Wenn Daten historisch wachsen

Antoniya Kuhlmeyer, Awin AG

MySQL gehört zu den am meisten verbreiteten Open-Source-Datenbank-Systemen. Es lässt sich mit wenigen Handgriffen installieren und auch die Replikation ist sehr einfach aufzusetzen. Mit stets wachsender Datenmenge kann allerdings ein kleines Missgeschick im Design oder in der Konfiguration eines MySQL-Servers zu einem späteren Zeitpunkt eine echte Herausforderung werden.

Wer mit relationalen Datenbanken und SQL vertraut ist, erhofft sich einen schnellen Einstieg in die MySQL-Technologie. Kommt man allerdings ungeplant und unvorbereitet zu deren produktiven Einsatz, läuft man Gefahr, auf einige Fallen zu stoßen. Der Artikel zeigt gängige Fallen im Design und Betrieb von MySQL-Datenbanken auf. Im Mittelpunkt stehen praktische Erfahrungen rund um Datenbank-Design, Anwendung verschiedener Storage Engines und deren Auswirkung auf Performance und Betrieb, High Availability und Replikation.

MySQL und relationale Datenbank-Systeme

Die Grundlagen für relationale Datenbanken wurden bereits in den 1960er-Jahren gelegt. Heutzutage gelten sie als eine bewährte Technik, um Daten zu organisieren, und bilden eine unentbehrliche und zentrale Komponente vieler IT-Systeme.

Auch wenn MySQL in die Kategorie der relationalen Datenbanksysteme fällt, bedeutet das nicht zwingend, dass MySQL die ACID-Eigenschaften erfüllt. Das wohl wichtigste und ungewöhnlichste Merkmal von MySQL ist seine Engine-Architektur, die Abfrage-Verarbeitung von der eigentlichen Datenspeicherung trennt. Die MySQL Storage Engines unterscheiden sich wesentlich in der Art, wie sie Daten intern ablegen. Die Wahl der Storage Engine bestimmt somit, ob zum Beispiel Fremdschlüssel, Transaktionen und welche Art von Indizes unterstützt werden.

Das Produktions-System

Die nachfolgend aufgeführten Szenarien basieren auf einem Legacy-Datenbank-System mit zwei Mastern in MySQL Version 5.5 und einer Datengröße von jeweils zehn Terabyte. Darüber hinaus verfügt das System über einen Mix von rund zwanzig Slaves, die unterschiedlichste Replikationsfilter einsetzen und in MySQL 5.5 oder 5.6 laufen. Die Replikation vom Master zu den Slaves erfolgt über Statement-basierte Logs, es werden also nur die SQL-Statements in die Logs geschrieben und diese dann lokal auf dem jeweiligen Server ausgeführt. Der Datenbank-Betrieb muss rund um die Uhr sichergestellt sein und jegliche Wartungsarbeiten, die den Betrieb unterbrechen, sind auf minimaler Dauer zu halten.

Die richtige Wahl der Storage Engine

Die Struktur der Daten in einer relationalen Datenbank spielt später eine entscheidende Rolle für Performance und Betrieb. Genauso wichtig ist eine möglichst präzise Kapazitätsplanung, basierend auf dem erwarteten Datenvolumen. Der Entwicklungsprozess von relationalen Datenbanken folgt dem Wasserfall-Modell und aus diesem Grund ist es umso wichtiger, dass man grobe Fehler von Beginn an ausschließt. Spätere Änderungen in der Struktur der Daten gestalten sich oft als schwierig, wenn die Datenbank bereits im produktiven Einsatz ist.

Im ununterbrochen laufenden Betrieb größerer Datenbank-Systeme ist es oft sogar unmöglich, gewisse Optimierungen im Nachhinein durchzuführen, ohne die Verfügbarkeit der Datenbank zu unterbrechen. In Umgebungen mit sehr schnellem Datenwachstum beobachtet man immer häufiger auftretende Performance-Probleme und auch die Wiederherstellung der Daten nach einem Ausfall beansprucht mehr Zeit, in der die Datenbank unter Umständen gar nicht verfügbar ist.

Diese Bedenken betreffen nicht nur MySQL-Datenbanken; die Handhabung einer MySQL-Datenbank ist allerdings zum größten Teil davon bestimmt, welche Storage Engine im Einsatz ist. Im folgenden Abschnitt werden reelle Anwendungsfälle für verschiedene MySQL Storage Engines vorgestellt und auf ihre Vor- und Nachteile im Live-Betrieb untersucht.

InnoDB

Seit Version 5.5.5 ist InnoDB die Default Storage Engine in MySQL. Sie ist konform mit den ACID-Eigenschaften, da es intern ein Transaktionsprotokoll gibt, um die Datenkonsistenz auch nach einem Absturz des Datenbank-Servers zu erhalten. InnoDB implementiert Fremdschlüssel und ist die einzige Engine, die Clustered Indizes unterstützt. Darüber hinaus sperrt InnoDB bei Datenzugriffen einzelne Zeilen und nicht die komplette Tabelle, was eine höhere Anzahl von gleichzeitigen Zugriffen auf derselben Tabelle erlaubt. Aus diesem Grund können allerdings auch Deadlocks auftreten, die InnoDB in den meisten Fällen automatisch erkennen und auflösen kann. Aufgrund seiner Robustheit sollte InnoDB in einem produktiven OLTP-System die erste Wahl sein, dennoch gibt es Szenarien, in denen die Performance ausbleibt.

Sehr große Tabellen: Im System der Autorin gab es eine besonders große InnoDB-Tabelle. Die Daten wurden nur eingefügt, aber nie verändert. Über die Zeit wuchs die Tabelle auf 1,9 TB und aufgrund fehlender Indizes war sie nur für einige wenige Anwendungsfälle sinnvoll abfragbar. Genauere Untersuchung zeigte, dass Applikationen nur Daten der letzten 24 bis 48 Stunden abfragten und somit ältere Daten potenziell entfernt werden konnten. Die Tabelle von alten Daten zu befreien, gestaltete sich dennoch schwieriger als erwartet, denn es war weder ein inkrementeller Primärschlüssel noch ein passender Index vorhanden. Die Erstellung eines solchen Index hätte die Tabelle für unbestimmte Zeit gesperrt und war somit auch keine Option im Produktionssystem.

Die Lösung: Hat man Prozesse, die auf Daten der letzten x Stunden/Tage zugreifen, ist es sinnvoll, über Partitionierung nachzudenken. Im Beispiel wurde eine zweite Tabelle erstellt und per MySQL-Replikation befüllt. Diese wurde in „RANGE"-Partitionen auf einer Datetime-Spalte organisiert. Lese-Operationen haben somit effektiv nur 1 bis 2 Tagespartitionen abgefragt statt der kompletten Datenmenge von 1,9 TB über einen nicht Zeitstempel-basierten Index. In Zahlen ausgedrückt bedeutete dies, dass der Index von 800 GB nicht mehr komplett durchsucht wurde, sondern lediglich maximal zwei Partitionen mit jeweiliger Größe von 2 bis 3 GB. Möchte man die Tabelle auf demselben Datenbankserver behalten, kann man alternativ mit einem Insert-Trigger arbeiten, der die neue Tabelle parallel zu der aktuellen inkrementell befüllt. Ist die neue Tabelle ausreichend befüllt, kann man die Replikation / den Trigger entfernen und die alte Tabelle außer Betrieb nehmen. Ein zusätzlicher Vorteil der Partitionierung ist, dass

man ältere Partitionen quasi ausschalten kann, ohne die Daten kopieren zu müssen. Es empfiehlt sich, einen Prozess einzuführen, der die Tages-Partitionen automatisiert im Voraus erstellt und abgelaufene dauerhaft entfernt, um somit unkontrolliertes Datenwachstum zu vermeiden.

Kleine Tabelle, viele Schreibzugriffe: Auch bei viel kleineren Datenmengen können bereits ein schlechtes Datenschema oder Applikationsdesign zu Problemen führen. Ein Beispiel dafür ist eine Tabelle, die als Zwischenablage für Daten eingesetzt wird und auf die viele Prozesse schreibend zugreifen. Es gibt dabei im Wesentlichen zwei Arten von Prozessen – die einen fügen Daten in die Tabelle ein, die anderen lesen diese, verarbeiten sie auf Applikationsebene und löschen sie anschließend aus der Tabelle. Die Tabelle verfügte nur über nicht eindeutige Indizes, um die Leseoperationen zu optimieren. Durch steigende Anzahl der zugreifenden Prozesse und die fein-granularen InnoDB-Sperren kommt es häufig zu Deadlocks, die InnoDB automatisch auflöst, indem es eine der konkurrierenden Transaktionen zurücksetzt.

Die Lösung: In einem OLTP-Datenbanksystem sollte man nicht auf geeignete Primärschlüssel verzichten. Verfügt die Tabelle über einen Clustered Index, der einen nummerischen Datentyp verwendet und inkrementell wächst, können konkurrierende Insert- und Delete-Operationen geschickt verteilt werden, sodass keine Deadlock-Situationen entstehen. Wenn die Prozesslogik es zulässt, kann man für Leseoperationen zusätzlich Gebrauch von geeigneten Transaction Isolation Levels machen, die steuern, welche Datenänderungen ab wann für parallel laufende Prozesse sichtbar sind. Generell sind Deadlocks auf Applikationsebene zu lösen, da sie durch die zugreifenden Prozesse ausgelöst werden.

MyISAM

Vor Version 5.5.5 war MyISAM die Standard Engine für Tabellen in MySQL. Im Gegensatz zu InnoDB wird für MyISAM-Tabellen kein Transaktions-Protokoll geführt, was letztendlich bedeutet, dass die Storage Engine nicht ACID-konform ist. Das macht MyISAM sehr anfällig für Daten-Korruptionen, wenn der Datenbank-Server unerwartet stoppt, etwa bei einem Absturz. Ist der Server wieder online, muss die Tabelle von halbgeschriebenen Daten bereinigt werden.

MyISAM unterstützt weder Fremdschlüssel noch Transaktionen. Um parallele Zugriffe auf dasselbe Datenset zu koordinieren, werden nur Sperren auf Tabellen-Ebene benutzt. Diese Faktoren schränken die parallele Abarbeitung von Abfragen enorm ein, wenn Schreibzugriffe involviert sind.

MyISAM-Daten sind in mehreren Files organisiert, die sich bei gestopptem MySQL-Prozess ohne Weiteres vom Filesystem kopieren lassen. Das kann vor allem beim Wiederherstellen von Daten oder bei der Migration von ganzen Datenbanken sehr hilfreich sein. Vor MySQL 5.6 war MyISAM die einzige Storage Engine, die Full-Text-Indizes unterstützte.

MyISAM-Tabellen in der Praxis: In einer produktiven Datenbank hat man selten gute Gründe, um MyISAM als Default Storage Engine einzusetzen. Wer große Datenmengen in MyISAM lagert, kann mit langen Wartezeiten und potenziellem Datenverlust rechnen, wenn diese Tabellen nach Absturz repariert werden müssen.

Logische Backups von MyISAM kollidieren mit Schreib-Operationen und können somit die schreibenden Zugriffe enorm verzögern. Ist der Datenbank-Server, auf dem die MyISAM-Korruption entstanden ist, zusätzlich als Replikat konfiguriert, kann die Tabellensperre während der Repair-Operation unter Umständen die Replikation blockieren. Eine MyISAM-Tabelle von ca. 1 TB benötigte in der genannten Umgebung dafür 12 bis 16 Stunden. Währenddessen war es unmöglich, Daten vom Master zu replizieren, was insgesamt mehr als 24 Stunden beanspruchte, bis alle Tabellen auf dem Slave-Server wieder verfügbar und synchronisiert waren.

Die Lösung: Für Tabellen ab einer Größe von mehreren Gigabyte ist es dringend empfohlen, diese in InnoDB zu konvertieren. Logische Backups von InnoDB-Tabellen blockieren Schreib-Operationen nicht und man kann durch einen bewussten Einsatz von Isolation-Level die Anzahl der parallelen Transaktionen erhöhen. Handelt es sich um größere Tabellen, kann man zusätzliche Tools wie Percona „pt-online-schema-change“ einsetzen, um eine nicht blockierende Konvertierung während des normalen Betriebs zu ermöglichen. Dabei wird intern eine neue Tabelle parallel aufgebaut und per Trigger aktuell gehalten, während Clients weiterhin auf die bestehende Tabelle zugreifen können. Mit diesem Ansatz konnte die Autorin eine MyISAM-Tabelle von 180 GB in Produktion in InnoDB konvertieren. Wichtig zu beachten ist, dass das Tool keine Änderung auf einer Tabelle vornehmen wird, wenn diese keinen Primärschlüssel oder zumindest keinen eindeutigen Schlüssel aufweist.

Merge MyISAM

In MySQL ist es möglich, eine Ansammlung von identischen Tabellen unter deren Beibehaltung zu einer logischen Tabelle zusammenzufassen. In diesem Fall spricht man von einer „Merge-Tabelle“. Dieses Konzept kann nur für MyISAM-Tabellen verwendet werden, die in Bezug auf Reihenfolge und Datentypen der jeweiligen Spalten identisch sind. Vorteile bringt das meist für die Clients, die auf solch eine Tabelle zugreifen, da sie nicht genau wissen müssen, in welcher der zugrunde liegenden Tabellen sich die gefragten Daten befinden.

Merge-Tabellen in der Praxis: Merge-Tabellen basieren auf MyISAM, daher bringen sie alle deren Vor- und Nachteile mit. Aus operativer Sicht ist der Einsatz dieser Engine jedoch nur bedingt sinnvoll.

Praktische Erfahrungen mit einer Merge-Tabelle von über 100 GB und mehr als 220 zugrunde liegenden Tabellen zeigen, dass vor allem auf Replikaten Konflikte zwischen Schreib- und Lese-Operationen entstehen, die den Replikationsprozess blockieren.

Die Lösung: MySQL unterstützt partitionierte Tabellen nativ seit Version 5.1. – ein Feature, das effizientere Abfragemöglichkeiten bietet und flexibler bei der Storage-Engine-Wahl ist. Ein weniger invasiver Ansatz, eine Merge-Tabelle zu ersetzen, wäre, eine View zu definieren, die die MyISAM-Tabellen vereint. In einer MySQL-5.5-Umgebung erwies sich das aufgrund der eher großen Datenmenge als unpraktikable Lösung.

Weitere Storage Engines in Einsatz

MySQL bietet einige weitere Engines, die für speziellere Anwendungsfälle geeignet sind:

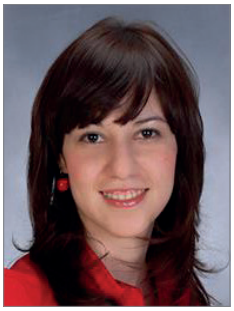
- *Archive*
  Diese Engine eignet sich, wie der Name auch schon sagt, für archivierte Daten. Spalten von Archive-Tabellen können nicht indiziert werden, was aber die hoch komprimierte und somit speichereffiziente Lagerung der Daten unterstützt.

- *Federated*
  Federated-Tabellen ermöglichen den Zugriff auf Daten, die auf einer anderen MySQL-Instanz oder entfernten Servern liegen. Für Clients ist der tatsächliche Speicherort der angefragten Daten transparent. In der Praxis ist die Anwendung dieser Engine nicht empfehlenswert, da selbst bei einer kleinen Anzahl von ausgegebenen Zeilen die Ausführungszeit um mehr als das Hundertfache steigt. In manchen Fällen führen Abfragen auf Federated-Tabellen zum Absturz der MySQL-Instanz, auf der die Clients die Abfragen absetzen.

- *Blackhole*
  Diese Engine eignet sich für komplexere MySQL-Replikations-Topologien, in denen ein oder mehrere Slaves selbst als Master agieren, um weitere Slaves mit Daten zu befüllen. Die Besonderheit von Blackhole-Tabellen ist, dass sie Schreib- und Lese-Zugriffe akzeptieren, aber Datenänderungen nicht persistieren. Auf diese Weise werden lediglich die MySQL-Binary-Logs befüllt, die bei der Replikation zu anderen Slaves weitergereicht werden. Der Vorteil bei diesem Ansatz ist, dass der Zwischen-Master selbst keine Verzögerung in der Replikationskette verursacht, da die Schreib-Operationen nur geloggt, aber nicht auf reellen Daten ausgeführt werden. Voraussetzung für solche Setups ist, dass der Zwischen-Master das Statement-Format für seine Binary-Logs benutzt. Potenzielle Probleme können auftreten, wenn versucht wird, persistierte Tabellen zu erstellen, die auf eine Blackhole-Tabelle referenzieren.

Fazit

MySQL bietet eine Vielzahl von Möglichkeiten und Features für die Speicherung von Daten. Die Wahl der Storage Engine beeinflusst nicht nur die interne Struktur der zu persistierenden Daten, sie bestimmt auch wesentliche Aspekte der Abfrage-Performance, der Backup-Strategien und der Replikations-Konfiguration. Mit steigender Datengröße ist es nicht immer einfach, bestehende Datenstrukturen zu verändern und zu verbessern. Der Artikel hat anhand einiger Beispiele aus der Praxis gezeigt, wie man Storage Engines kombinieren kann, um eine bessere Performance zu erzielen oder weniger Speicherplatz zu belegen.



Antoniya Kuhlmeyer
antoniya.kuhlmeyer@awin.com