**Replication**

**Themen:**

* Überblick
* Grundlegende Server-Einrichtung
* Entwurf fortgeschrittener Replikationskonfigurationen
* Verwaltung und Optimierung Ihrer replizierten Server

**Einleitung:**

* Grundlage für den Aufbau großer, leistungsstarker Anwendungen auf MySQL-Basis unter Verwendung der sogenannten "Scale-Out"-Architektur
* Replikation ermöglicht es, einen oder mehrere Server als Replikas (oder Slaves) eines anderen Servers zu konfigurieren
* Hält deren Daten mit der Masterkopie synchron
* Nicht nur nützlich für Hochleistungsanwendungen => auch ein Eckpfeiler vieler Strategien für hohe Verfügbarkeit, Skalierbarkeit, Notfallwiederherstellung, Backups, Analysen, Data Warehousing und viele andere Aufgaben
* Genauso wichtig sind Korrektheit und Zuverlässigkeit in Bezug auf Replikation wie auch die Leistung

**Überblick:**

* Das grundlegende Problem, das die Replikation löst, besteht darin, die Daten eines Servers mit denen eines anderen synchron zu halten
* Viele Replikas können sich mit einem einzigen Master verbinden und mit diesem synchron bleiben, und eine Replika kann als Master fungieren
* Master und Replikas können in vielen verschiedenen Konfigurationen angeordnet werden
* MySQL unterstützt zwei Arten von Replikation: statement-based replication (Anweisungsbasierte Replikation, seit MySQL 3.23) und row-based replication (Zeilenbasierte Replikation, seit MySQL 5.1)
* Beide Arten funktionieren, indem Änderungen im binären Log des Masters aufgezeichnet und im Replika wiedergegeben werden
* Beide sind asynchron => die Datenkopie der Replika ist zu keinem bestimmten Zeitpunkt garantiert auf dem neuesten Stand
* Es gibt keine Garantie dafür, wie groß die Latenz auf der Replika sein könnte
* Große Abfragen können dazu führen, dass die Replika Sekunden, Minuten oder sogar Stunden hinter dem Master zurückbleibt
* Die Replikation von MySQL ist größtenteils abwärtskompatibel => ein neuerer Server kann in der Regel problemlos als Replika eines älteren Servers fungieren
* Ältere Serverversionen können jedoch oft nicht als Replikas neuerer Versionen dienen, da sie möglicherweise neue Funktionen oder die SQL-Syntax des neueren Servers nicht verstehen und es Unterschiede in den von der Replikation verwendeten Dateiformaten geben kann
* Zum Beispiel können Sie nicht von einem MySQL 5.1 Master zu einer MySQL 4.0 Replika replizieren
* Es ist ratsam, Ihre Replikationskonfiguration zu testen, bevor Sie von einer Haupt- oder Nebenversion auf eine andere aktualisieren
* Upgrades innerhalb einer Nebenversion, wie z.B. von 5.1.51 auf 5.1.58, sind in der Regel kompatibel
* Die Replikation fügt dem Master im Allgemeinen nicht viel Overhead hinzu
* Sie erfordert, dass das binäre Logging auf dem Master aktiviert ist, was einen erheblichen Overhead verursachen kann => dies ist jedoch ohnehin für ordnungsgemäße Backups und Point-in-Time-Recovery erforderlich
* Abgesehen vom binären Logging fügt jede angeschlossene Replika während des normalen Betriebs eine geringe Last (hauptsächlich Netzwerk-I/O) auf den Master hinzu
* Wenn Sie eine sehr hohe Arbeitslast (z.B. 5.000 oder mehr Transaktionen pro Sekunde) auf viele Replikas replizieren, kann der Overhead, alle Replika-Threads zu aktivieren, um ihnen die Ereignisse zu senden, erheblich werden
* Replikation eignet sich relativ gut zum Skalieren von Lesevorgängen, die Sie an eine Replika weiterleiten können, aber es ist keine gute Methode, um Schreibvorgänge zu skalieren, es sei denn, Sie entwerfen es entsprechend
* Das Anbinden vieler Replikas an einen Master führt einfach dazu, dass die Schreibvorgänge mehrfach ausgeführt werden, jeweils einmal auf jeder Replika. Das gesamte System ist durch die Anzahl der Schreibvorgänge begrenzt, die der schwächste Teil ausführen kann
* Replikation ist auch verschwenderisch, wenn mehr als nur wenige Replikas vorhanden sind, da sie im Wesentlichen viele Daten unnötig dupliziert
* Zum Beispiel hat ein einzelner Master mit 10 Replikas 11 Kopien derselben Daten und dupliziert die meisten dieser Daten in 11 verschiedenen Caches

**Probleme, die durch Replikation gelöst werden**

* Datenverteilung
  + Die Replikation von MySQL ist im Allgemeinen nicht bandbreitenintensiv, obwohl die zeilenbasierte Replikation (row-based replication) in MySQL 5.1 mehr Bandbreite verbrauchen kann als die statementsbasierte Replikation (statement-based replication)
  + Sie ist nützlich, um entfernte Datenkopien zu pflegen, selbst über intermittierende Verbindungen, aber eine geringe Replikationsverzögerung erfordert eine stabile, latenzarme Verbindung
* Lastverteilung
  + Die MySQL-Replikation ermöglicht es, Leseabfragen auf mehrere Server zu verteilen, was sie ideal für leselastige Anwendungen macht, wobei eine einfache Lastverteilung durch einfache Codeänderungen erreicht werden kann
  + Lastenverteilung kann mit Methoden von festcodierten Hostnamen und Round-Robin-DNS bis hin zu fortgeschrittenen Lösungen wie Netzwerkloadbalancern und dem Linux Virtual Server (LVS) Projekt verwaltet werden
* Sicherungen
  + Replikation ist eine wertvolle Technik zur Unterstützung von Sicherungen
  + Ein Replikat ist jedoch weder ein Backup noch ein Ersatz für Backups
* Hohe Verfügbarkeit und Failover
  + Replikation kann helfen, MySQL in Ihrer Anwendung zu vermeiden, dass es zu einem einzigen Ausfallpunkt (Single Point of Failure) wird
  + Eine gute Failover-Lösung mit Replikation kann helfen, die Ausfallzeiten erheblich zu reduzieren
* Testing von MySQL-Updates
  + Es ist gängige Praxis, ein Replikat mit einer aktualisierten MySQL-Version einzurichten und es zu verwenden, um sicherzustellen, dass Ihre Abfragen wie erwartet funktionieren, bevor jede Instanz aktualisiert, wird

**Wie Replikation funktioniert**

1. Der Master zeichnet Änderungen an seinen Daten in seinem Binärlog auf (Diese Aufzeichnungen werden als Binärlog-Ereignisse bezeichnet)
2. Der Replikat kopiert die Binärlog-Ereignisse des Masters in sein Relay-Log
3. Der Replikat spielt die Ereignisse im Relay-Log ab und wendet die Änderungen auf seine eigenen Daten an

* Der erste Teil des Prozesses ist das Binärlogging auf dem Master
* Unmittelbar bevor jede Transaktion, die Daten aktualisiert, auf dem Master abgeschlossen wird, zeichnet der Master die Änderungen in seinem Binärlog auf
* MySQL schreibt Transaktionen seriell im Binärlog, auch wenn die Anweisungen in den Transaktionen während der Ausführung miteinander vermischt wurden
* Nach dem Schreiben der Ereignisse in das Binärlog teilt der Master dem Storage Engine(s) mit, die Transaktionen zu committen
* Der nächste Schritt ist, dass der Replikat das Binärlog des Masters auf seine eigene Festplatte kopiert, in das sogenannte Relay-Log
* Er startet einen Worker-Thread, der als I/O-Slave-Thread bezeichnet wird, der eine gewöhnliche Client-Verbindung zum Master öffnet und dann einen speziellen Binlog-Dump-Prozess startet
* Der Binlog-Dump-Prozess liest Ereignisse aus dem Binärlog des Masters
* Wenn er mit dem Master aufholt, geht er in den Schlafmodus und wartet darauf, dass der Master ihm signalisiert, wenn neue Ereignisse vorhanden sind
* Der I/O-Thread schreibt die Ereignisse in das Relay-Log des Replikats
* Der SQL-Slave-Thread übernimmt den letzten Teil des Prozesses
* Er liest und spielt Ereignisse aus dem Relay-Log ab und aktualisiert so die Daten des Replikats, sodass sie mit denen des Masters übereinstimmen
* Solange dieser Thread mit dem I/O-Thread Schritt hält, bleibt das Relay-Log normalerweise im Cache des Betriebssystems, sodass Relay-Logs nur sehr geringe Overhead-Kosten haben
* Die Ereignisse, die der SQL-Thread ausführt, können optional in das eigene Binärlog des Replikats geschrieben werden, was für Szenarien, die wir später erwähnen, nützlich ist
* Abbildung 10-1 zeigte nur die beiden Replikations-Threads, die auf dem Replikat laufen, aber es gibt auch einen Thread auf dem Master: Wie bei jeder Verbindung zu einem MySQL-Server startet die Verbindung, die der Replikat zum Master öffnet, einen Thread auf dem Master
* Diese Replikationsarchitektur entkoppelt die Prozesse des Abrufens und Abspielens von Ereignissen auf dem Replikat, was es ihnen ermöglicht, asynchron zu arbeiten => Der I/O-Thread kann unabhängig vom SQL-Thread arbeiten
* Das bedeutet, dass Updates, die parallel (in verschiedenen Threads) auf dem Master ausgeführt werden könnten, auf dem Replikat nicht parallelisiert werden können, da sie in einem einzelnen Thread ausgeführt werden => Performance-Engpass für viele Workloads
* Es gibt einige Lösungen dafür, aber die meisten Benutzer sind immer noch der Einschränkung auf einen einzelnen Thread unterworfen

**Einrichtung der Replikation**

* Das einfachste Szenario ist ein frisch installierter Master und Replikat
  + Einrichtung von Replikationskonten auf jedem Server
  + Konfiguration des Masters und des Replikats
  + Das Replikat anweisen, sich mit dem Master zu verbinden und von ihm zu replizieren
* Angenommen, Ihre Server heißen server1 (IP-Adresse 192.168.0.1) und server2 (IP-Adresse 192.168.0.2)

**Erstellen von Replikationskonten**

* MySQL hat einige spezielle Privilegien, die es den Replikationsprozessen ermöglichen, zu laufen
* Der Slave-I/O-Thread, der auf dem Replikat läuft, stellt eine TCP/IP-Verbindung zum Master her
* Es muss ein Benutzerkonto auf dem Master erstellt werden und diesem die richtigen Privilegien zugewiesen werden, damit der I/O-Thread sich als dieser Benutzer verbinden und das Binärlog des Masters lesen kann
  + GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.*
* Wir erstellen dieses Benutzerkonto sowohl auf dem Master als auch auf dem Replikat
  + Das Konto, das Sie zur Überwachung und Verwaltung der Replikation verwenden, benötigt das REPLICATION CLIENT Privileg, und es ist einfacher, dasselbe Konto für beide Zwecke zu verwenden (anstatt ein separates Benutzerkonto nur für diesen Zweck zu erstellen)
  + Wenn Sie das Konto auf dem Master einrichten und dann das Replikat davon klonen, wird das Replikat korrekt eingerichtet, um als Master zu agieren, falls Sie möchten, dass das Replikat und der Master die Rollen tauschen

**Konfiguration des Masters und des Replikats**

* Der nächste Schritt besteht darin, einige Einstellungen auf dem Master zu aktivieren, den wir hier server1 nennen
  + Es müssen die Binärprotokollierung aktiviert und eine Server-ID angegeben werden
* Wenn die Binärprotokollierung im Konfigurationsfile des Masters nicht bereits angegeben wurde, müssen Sie MySQL neu starten
* Es muss explizit eine einzigartige Server-ID angegeben werden => wir haben uns entschieden, 10 anstelle von 1 zu verwenden, da 1 der Standardwert ist, den ein Server normalerweise wählt, wenn kein Wert angegeben wird
* Um zu überprüfen, ob die Binärlogdatei auf dem Master erstellt wurde, führen Sie SHOW MASTER STATUS aus
* Das Replikat benötigt eine Konfiguration in seiner my.cnf-Datei, die der des Masters ähnelt, und Sie müssen MySQL auch auf dem Replikat neu starten
* Auf einem Replikat ist nur der Parameter server\_id erforderlich, aber wir haben auch log\_bin aktiviert und der Binärlogdatei einen expliziten Namen gegeben
* Zudem haben wir zwei andere optionale Konfigurationsparameter hinzugefügt: relay\_log (um den Speicherort und den Namen des Relay-Logs anzugeben) und log\_slave\_updates (um das Replikat die replizierten Ereignisse in sein eigenes Binärlog zu schreiben)
* Die letztere Option verursacht zusätzlichen Aufwand für die Replikate, aber wir haben gute Gründe, diese optionalen Einstellungen auf jedem Replikat hinzuzufügen

**Starten des Replikats**

* Der nächste Schritt besteht darin, dem Replikat zu sagen, wie es sich mit dem Master verbinden und dessen Binärlogs abspielen soll
* Sie sollten nicht die my.cnf-Datei dafür verwenden, sondern den Befehl CHANGE MASTER TO
* Dieser Befehl ersetzt die entsprechenden my.cnf-Einstellungen vollständig
* CHANGE MASTER TO MASTER\_HOST='server1',  
  -> MASTER\_USER='repl',  
  -> MASTER\_PASSWORD='p4ssword',  
  -> MASTER\_LOG\_FILE='mysql-bin.000001',  
  -> MASTER\_LOG\_POS=0;
* Die Spalten Slave\_IO\_State, Slave\_IO\_Running und Slave\_SQL\_Running zeigen an, dass die Replikationsprozesse nicht laufen
* Aufmerksame Leser werden auch bemerken, dass die Log-Position 4 statt 0 ist => 0 ist eigentlich keine Log-Position; es bedeutet einfach "am Anfang der Log-Datei"
* Um die Replikation zu starten, führen Sie den folgenden Befehl aus: START SLAVE;
* Seconds\_Behind\_Master ist nicht mehr NULL => Der I/O-Thread wartet auf ein Ereignis vom Master, was bedeutet, dass er alle Binärlogs des Masters abgerufen hat
* Wenn Sie eine Änderung auf dem Master vornehmen, sollten Sie sehen, wie die verschiedenen Datei- und Positionswerte auf dem Replikat inkrementiert werden => Sie sollten auch die Änderungen in den Datenbanken auf dem Replikat sehen
* Auf dem Replikat sollten Sie zwei Threads sehen. Einer ist der I/O-Thread und der andere der SQL-Thread: SHOW PROCESSLIST\G
* Diese Prozesse laufen immer unter dem Benutzerkonto "system user", aber die anderen Spaltenwerte können variieren

**Ein Replikat von einem anderen Server initialisieren**

* Die vorherigen Setup-Anweisungen gingen davon aus, dass Sie den Master und das Replikat mit den standardmäßigen Anfangsdaten nach einer frischen Installation gestartet haben
* In der Regel haben Sie einen Master, der schon eine Weile läuft, und möchten ein frisch installiertes Replikat mit dem Master synchronisieren, auch wenn es nicht die Daten des Masters hat
* Es gibt mehrere Möglichkeiten, ein Replikat von einem anderen Server zu initialisieren oder zu "klonen":
  + Das Kopieren von Daten vom Master
  + Das Klonen eines Replikats von einem anderen Replikat
  + Das Starten eines Replikats von einem aktuellen Backup
* Sie benötigen drei Dinge, um ein Replikat mit einem Master zu synchronisieren:
  + Eine Momentaufnahme der Daten des Masters zu einem bestimmten Zeitpunkt
  + Die aktuelle Log-Datei des Masters und der Byte-Offset innerhalb dieses Logs zum genauen Zeitpunkt der Momentaufnahme (Log-Datei-Koordinaten) => Finden Sie die Log-Datei-Koordinaten des Masters mit dem Befehl SHOW MASTER STATUS
  + Die Binärlogdateien des Masters von diesem Zeitpunkt bis zur Gegenwart
* Kalte Kopie: Beinhaltet das Herunterfahren des Masters, um seine Dateien auf das Replikat zu kopieren, dann wird der Master mit einem neuen Binärlog neu gestartet und das Replikat so konfiguriert, dass es mit diesem Log startet, wobei der Nachteil besteht, dass der Master Ausfallzeiten benötigt
* Bei einer warmen Kopie: Es können Dateien kopiert werden, während der Server noch läuft
* Verwendung von mysqldump: Für InnoDB-Tabellen verwenden Sie mysqldump --single-transaction --all-databases --master-data=1, um Daten vom Master zu dumpen, sie ins Replikat zu laden und die Koordinaten des Replikats mit den Binärlog-Koordinaten des Masters zu synchronisieren; für nicht-transaktionale Tabellen verwenden Sie --lock-all-tables für einen konsistenten Dump
* Bei einer Momentaufnahme oder einem Backup: Initialisieren Sie das Replikat, indem Sie eine Momentaufnahme oder ein Backup des Masters wiederherstellen und das Replikat mit den entsprechenden Binärlog-Koordinaten unter Verwendung von CHANGE MASTER TO konfigurieren, wobei alle Binärlogs seit dem Backup erhalten bleiben
* Von einem anderen Replikat: Ein Replikat kann unter Verwendung von Momentaufnahme- oder Kopiertechniken geklont werden, aber die Option --master-data mit mysqldump funktioniert nicht, und Sie müssen SHOW SLAVE STATUS verwenden, um die richtigen Binärlog-Koordinaten zu erhalten
  + Der größte Nachteil beim Klonen eines Replikats von einem anderen ist, dass, wenn Ihr Replikat nicht mehr mit dem Master synchron ist, Sie falsche Daten klonen
* Verwenden Sie NICHT LOAD DATA FROM MASTER oder LOAD TABLE FROM MASTER! Sie sind veraltet, langsam und sehr gefährlich => Sie funktionieren auch nur mit MyISAM

**Empfohlene Replikationskonfiguration**

* Es gibt viele mögliche Replikationsparameter, und die meisten von ihnen haben zumindest einige Auswirkungen auf Datensicherheit und Leistung
* Wir zeigen eine empfohlene, „sichere“ Replikationskonfiguration, die die Wahrscheinlichkeit von Problemen minimiert
* Die wichtigste Einstellung für das Binär-Logging auf dem Master ist **sync\_binlog**:  
  **sync\_binlog=1**
* Sorgt dafür, dass MySQL den Inhalt des Binärlogs bei jedem Transaktions-Commit auf die Festplatte synchronisiert => keine verlorenen Log-Ereignisse im Falle eines Absturzes
* Wenn Sie diese Option deaktivieren, hat der Server etwas weniger Arbeitsaufwand, aber Binärlog-Einträge könnten nach einem Serverabsturz beschädigt oder verloren sein
* Auf einem Replikat, das nicht als Master fungieren muss, erzeugt diese Option unnötigen Overhead
* Gilt nur für das Binärlog, nicht für das Relay-Log
* Wir empfehlen auch, einen Basisnamen für das Binärlog explizit anzugeben, um einheitliche Binärlog-Namen auf allen Servern zu erstellen und Änderungen der Binärlog-Namen zu vermeiden, falls sich der Hostname des Servers ändert
* Geben Sie ein Argument für die **log\_bin**-Option an, optional mit einem absoluten Pfad, aber in jedem Fall mit dem Basisnamen
* Auf dem Replikat empfehlen wir zudem die folgenden Konfigurationsoptionen zu aktivieren. Verwenden Sie auch hier einen absoluten Pfad für den Speicherort des Relay-Logs:
  + **relay\_log=/path/to/logs/relay-bin**
  + **skip\_slave\_start**
  + **read\_only**
* Die **relay\_log**-Option verhindert hostnamenbasierte Relay-Log-Dateinamen, was dieselben Probleme vermeidet, die wir bereits für den Master erwähnt haben
* Die Option **skip\_slave\_start** verhindert, dass das Replikat nach einem Absturz automatisch startet. Dies gibt Ihnen die Möglichkeit, den Server bei Problemen zu reparieren
* Die Option **read\_only** verhindert, dass die meisten Benutzer nicht-temporäre Tabellen ändern. Ausnahmen sind der Replikations-SQL-Thread und Threads mit dem **SUPER**-Privileg => vermeiden Sie es, normalen Benutzerkonten das **SUPER**-Privileg zu geben
* Ein Replikat kann nach einem Absturz leicht beschädigt werden, da die Relay-Logs und die **master.info**-Datei nicht absturzsicher sind
* Wenn ein Replikat stark hinter dem Master zurückliegt, kann der Slave-I/O-Thread viele Relay-Logs schreiben
* Der Replikations-SQL-Thread entfernt diese, sobald er mit deren Verarbeitung fertig ist (dies kann mit der Option **relay\_log\_purge** geändert werden)
* Wenn das Replikat stark im Rückstand ist, könnte der I/O-Thread tatsächlich den Speicherplatz der Festplatte füllen
* Die Lösung für dieses Problem ist die Konfigurationsvariable **relay\_log\_space\_limit**
* Wenn die Gesamtsumme aller Relay-Logs größer wird als der Wert dieser Variablen, stoppt der I/O-Thread und wartet, bis der SQL-Thread weiteren Speicherplatz freigibt
* Wenn das Replikat noch nicht alle Relay-Logs vom Master abgerufen hat, könnten diese Logs für immer verloren gehen, falls der Master abstürzt
* Sofern Sie sich keine Sorgen um den Speicherplatz machen, ist es wahrscheinlich eine gute Idee, dem Replikat so viel Speicherplatz für Relay-Logs zu erlauben, wie es benötigt => deshalb haben wir die **relay\_log\_space\_limit**-Einstellung nicht in unsere empfohlene Konfiguration aufgenommen

**Statement-basierte Replikation**

* MySQL 5.0 und älter unterstützen nur die statement-basierte Replikation (auch als logische Replikation bezeichnet)
* Funktioniert, indem die Abfrage, die die Daten auf dem Master geändert hat, protokolliert wird
* Wenn das Replikat das Ereignis aus dem Relay-Log liest und ausführt, wird die tatsächliche SQL-Abfrage erneut ausgeführt, die der Master ausgeführt hat
* Hat sowohl Vorteile als auch Nachteile
* Der offensichtlichste Vorteil ist, dass sie relativ einfach zu implementieren ist
* Das einfache Protokollieren und Wiederholen von Anweisungen, die Daten ändern, hält das Replikat theoretisch mit dem Master synchron
* Ein weiterer Vorteil der statement-basierten Replikation ist, dass die Binärlog-Ereignisse in der Regel recht kompakt sind => verbraucht nicht viel Bandbreite
* In der Praxis ist die statement-basierte Replikation jedoch nicht so einfach, wie es scheint, da viele Änderungen auf dem Master von Faktoren abhängen, die über den reinen Abfragetext hinausgehen
* Beispielsweise werden Anweisungen zu leicht oder sogar deutlich unterschiedlichen Zeiten auf dem Master und dem Replikat ausgeführt
* Deshalb enthält das Binärlog-Format von MySQL mehr als nur den Abfragetext; es überträgt auch Metadaten wie den aktuellen Zeitstempel
* Es gibt einige Anweisungen, die MySQL nicht korrekt replizieren kann, z.B. Abfragen, die die Funktion **CURRENT\_USER()** verwenden
* Gespeicherte Routinen und Trigger sind ebenfalls problematisch bei der statement-basierten Replikation
* Ein weiteres Problem ist, dass Änderungen serialisierbar sein müssen => erfordert mehr Sperren (Locks), manchmal deutlich mehr

**Row-basierte Replikation**

* MySQL 5.1 führte die Unterstützung für row-basierte Replikation ein, die die tatsächlichen Datenänderungen im Binärlog speichert und der Implementierung in den meisten anderen Datenbanksystemen ähnelt
* Die größten Vorteile sind, dass MySQL jede Anweisung korrekt replizieren kann und einige Anweisungen wesentlich effizienter repliziert werden können
* MySQL kann einige Änderungen effizienter mit row-basierter Replikation replizieren, da das Replikat die Abfragen, die die Zeilen auf dem Master geändert haben, nicht erneut ausführen muss
* Das Wiederholen bestimmter Abfragen kann sehr ressourcenintensiv sein (siehe Beispiel auf Seite 461)
* Beispielsweise scannt eine Abfrage viele Zeilen in der Quelltabelle, führt jedoch nur zu drei Zeilen in der Zieltabelle
* Wenn dieses Ereignis als Anweisung (statement-based) repliziert wird, muss das Replikat all diese Arbeit erneut ausführen, nur um ein paar Zeilen zu erstellen. Mit row-basierter Replikation ist dies jedoch trivial und effizient => in diesem Fall ist die row-basierte Replikation viel effizienter
* Andererseits ist das folgende Ereignis deutlich günstiger mit statement-basierter Replikation zu replizieren:

mysql> UPDATE enormous\_table SET col1 = 0;

* Die Verwendung von row-basierter Replikation für diese Abfrage wäre sehr teuer, da jede Zeile geändert wird
* Jede Zeile müsste ins Binärlog geschrieben werden, was das Binärlog-Ereignis extrem groß machen würde
* Dies würde sowohl beim Protokollieren als auch bei der Replikation zu einer höheren Last auf dem Master führen, und das langsamere Protokollieren könnte die Parallelität verringern
* Die Durchführung einer Point-in-Time-Wiederherstellung ist mit einem Binärlog im row-basierten Format schwieriger, aber nicht unmöglich => ein Log-Server kann hier hilfreich sein
* Da kein Format in jeder Situation perfekt ist, kann MySQL dynamisch zwischen statement-basierter und row-basierter Replikation wechseln
* Standardmäßig wird die statement-basierte Replikation verwendet, aber wenn MySQL ein Ereignis erkennt, das nicht korrekt als Statement repliziert werden kann, wechselt es automatisch zur row-basierten Replikation
* Sie können das Format auch manuell steuern, indem Sie die Session-Variable **binlog\_format** setzen

**Vorteile der Statement-basierten Replikation**

* **Flexibler bei unterschiedlichen Schemata:** Funktioniert in mehr Fällen, wenn das Schema auf dem Master und dem Replikat unterschiedlich ist
* **Kompatible Datentypen:** Kann in mehr Szenarien eingesetzt werden, bei denen Tabellen unterschiedliche, aber kompatible Datentypen, unterschiedliche Spaltenreihenfolgen usw. haben
* **Erleichtert Schema-Änderungen:** Vereinfacht das Durchführen von Schema-Änderungen auf einem Replikat, das später als Master promoted werden soll, was die Ausfallzeit reduziert
* **Mehr operative Flexibilität:** Ermöglicht flexiblere Betriebsabläufe, da Änderungen leichter gesteuert werden können

**Bessere Nachvollziehbarkeit:** Alle Änderungen erfolgen über einen bekannten Mechanismus (SQL-Anweisungen), was die Fehlersuche und das Verständnis von Problemen erleichtert

**Nachteile der Statement-basierten Replikation**

* **Flexibler bei unterschiedlichen Schemata:** Funktioniert in mehr Fällen, wenn das Schema auf dem Master und dem Replikat unterschiedlich ist
* **Bugs in älteren Versionen:** Viele Fehler im Zusammenhang mit der Replikation von Stored Procedures, Triggern usw. in den Versionen 5.0 und 5.1

**Vorteile der Row-basierten Replikation**

* **Probleme mit bestimmten SQL-Features:** Schwierigkeiten bei der Replikation von temporären Tabellen, gemischten Storage-Engines, spezifischen SQL-Konstrukten und nicht-deterministischen Anweisungen – diese Probleme reichen von ärgerlich bis kritisch
* **Höhere Kompatibilität:** Funktioniert zuverlässig mit allen SQL-Konstrukten, es gibt weniger Fälle, in denen es zu Problemen kommt
* **Stabil bei komplexen Änderungen:** Scheitert in der Regel nur bei komplexen Aufgaben wie Schema-Änderungen auf dem Replikat
* **Weniger Sperrprobleme:** Ermöglicht reduziertes Locking, da keine strikte Serialisierung für die Wiederholbarkeit erforderlich ist
* **Präzise Änderungsprotokollierung:** Protokolliert die tatsächlich geänderten Daten, nicht nur die Anweisung => man muss nicht raten, ob eine Abfrage Daten geändert hat
* **Unterstützung bei der Datenwiederherstellung:** In einigen Fällen werden auch die alten Daten gespeichert, was die Wiederherstellung von Daten erleichtern kann
* **Weniger CPU-Last:** Oft weniger CPU-intensiv, da keine komplexe SQL-Parsing- und Ausführungslogik erforderlich ist
* **Schnellere Erkennung von Inkonsistenzen:** Erleichtert das Erkennen von Dateninkonsistenzen
  + **Beispiel:** Bei der statement-basierten Replikation führt ein Update auf dem Master zu keinem Fehler, wenn die Zeile auf dem Replikat fehlt. Die row-basierte Replikation erkennt diesen Fehler und stoppt die Replikation, was eine sofortige Überprüfung ermöglicht.

**Nachteile der Row-basierten Replikation**

* **Fehlende SQL-Transparenz:** Die ursprüngliche SQL-Anweisung wird nicht im Log gespeichert => erschwert die Nachvollziehbarkeit von Änderungen
* **Black-Box-Charakter:** Der Replikationsprozess ist weniger transparent, was die Fehlersuche erschwert
* **Komplexere Fehlerdiagnose:** Probleme sind oft schwerer zu debuggen, da weniger Einblick in den internen Ablauf besteht
* **Probleme bei mehrstufiger Replikation:**
  + Wird eine Anweisung mit @@binlog\_format = STATEMENT auf dem Master ausgeführt, wird sie dort als Statement protokolliert
  + Nachgelagerte Replikate (Second-Level-Replikate) können diese jedoch als row-basiert weiterleiten, was zu Inkonsistenzen bei der Protokollierung führt
* **Einschränkungen bei Schema-Änderungen:** Kann einige Operationen nicht handhaben, die mit statement-basierter Replikation funktionieren, z.B. Schema-Änderungen auf Replikaten
* **Strengere Replikationsanforderungen:**
  + Replikation stoppt bei Fehlern, z.B. wenn eine erwartete Zeile auf dem Replikat fehlt (im Gegensatz zur statement-basierten Replikation, die einfach weitermacht)
  + Dies kann jedoch auch als Vorteil betrachtet werden, da es auf Inkonsistenzen hinweist => kann mit der Option slave\_exec\_mode konfiguriert werden

**Partitioned Tables (Partitionierte Tabellen)**

* Eine partitionierte Tabelle ist eine einzelne logische Tabelle, die aus mehreren physischen Subtabellen besteht.
* Der Partitionierungscode ist lediglich eine Hülle um eine Gruppe von Handler-Objekten (Handler objects), die die zugrunde liegenden Partitionen repräsentieren => leitet Anfragen über die Handler-Objekte an die Storage Engine (Speicher-Engine) weiter.
* Ist eine Art Blackbox (black box), die die zugrunde liegenden Partitionen auf der SQL-Ebene verbirgt => diese sind jedoch relativ einfach im Dateisystem zu erkennen, wo die Komponenten-Tabellen mit einer Hash-getrennten Namenskonvention sichtbar sind.
* Die Art und Weise, wie MySQL die Partitionierung implementiert, bedeutet, dass Indexe pro Partition definiert werden, anstatt über die gesamte Tabelle erstellt zu werden.
* Dies unterscheidet sich von Oracle, wo Indexe und Tabellen auf flexiblere und komplexere Weise partitioniert werden können.
* MySQL entscheidet, welche Partition jede Datenzeile enthält, basierend auf der PARTITION BY-Klausel (PARTITION BY clause), die für die Tabelle definiert ist.
* Der Query-Optimizer (Query optimizer) kann Partitionen beim Ausführen von Abfragen ausschließen (pruning) => Abfragen untersuchen nicht alle Partitionen, sondern nur diejenigen, die die gesuchten Daten enthalten.
* Der Hauptzweck der Partitionierung besteht darin, als grobe Form der Indexierung und Datenclusterung über die Tabelle zu dienen.
* Kann dazu beitragen, große Teile der Tabelle vom Zugriff auszuschließen und zusammengehörige Zeilen nahe beieinander zu speichern.
* Partitionierung kann besonders vorteilhaft sein, insbesondere in bestimmten Szenarien:
  + Wenn die Tabelle zu groß ist, um in den Speicher zu passen, oder wenn sich „heiße“ (hot) Zeilen am Ende einer Tabelle befinden, die viele historische Daten enthält.
  + Partitionierte Daten sind einfacher zu verwalten als nicht partitionierte Daten. Zum Beispiel ist es einfacher, alte Daten zu verwerfen, indem eine gesamte Partition gelöscht wird – dies kann schnell erfolgen. Zudem lassen sich einzelne Partitionen optimieren, überprüfen und reparieren.
  + Partitionierte Daten können physisch verteilt werden, sodass der Server mehrere Festplatten effizienter nutzen kann.
  + Partitionierung kann dabei helfen, einige Engpässe (bottlenecks) in bestimmten Workloads zu vermeiden, z. B. per-Index-Mutexe (per-index mutexes) mit InnoDB oder per-Inode-Sperren (per-inode locking) mit dem ext3-Dateisystem.
  + Falls erforderlich, können einzelne Partitionen gesichert und wiederhergestellt werden, was besonders bei extrem großen Datenmengen hilfreich ist.
* Die Partitionierung hat die CREATE TABLE- und ALTER TABLE-Befehle deutlich komplexer gemacht.
* Es gibt einige Einschränkungen für partitionierte Tabellen:
  + Es gibt ein Limit von 1.024 Partitionen pro Tabelle.
  + Jeder Primary Key (Primärschlüssel) oder Unique Index (eindeutiger Index) muss alle Spalten im Partitionierungsausdruck enthalten.
  + Fremdschlüssel-Beschränkungen (foreign key constraints) können nicht verwendet werden

**Wie Partitionierung funktioniert (How Partitioning Works)**

* Nochmals: Partitionierte Tabellen bestehen aus mehreren zugrunde liegenden Tabellen, die durch Handler-Objekte (Handler objects) repräsentiert werden.
* Die Partitionen können nicht direkt angesprochen werden.
* Jede Partition wird von der Storage Engine (Speicher-Engine) auf normale Weise verwaltet (alle Partitionen müssen dieselbe Storage Engine verwenden).
* Alle über die Tabelle definierten Indexe werden tatsächlich als identische Indexe über jede zugrunde liegende Partition implementiert.
* Aus Sicht der Storage Engine sind die Partitionen einfach nur Tabellen:
  + Sie „weiß“ nicht wirklich, ob eine bestimmte Tabelle, die sie verwaltet, eine eigenständige Tabelle oder nur ein Teil einer größeren partitionierten Tabelle ist.

**Operationen auf einer partitionierten Tabelle werden durch folgende logische Abläufe umgesetzt:**

* **SELECT-Abfragen (SELECT queries)**
  + Wenn eine partitionierte Tabelle abgefragt wird, öffnet und sperrt die Partitionierungsschicht (partitioning layer) alle zugrunde liegenden Partitionen. Der Query-Optimizer (query optimizer) bestimmt, ob bestimmte Partitionen ignoriert (pruned) werden können, und dann leitet die Partitionierungsschicht die Handler-API-Aufrufe an die Storage Engine weiter, die die Partitionen verwaltet.
* **INSERT-Abfragen (INSERT queries)**
  + Beim Einfügen einer Zeile öffnet und sperrt die Partitionierungsschicht alle Partitionen, bestimmt, welche Partition die Zeile erhalten soll, und leitet die Zeile an diese Partition weiter.
* **DELETE-Abfragen (DELETE queries)**
  + Funktioniert genauso wie INSERT und leitet die Löschanfrage an die entsprechende Partition weiter.
* **UPDATE-Abfragen (UPDATE queries)**
  + Wie zuvor: Die Zeile wird abgerufen, modifiziert und die Partition bestimmt, die die neue Zeile enthalten soll. Anschließend wird die Zeile mit einer Einfügeanfrage (insertion request) an die Zielpartition weitergeleitet und die Löschanfrage (deletion request) an die Quellpartition gesendet.
* Einige dieser Operationen unterstützen das **Pruning** (Ausschließen nicht relevanter Partitionen) => Beispielsweise muss der Server beim Löschen einer Zeile diese zuerst lokalisieren.
* Der Server kann Partitionen, die die Zeile nicht enthalten können, ausschließen, wenn eine WHERE-Klausel (WHERE clause) verwendet wird, die mit dem Partitionierungsausdruck übereinstimmt => gilt auch für UPDATE-Abfragen.
* INSERT-Abfragen sind von Natur aus selbstgepruned (self-pruned).
* Obwohl die Partitionierungsschicht alle Partitionen öffnet und sperrt, bedeutet dies nicht, dass die Partitionen dauerhaft gesperrt bleiben – dieser Sperr- und Entsperrzyklus ähnelt der Art und Weise, wie Abfragen gegen gewöhnliche InnoDB-Tabellen ausgeführt werden

**Arten der Partitionierung (Types of Partitioning)**

* MySQL unterstützt mehrere Arten der Partitionierung
* Die am häufigsten verwendete Art ist die **Range-Partitionierung (range partitioning)**, bei der jede Partition so definiert wird, dass sie einen bestimmten Wertebereich für eine Spalte oder mehrere Spalten akzeptiert – oder eine Funktion über diese Spalten
* Zum Beispiel könnte man Verkäufe (sales) basierend auf dem Jahrzehnt partitionieren
* In der Partitionierungsklausel (partitioning clause) können viele Funktionen verwendet werden => Hauptanforderung ist, dass sie eine nicht-konstante, deterministische Ganzzahl (integer) zurückgeben, z. B. **YEAR()**
* Ab MySQL 5.5 kann der Partitionierungstyp **RANGE COLUMNS** verwendet werden, sodass direkt nach Datums-Spalten partitioniert werden kann, ohne eine Funktion zu verwenden
* Die Partitionierung nach Zeitintervallen ist eine gängige Methode zur Arbeit mit datumsbasierten Daten
* MySQL unterstützt außerdem die Key-, Hash- und List-Partitionierung (key, hash, list partitioning) – einige davon ermöglichen auch Subpartitionen (subpartitions), die jedoch selten in Produktionsumgebungen genutzt werden
* Eine Anwendung von Subpartitionen, die wir gesehen haben, bestand darin, einen per-Index-Mutex (per-index mutex) in InnoDB zu umgehen – bei einer Tabelle, die ähnlich wie unser vorheriges Beispiel aufgebaut war
* Die Hash-Subpartitionierung (subpartitioning by hash) half dabei, die Daten in kleinere Stücke zu zerlegen und das Problem zu entschärfen

**Zwei Anwendungsfälle**

* Partitionierung per Modulo-Funktion (modulo function) zur Erstellung einer Round-Robin-Tabelle (round-robin table), die nur eine begrenzte Menge an Daten speichert
  + Beispiel: Partitionierung von Datums-basierten Daten nach Tageswert modulo 7, um immer nur die letzten 7 Tage der Daten zu behalten
* Partitionierung nach einem Ausdruck wie HASH(id DIV 1000000)
  + Erstellt eine neue Partition für jede Million eingefügter Zeilen
  + Ziel: Die neuesten Daten zusammen gruppieren, ohne den Primary Key (primären Schlüssel) ändern zu müssen

**Wie man Partitionierung verwendet (How to Use Partitioning)**

* Stell dir vor, du möchtest Abfragen über Datenbereiche aus einer riesigen Tabelle ausführen, die viele Jahre historischer Metriken in zeitlicher Reihenfolge enthält
* Du möchtest Berichte über den letzten Monat erstellen => etwa 100 Millionen Zeilen sollen zurückgegeben werden
* Angenommen, du hast Hardware aus dem Jahr 2012 und deine Tabelle ist 10 Terabyte groß
* Sie ist viel größer als der verfügbare Speicher und läuft auf traditionellen Festplatten, nicht auf Flash-Speicher
* Eines ist sicher: Du kannst nicht jedes Mal die gesamte Tabelle scannen, wenn du eine Abfrage durchführst – sie ist einfach zu groß
* Und du möchtest keinen Index verwenden, weil der Wartungsaufwand und der Speicherplatzverbrauch zu hoch wären
* In manchen Fällen kann das mit ein oder zwei Indexen funktionieren, aber selten mit mehr
* Bei sehr großen Tabellen funktionieren B-Tree-Indexe (B-Tree indexes) nicht mehr.
* Die Kosten für die Pflege des Indexes (Speicherplatz, I/O-Operationen) sind ebenfalls sehr hoch
* Es bleiben nur zwei praktikable Optionen: Entweder muss deine Abfrage einen sequentiellen Scan über einen Teil der Tabelle durchführen, oder der gewünschte Teil der Tabelle und der Index müssen vollständig in den Speicher passen
* Der Schlüssel ist, Partitionierung als eine grobe Form der Indexierung zu betrachten, die nur sehr wenig Overhead verursacht und dich in die Nähe der gewünschten Daten bringt
* Von dort aus kannst du entweder das relevante Datengebiet sequentiell scannen oder es in den Speicher laden und indexieren
* Partitionierung hat einen geringen Overhead, weil es keine Datenstruktur gibt, die auf einzelne Zeilen zeigt und ständig aktualisiert werden muss
* Partitionierung identifiziert Daten nicht auf Zeilenebene und besitzt keine separate Datenstruktur
* Stattdessen gibt es eine mathematische Formel (equation), die bestimmt, welche Partitionen welche Kategorien von Zeilen enthalten können

**Zwei Strategien, die bei sehr großen Datenmengen funktionieren:**

* **Daten scannen, nicht indexieren (Scan the data, don’t index it):**
  + Tabellen ohne Indexe erstellen und Partitionierung als einziges Mittel verwenden, um zu den gewünschten Zeilen zu navigieren => funktioniert gut, wenn eine WHERE-Klausel verwendet wird, die die Abfrage auf eine kleine Anzahl von Partitionen beschränkt
* **Daten indexieren und „heiße“ Daten separieren (Index the data, and segregate hot data):**
  + Wenn der Großteil der Daten selten genutzt wird, aber ein „heißer“ (hot) Teil regelmäßig abgefragt wird, kann man die Partitionierung so gestalten, dass die heißen Daten in einer einzigen Partition gespeichert werden, die klein genug ist, um mit ihren Indexen vollständig in den Speicher zu passen
  + Dann können Indexe hinzugefügt und Abfragen optimiert werden, ähnlich wie bei kleineren Tabellen

**Was schiefgehen kann (What Can Go Wrong)**

* Die beiden Partitionierungsstrategien basieren auf zwei wichtigen Annahmen:
  + Man kann die Suche durch das Pruning von Partitionen beim Abfragen eingrenzen
  + Die Partitionierung selbst ist nicht sehr kostspielig
* Diese Annahmen sind nicht immer gültig

**NULLs können das Pruning verhindern (NULLs can defeat pruning)**

* Das Ergebnis der Partitionierungsfunktion kann NULL sein
* Dies passiert auch, wenn order\_date nicht NULL-fähig ist, weil du einen Wert speichern kannst, der kein gültiges Datum ist
* Jede Zeile, deren order\_date entweder NULL oder kein gültiges Datum ist, wird in der ersten Partition gespeichert, die du definierst (siehe Beispiel)
* Wenn du nach dem Jahr 2012 suchst, musst du zwei Partitionen statt nur einer durchsuchen, um die Zeilen zu finden, was definitiv unerwünscht ist – besonders wenn die erste Partition groß ist
* Workaround mit einer dedizierten Partition: PARTITION p\_nulls VALUES LESS THAN (0)
* Oder du kannst nach der Spalte selbst partitionieren, anstatt eine Funktion über der Spalte zu verwenden: PARTITION BY RANGE COLUMNS(order\_date)

**Mismatch zwischen PARTITION BY und Index (Mismatched PARTITION BY and index)**

* Wenn du einen Index definierst, der nicht mit der Partitionierungs-Klausel übereinstimmt, können Abfragen möglicherweise nicht pruned werden
* Angenommen, du definierst einen Index auf a und partitionierst nach b
* Jede Partition wird ihren eigenen Index haben, und eine Abfrage auf diesem Index öffnet und überprüft jeden Indexbaum in jeder Partition
* Dies könnte schnell sein, wenn die Nicht-Blatt-Knoten (non-leaf nodes) jedes Indexes im Speicher sind
* Um dieses Problem zu vermeiden, solltest du versuchen, auf nicht partitionierten Spalten keine Indexe zu erstellen, es sei denn, deine Abfragen enthalten auch einen Ausdruck, der beim Pruning der Partitionen hilft
* Das klingt einfach zu vermeiden, aber es kann dich überraschen
* Zum Beispiel: Angenommen, eine partitionierte Tabelle ist die zweite Tabelle in einem Join, und der Index, der für den Join verwendet wird, ist nicht Teil der Partitionierungs-Klausel => jede Zeile im Join wird jede Partition in der zweiten Tabelle durchsuchen

**Die Auswahl von Partitionen kann teuer sein (Selecting partitions can be costly)**

* Fragen wie „Wo finde ich Zeilen, die dieser Abfrage entsprechen?“ können bei der Range-Partitionierung teuer sein, weil der Server die Liste der Partitionierungsdefinitionen durchsucht, um die richtige zu finden
* Eine lineare Suche ist nicht sehr effizient, wie sich herausstellt, daher steigen die Kosten, je mehr Partitionen es gibt
* Die Abfragen, bei denen wir den schlimmsten Overhead bei diesem Typ festgestellt haben, sind zeilenweise Einfügungen (row-by-row inserts)
* Für jede Zeile, die du in eine Tabelle einfügst, die nach Range partitioniert ist, muss der Server die Liste der Partitionen durchsuchen, um das Ziel zu wählen
* Du kannst dieses Problem mildern, indem du die Anzahl der Partitionen begrenzt, die du definierst
* Etwa hundert Partitionen funktionieren für die meisten Systeme
* Andere Partitionstypen, wie Key- und Hash-Partitionen, haben diese Einschränkung nicht

**Das Öffnen und Sperren von Partitionen kann teuer sein (Opening and locking partitions can be costly)**

* Das Öffnen und Sperren von Partitionen, wenn eine Abfrage auf eine partitionierte Tabelle zugreift, ist eine andere Art von Overhead pro Partition
* Öffnen und Sperren erfolgen vor dem Pruning, sodass dieser Overhead nicht pruned werden kann
* Diese Art von Overhead ist unabhängig vom Partitionierungstyp und betrifft alle Arten von Anweisungen
* Sie fügt einen besonders merklichen Overhead bei kurzen Operationen hinzu, wie z. B. Einzelzeilen-Suchen nach Primärschlüssel
* Versuche, Operationen in Bulk auszuführen, wie z. B. Multirow-Inserts
* Begrenze die Anzahl der definierten Partitionen, um die Situation zu verbessern

**Wartungsoperationen können teuer sein (Maintenance operations can be costly)**

* Einige Partition-Wartungsoperationen sind sehr schnell, wie z. B. das Erstellen oder Löschen von Partitionen
* Andere Operationen, wie REORGANIZE PARTITION, funktionieren ähnlich wie ALTER: Sie kopieren Zeilen
* Zum Beispiel arbeitet REORGANIZE PARTITION, indem es eine neue temporäre Partition erstellt, Zeilen dorthin verschiebt und die alte Partition löscht, wenn es fertig ist

**Weitere Einschränkungen in der aktuellen Implementierung (Other Limitations in the current implementation)**

* Alle Partitionen müssen dieselbe Storage Engine verwenden
* Es gibt Einschränkungen bezüglich der Funktionen und Ausdrücke, die in einer Partitionierungsfunktion verwendet werden können
* Einige Storage Engines funktionieren nicht mit Partitionierung

**Optimierung von Abfragen (Optimizing Queries)**

* Partitionierung führt zu neuen Wegen, Abfragen zu optimieren, bringt aber auch entsprechende Fallstricke mit sich
* Noch einmal: Die größte Chance besteht darin, dass der Optimierer die Partitionierungsfunktion verwenden kann, um Partitionen zu prunen
* Wie bei einem grobgranularen Index lässt das Pruning Abfragen auf deutlich weniger Daten zugreifen, als sie es sonst tun würden (im besten Fall)
* Es ist sehr wichtig, den partitionierten Schlüssel in der WHERE-Klausel anzugeben, selbst wenn er ansonsten redundant ist, damit der Optimierer ungenutzte Partitionen prunen kann
* Wenn du dies nicht tust, muss der Abfrageausführungsmechanismus alle Partitionen in der Tabelle durchsuchen, was bei großen Tabellen extrem langsam sein kann
* Du kannst EXPLAIN PARTITIONS verwenden, um zu sehen, ob der Optimierer Partitionen pruned:
  + mysql> EXPLAIN PARTITIONS SELECT \* FROM sales \G
  + Ergebnis: Partitionen: p\_2010, p\_2011, p\_2012
  + Unterschied hier: WHERE day > '2011-01-01' => nur 2 Partitionen
* Der Optimierer ist ziemlich gut im Prunen; zum Beispiel kann er Bereiche in Listen von diskreten Werten umwandeln und auf jedes Element in der Liste prunen
* Die folgende WHERE-Klausel ist theoretisch prunbar, aber MySQL kann sie nicht prunen:
  + mysql> EXPLAIN PARTITIONS SELECT \* FROM sales\_by\_day WHERE YEAR(day) = 2010\G
* MySQL kann nur auf Vergleiche mit den Partitionierungsspalten prunen
  + Es kann nicht auf das Ergebnis eines Ausdrucks prunen, auch wenn der Ausdruck identisch mit der Partitionierungsfunktion ist
  + Ähnlich wie bei indexierten Spalten, die im Abfrageausdruck isoliert sein müssen, damit der Index verwendet werden kann
  + mysql> EXPLAIN PARTITIONS SELECT \* FROM sales\_by\_day WHERE day BETWEEN '2010-01-01' AND '2010-12-31'\G
  + Weil die WHERE-Klausel nun direkt auf die Partitionierungsspalte verweist und nicht auf einen Ausdruck, kann der Optimierer andere Partitionen prunen
* Der Optimierer ist auch intelligent genug, Partitionen während der Abfrageverarbeitung zu prunen
  + Wenn zum Beispiel eine partitionierte Tabelle die zweite Tabelle in einem Join ist und die Join-Bedingung der partitionierte Schlüssel ist, wird MySQL nur nach übereinstimmenden Zeilen in den relevanten Partitionen suchen
* EXPLAIN zeigt das Partition-Pruning nicht, weil es zur Laufzeit passiert, nicht zur Abfrageoptimierungszeit

**Merge-Tabellen (Merge Tables)**

* Partitionierung erzwingt rigoros die Abstraktion, verweigert den Zugriff auf die zugrunde liegenden Partitionen und erlaubt nur den Zugriff auf die partitionierte Tabelle
* Merge-Tabellen ermöglichen den Zugriff auf die zugrunde liegenden Tabellen getrennt von der Merge-Tabelle
* Wie partitionierte Tabellen sind Merge-Tabellen Wrapper um zugrunde liegende MyISAM-Tabellen mit derselben Struktur
* Partitionierung ist besser in den Abfrageoptimierer integriert und ist der Weg der Zukunft
* Merge-Tabellen sind quasi-depreziert und könnten irgendwann entfernt werden.
* Eine Merge-Tabelle ist im Wesentlichen nur ein Container, der die echten Tabellen hält
* Du gibst an, welche Tabellen mit einer speziellen UNION-Syntax in der CREATE TABLE-Anweisung eingeschlossen werden sollen
* Die zugrunde liegenden Tabellen haben genau die gleiche Anzahl und Typen von Spalten, und alle Indizes, die auf der Merge-Tabelle existieren, existieren auch auf den zugrunde liegenden Tabellen
* MERGE-Tabellen (auch bekannt als UNION-Tabellen) wurden in MySQL 8.0 entfernt
* Aufgrund moderner Alternativen und deren Einschränkungen hat MySQL den MERGE-Speichertyp später veraltet und entfernt
* Im Gegensatz dazu sind die Partitionen einer partitionierten Tabelle vor dem MySQL-Server verborgen und nur durch die partitionierte Tabelle zugänglich