL berechnet den **Hash-Wert** von **'Peter'** und verwendet diesen, um den Zeiger im Index nachzuschlagen
  + f('Peter') = 8784 → MySQL schaut im Index bei der Position 8784 nach und findet den Zeiger auf Zeile 3
  + Der letzte Schritt ist, den Wert in Zeile 3 mit **'Peter'** zu vergleichen, um sicherzustellen, dass es die richtige Zeile ist
* Die Indizes selbst speichern nur kurze **Hash-Werte** → Hash-Indizes sind sehr kompakt, und **Suchen** (lookups) sind blitzschnell

**Einschränkungen (Limitations)**

* Der Index enthält nur Hash-Werte (hash codes) und Zeiger auf Zeilen (row pointers) und nicht die Werte selbst. Daher kann MySQL die Werte im Index nicht nutzen, um das Lesen der Zeilen zu vermeiden → Der Zugriff auf die in den Speicher geladenen Zeilen ist jedoch sehr schnell, sodass dies die Leistung nicht beeinträchtigt
* Hash-Indizes können nicht für Sortierungen (sorting) verwendet werden → Sie werden nicht in einer sortierten Reihenfolge gespeichert
* Sie unterstützen keine partielle Schlüsselübereinstimmung (partial key matching), weil der Hash-Wert aus dem gesamten indexierten Wert berechnet wird → Wenn du (A, B) hast und die WHERE-Klausel sich nur auf A bezieht, wird der Hash-Index nicht helfen
* Hash-Indizes unterstützen nur Gleichheitsvergleiche (equality comparisons) wie =, <=> oder IN()-Operatoren → Sie beschleunigen keine Bereichsabfragen (range queries)
* Der Zugriff auf Daten in einem Hash-Index ist sehr schnell, es sei denn, es gibt viele Kollisionen (collisions) (mehrere Werte mit demselben Hash-Wert) → Die Speicher-Engine muss jedem Zeiger auf die Zeile (row pointer) in der verketteten Liste folgen und deren Werte mit dem gesuchten Wert vergleichen, um die richtige Zeile(n) zu finden
* Einige Index-Wartungsoperationen (index maintenance operations) können langsam sein, wenn es viele Hash-Kollisionen gibt. Zum Beispiel, wenn du einen Index auf einer Spalte mit sehr niedriger Selektivität (low selectivity) erstellst (viele Hash-Kollisionen) und eine Zeile aus der Tabelle löschst → Das Finden des Zeigers im Index kann teuer sein
  + All diese Einschränkungen machen Hash-Indizes nur in speziellen Fällen nützlich
  + Wenn sie den Bedürfnissen der Anwendung entsprechen, können sie die Leistung dramatisch verbessern → z. B. in DWH (Data Warehousing) mit dem Stern-Schema (star-schema), das viele Joins erfordert
  + Einige Speicher-Engines (wie InnoDB) können feststellen, dass bestimmte Index-Werte sehr häufig verwendet werden, und bauen einen Hash-Index für diese Werte im Speicher (memory) auf, der zusätzlich zu den B-Tree-Indizes verwendet wird

**Eigene Hash-Indizes erstellen**

* Du kannst **Hash-Indizes** emulieren → Dies gibt dir Zugang zu einigen der gewünschten Eigenschaften von Hash-Indizes, wie zum Beispiel einer sehr kleinen Indexgröße für sehr lange Schlüssel
* **Idee**
  1. Erstelle einen **Pseudo-Hash-Index** auf einem Standard **B-Tree-Index**
  2. Der B-Tree-Index wird weiterhin für **Suchen** (lookups) verwendet, aber es werden die **Hash-Werte der Schlüssel** (hash values) anstelle der Schlüssel selbst verwendet
  3. Gib die **Hash-Funktion** manuell in der **WHERE-Klausel** der Abfrage an
* Funktioniert gut für **URL-Suchen** → SELECT id FROM url WHERE url="http://www.mysql.com" -> AND url\_crc=CRC32("http://www.mysql.com");
  1. Funktioniert gut, weil der **MySQL-Abfrageoptimierer** (query optimizer) bemerkt, dass es einen kleinen, hochselektiven Index auf der **url\_crc-Spalte** gibt
  2. Schneller als die Alternative, die vollständige URL als Zeichenkette zu indexieren
* Negativ: Die Notwendigkeit, die **Hash-Werte** zu pflegen
* Erstelle eine Tabelle mit der **url**- und der **url\_crc-Spalte** sowie einen **Trigger** bei Insert oder Update
* Verwende keine komplexen **Hash-Funktionen** mit langen (Zeichenketten) Rückgaben, da dies eine Verschwendung von Speicherplatz und Leistung ist → Einfache **Hash-Funktionen** können akzeptable Kollisionsraten mit besserer Leistung bieten

**Umgang mit Hash-Kollisionen**

* Wenn du nach einem Wert durch seinen **Hash** suchst, musst du auch den **literalen Wert** in deiner WHERE-Klausel angeben → Andernfalls würde es mehr als einen Wert zurückgeben, wenn es zu einer **Hash-Kollision** (hash collision) kommt
* **Hash-Kollisionen** wachsen schneller als man denkt → Das sogenannte **Geburtstagsparadoxon** (Birthday Paradox)
* **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, kannst du vereinfachen und etwas Effizienz gewinnen, indem du weniger **Bit-große** Ganzzahlen verwendest

**Spatial (R-Tree) Indizes**

* Werden mit **partiellen Typen** wie **GEOMETRY** verwendet
* Erfordern nicht, dass deine WHERE-Klauseln auf einem **linken Präfix** (leftmost prefix) des Indexes basieren
* Indizieren die Daten gleichzeitig in allen Dimensionen
* Suchen können jede beliebige Kombination von Dimensionen effizient nutzen
* Du musst die MySQL **GIS-Funktionen** (GIS functions) wie **MBRCONTAINS()** verwenden, damit dies funktioniert → MySQLs GIS-Unterstützung ist nicht groß (die meisten verwenden sie nicht)
* Die bevorzugte Lösung für **GIS** in einer Open-Source-RDBMS ist **PostGIS** in **PostgreSQL**

**Volltext-Indizes**

* Ein spezieller Indextyp, der **Schlüsselwörter** (keywords) 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
* Hat viele Feinheiten, wie **Stoppwörter** (stopwords), **Stemming** und **Pluralformen**, sowie **Boolesche Suche**(Boolean searching)
* Viel mehr analog zu dem, was eine **Suchmaschine** (search engine) tut, als zu 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

**Andere Index-Typen**

* Verschiedene **drittanbieter Speicher-Engines** (third-party storage engines) verwenden unterschiedliche Datentypen für ihre Indizes
* **TokuDB** verwendet **Fraktalbaum-Indizes** (fractal tree indexes) → Eine neu entwickelte Datenstruktur, die einige der gleichen Vorteile wie B-Tree-Indizes hat, ohne einige der Nachteile
* **InnoDB** beinhaltet **Clustered Indexes** und **Covering Indexes**
* **ScaleDB** verwendet **Patricia-Tries**, und andere Technologien wie **InfiniDB** oder **Infobright** haben ihre eigenen speziellen Datenstrukturen zur Optimierung von Abfragen

**Vorteile von Indizes**

* Ermöglichen es dem Server, schnell zu einer gewünschten Position in der Tabelle zu navigieren → Haben mehrere zusätzliche Vorteile, basierend auf den Eigenschaften der Datenstrukturen, die zu ihrer Erstellung verwendet wurden
* Für **B-Tree-Indizes** kann MySQL die sortierten Daten für Abfragen mit Klauseln wie **ORDER BY** und **GROUP BY** ausnutzen
* Da die Daten **vorgesortiert** sind, speichert ein **B-Tree-Index** auch verwandte Werte nahe beieinander
* Der Index speichert tatsächlich eine Kopie der Werte, sodass einige Abfragen direkt aus dem Index befriedigt werden können. Drei Hauptvorteile:
  + Indizes reduzieren die Menge der Daten, die der Server untersuchen muss
  + Indizes helfen dem Server, **Sortierungen** (sorting) und **temporäre Tabellen** (temporary tables) zu vermeiden
  + Indizes verwandeln zufällige **E/A** (I/O) in sequentielle **E/A**
* Lies dazu "Relational Database Index Design and the Optimizers" von **Tapio Lahdenmaki** und **Mike Leach**(Wiley) für mehr Informationen

**Ist ein Index die beste Lösung?**

* Ein Index ist **nicht immer das richtige Werkzeug**
* **Index =** am effektivsten, wenn sie der Speichermaschine helfen, Zeilen zu finden, ohne mehr Arbeit hinzuzufügen, als sie vermeiden
* Für **kleine Tabellen** ist es oft effektiver, einfach alle Zeilen in der Tabelle zu lesen
* Für **mittlere bis große Tabellen** können Indizes sehr effektiv sein
* Für **enorme Tabellen** kann der **Overhead** (Überkopfaufwand) der Indizierung sowie die Arbeit, die erforderlich 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 Tabellen zu speichern (Partitionierung in mehrere Tabellen)
* Auf **Terabyte-Skala** macht es keinen Sinn mehr, einzelne Zeilen zu lokalisieren; Indizes werden durch **Metadaten pro Block** ersetzt

**Indexierungsstrategien für hohe Leistung**

* Es gibt viele spezielle Optimierungen und spezialisierte Verhaltensweisen

**Strategie 1: Isolierung der Spalte**

* MySQL kann in der Regel keine Indizes auf Spalten verwenden, es sei denn, die Spalten sind in der Abfrage isoliert
* „**Isolierung** der Spalte“ bedeutet, dass die Spalte nicht Teil eines Ausdrucks oder einer Funktion in der Abfrage sein sollte, nicht wie folgt: WHERE actor\_id + 1 = 5;
* Ein Mensch könnte dies leicht vereinfachen, aber MySQL kann es nicht → Gewöhne dir an, deine WHERE-Kriterien zu vereinfachen, sodass die indexierte Spalte allein auf einer Seite des Vergleichsoperators steht

**Strategie 2: Prefix-Indizes und Index-Selektivität**

* Manchmal müssen sehr lange **Zeichenketten-Spalten** indexiert werden → Dadurch werden die Indizes groß und langsam
* Eine Strategie ist, einen **Hash-Index** zu verwenden, wie bereits erklärt
* Oft kann Speicherplatz gespart und eine gute Leistung erzielt werden, indem nur die ersten **Zeichen** statt des gesamten Werts indexiert werden → Weniger Speicherplatz, aber auch weniger selektiv
* **Index-Selektivität** = Verhältnis der Anzahl der **verschiedenen indexierten Werte** (die **Kardinalität**) zur Gesamtzahl der Zeilen in der Tabelle (#T), und reicht von 1/#T bis 1
* Ein **hochselektiver 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, um Speicherplatz zu sparen → Du möchtest, dass die Kardinalität des Präfixes nahe der Kardinalität der gesamten Spalte liegt
* 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 **Suffix-Indizes** 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 (+ Trigger verwenden, um den Index zu pflegen)

**Multicolumn-Indizes**

* Das Buch von Lahdenmaki und Leach führt ein Drei-Sterne-System ein, um die Eignung eines Indexes für eine Abfrage zu bewerten:
  + Ein Index erhält einen Stern, wenn er relevante Zeilen nebeneinander platziert
  + Einen zweiten Stern, wenn seine Zeilen in der Reihenfolge sortiert sind, wie sie die Abfrage benötigt
  + Einen letzten Stern, wenn er alle Spalten enthält, die für die Abfrage benötigt werden
* Häufige Fehler sind, viele oder alle Spalten separat zu indexieren oder Spalten in der falschen Reihenfolge zu indexieren
  + Der erste Fehler ist, viele Spalten separat zu indexieren → passiert oft, wenn Leute vage, aber autoritär klingende Ratschläge geben, wie zum Beispiel „Erstelle Indizes auf Spalten, die in der WHERE-Klausel erscheinen“
  + Das führt höchstens zu **Ein-Sterne-Indizes**
* Manchmal, wenn du keinen Drei-Sterne-Index entwerfen kannst, ist es viel besser, die WHERE-Klausel zu ignorieren und auf die **optimale Reihenfolge der Zeilen** zu achten oder stattdessen einen **Covering-Index** zu erstellen
* MySQL kann mit solchen schlecht indexierten Tabellen ein wenig umgehen, indem es eine Strategie namens **Index-Merge** verwendet, die es einer Abfrage ermöglicht, mehrere Indizes aus einer Tabelle zu kombinieren, um die gewünschten Zeilen zu finden

**Beispiel**

* Tabelle **film\_actor** mit Indexen auf **film\_id** und **actor\_id**:
  + Keiner ist gut geeignet für beide WHERE-Bedingungen
  + In älteren Versionen von MySQL würde dies einen **Tabellen-Scan** zur Folge haben, es sei denn, die Abfrage wird als **UNION** zweier Abfragen geschrieben
  + In MySQL 5.0 oder neuer kann die Abfrage beide Indizes nutzen, indem sie sie gleichzeitig scannt und die Ergebnisse zusammenführt
* Drei Varianten des Algorithmus:
  + **Union** für OR-Bedingungen
  + **Intersection** für AND-Bedingungen
  + **Unions von Intersections** für Kombinationen beider
* Wenn der Server Indizes **intersektiert** (meistens für AND-Bedingungen), bedeutet das normalerweise, dass du einen einzelnen Index mit allen relevanten Spalten benötigst, anstatt mehrere Indizes zu kombinieren
* Wenn der Server Indizes **vereint** (meistens für OR-Bedingungen), können die Puffer-, Sortier- und Zusammenführungsoperationen des Algorithmus viele CPU- und Speicherressourcen beanspruchen → Besonders dann, wenn nicht alle Indizes sehr selektiv sind, sodass die Scans viele Zeilen zurückgeben, was die Zusammenführungsoperation belastet
* Der **Optimierer** berücksichtigt diese Kosten nicht → er optimiert nur die Anzahl der zufälligen Seitenlesungen → dies kann dazu führen, dass die Abfrage als schneller eingeschätzt wird als ein vollständiger Tabellen-Scan, obwohl sie in Wirklichkeit langsamer ist
* Manchmal ist es effizienter, solche Abfragen umzuschreiben, indem man ein **UNION** verwendet, wie man es früher in MySQL 4.1 und älteren Versionen tun musste
* Du kannst Index-Merges mit der Option **optimizer\_switch** oder der entsprechenden Variablen deaktivieren oder auch den **IGNORE INDEX**-Befehl verwenden

**Choosing a good column order**

* Eine der häufigsten Ursachen für Verwirrung
* Die korrekte Reihenfolge hängt von den Abfragen ab, die den Index verwenden
* Wähle die Indexreihenfolge so, dass Zeilen in einer Weise sortiert und gruppiert werden, die der Abfrage zugutekommt
* Gilt nur für **B-Tree-Indexes** → Hash- und andere Indexarten speichern ihre Daten nicht in sortierter Reihenfolge
* 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
* Platziere die **selektivsten Spalten** zuerst im Index → dies kann sinnvoll sein, wenn keine Sortierung oder Gruppierung zu berücksichtigen ist → der Zweck ist nur die Optimierung von **WHERE-Lookups**
* Hängt auch von den tatsächlichen Werten ab, mit denen du Zeilen suchst (→ die Verteilung der Werte)
* Wähle die Spaltenreihenfolge so, dass sie für die Abfragen, die du am häufigsten ausführst, so selektiv wie möglich ist

**Clustered-Indexes**

* Kein separater Index-Typ, sondern eine Methode der Datenspeicherung
* InnoDB’s **clustered indexes** speichern einen **B-Tree-Index** und die Zeilen zusammen in derselben Struktur
* „Clustered“ bezieht sich darauf, dass Zeilen mit benachbarten Schlüsselwerten nahe beieinander gespeichert werden
* Wenn eine Tabelle einen **clustered index** hat, werden die Zeilen in den **Blattseiten** des Index gespeichert → jede Tabelle kann nur einen clustered index haben (es können keine Zeilen an zwei Orten gleichzeitig gespeichert werden)
* **Covering-Indexes** ermöglichen es, mehrere **clustered indexes** zu emulieren
* Nicht alle **Storage Engines** unterstützen clustered indexes
* Blattseiten enthalten vollständige Zeilen → Knoten-Seiten enthalten nur die indexierten Spalten
* Einige Datenbank-Server ermöglichen es dir, den Index auszuwählen, nach dem die Tabelle geclustert wird, andere verwenden einfach den **Primärschlüssel** (InnoDB definiert sogar einen versteckten PK, wenn kein einzigartiger nicht nullbarer Index vorhanden ist)
* Das Clustern von **PKs** kann die Performance verbessern, kann jedoch auch ernsthafte Performance-Probleme verursachen

**Vorteile des Clusterns von Daten:**

* **Verwandte Daten zusammenhalten**
* **Schnellerer Datenzugriff** → Daten werden schneller abgerufen als bei **non-clustered indexes**
* Abfragen, die **Covering-Indexes** verwenden, können die PK-Werte verwenden, die im Blattknoten enthalten sind

**Nachteile:**

* **Clustering** bietet die größten Verbesserungen für **I/O-gebundene Workloads** → Wenn die Daten in den Speicher passen, spielt die Reihenfolge keine Rolle → keine Vorteile
* Die **Einfügegeschwindigkeit** hängt stark von der Einfüge-Reihenfolge ab → Das Einfügen von Zeilen in PK-Reihenfolge ist der schnellste Weg, Daten in die Tabelle zu laden
* Es könnte eine gute Idee sein, die Tabelle nach dem Laden großer Datenmengen mit **OPTIMIZE TABLE** neu zu organisieren, wenn du die Zeilen nicht in PK-Reihenfolge geladen hast
* **Aktualisierungen** des **clustered index** sind teuer → Sie zwingen dazu, jede aktualisierte Zeile an einen neuen Speicherort zu verschieben
* Tabellen sind anfällig für **Page Splits**, wenn neue Zeilen eingefügt werden → Eine Seite muss in zwei Teile gespalten werden, um die Zeile unterzubringen, was dazu führen kann, dass die Tabelle mehr Speicherplatz auf der Festplatte benötigt
* Clustered-Tabellen können bei **Full-Table-Scans** langsamer sein, besonders wenn die Zeilen weniger dicht gepackt oder nicht sequenziell gespeichert sind aufgrund von Page Splits
* **Sekundäre (non-clustered) Indizes** können größer sein als erwartet, weil ihre Blattknoten die Primärschlüsselspalten der referenzierten Zeilen enthalten
* Der Zugriff auf sekundäre Indizes erfordert **zwei Index-Lookups** anstelle von einem → weil ein Blattknoten keinen **Zeilen-Pointer** auf den physischen Standort der referenzierten Zeile speichert; er speichert stattdessen die Primärschlüsselwerte der Zeile

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

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

**MyISAM’s Datenlayout:**

* Speichert die Zeilen auf der Festplatte in der Reihenfolge, in der sie eingefügt wurden
* Zeilen sind **fest in der Größe**, 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
* Kein struktureller Unterschied zwischen einem **Primärschlüssel** oder einem anderen Index → Der PK ist einfach ein einzigartiger, nicht nullbarer Index mit dem Namen **PRIMARY**

**InnoDB’s Datenlayout:**

* Speichert dieselben Daten aufgrund seiner **clustered Organization** sehr unterschiedlich
* Der **clustered index** „ist“ die Tabelle in InnoDB, 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 → 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**

* Hohe Parallelitätslasten (high-concurrency workloads): Das Einfügen von Daten in Primärschlüsselreihenfolge (primary key order) kann 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 (concurrent inserts) um next-key Locks konkurrieren
* Ein weiteres Problem ist der AUTO\_INCREMENT-Sperrmechanismus, der die Konkurrenz bei Multi-Thread-Szenarien weiter verstärken kann
* Wenn diese Probleme auftreten, könnten Lösungen sein:
  + Neu-Design der Tabelle oder Anwendung, um Einfügungen gleichmäßiger zu verteilen
  + 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
* MySQL muss 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 dich nicht automatisch vor Fehlern schützen und die Leistung nicht verbessern, sondern unnötige Arbeit verursachen
* **Best Practice:**
  + Vermeide das Erstellen doppelter Indizes
  + Falls doppelte Indizes bereits existieren, sollten diese entfernt werden, um die Leistung zu verbessern und unnötigen Overhead zu reduzieren