Datenbanken

Die alte Welt:
Relationale Datenbanken, Konzepte, Entwurf und Programmierung

Till Hänisch

Relationale Datenbanken

Die alte Welt: Relationale Datenbanken, Konzepte, Entwurf und Programmierung

Till Hänisch

This book is for sale at http://leanpub.com/realtionaledatenbanken

This version was published on 2015-03-09

Contents

Verteilte Datenbanken

Befindet sich eine Datenbank nicht auf einem einzelnen Rechner, sondern sind die Daten über mehrere Rechner verteilt, spricht man von einer verteilten Datenbank. Dies ist ein theoretisch ansprechendes Konzept, können doch die Daten je nach Lokalisierung der Nutzung, Organisation von Verantwortlichkeiten, Verwendungszweck oder technologischer Anforderungen auf verschiedene Systeme verteilt werden. Prinzipiell stellen auch verteilte Transaktionen kein Problem dar. In den neunziger Jahren wurde auf diesem Gebiet umfangreich geforscht und entwickelt[1].

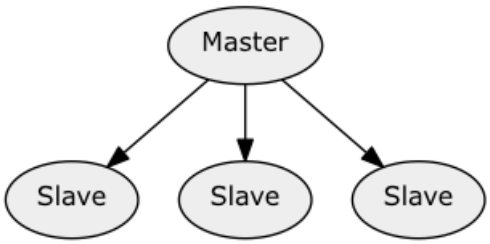
Eine verteilte Transaktion etwa lässt sich anschaulich mit dem Two Phase Commit (2PC) Protokoll realisieren. Das Prinzip entspricht dem Vorgehen des Pfarrers bei einer Trauung: Der Pfarrer (Transaction Coordinator) fragt zunächst alle Parteien (Datenbanken oder Resource Manager) ob sie bereit sind (Prepare to commit). Antwortet eine Partei mit „Ja“, kann die Zustimmung nicht mehr rückgängig gemacht werden. Antwortet eine Partei mit „Nein“ oder überhaupt nicht, wird die Transaktion abgebrochen. Haben alle mit „Ja“ geantwortet, erklärt der Pfarrer (Transaction Coordinator) die Transaktion für erfolgreich und teilt dies allen mit (Commit).

Die Probleme zeigen sich im praktischen Betrieb. Eine Transaktion kann erst dann beendet werden, wenn entweder alle Partner geantwortet haben oder ein entsprechender Timeout entscheidet, dass einer der Partner nicht erreichbar ist. Für viele praktische Belange ist diese Latenz zu groß. Deshalb werden in der Regel asynchrone Verfahren zur Kopplung verwendet [2].

Aber auch ohne die Probleme verteilter Transaktionen sind verteilte Datenbanken hinsichtlich der Performance problematisch. Die Optimierung verteilter Joins ist zwar theoretisch möglich, aber in der Praxis nur schwer durchführbar.

Praktisch haben sich zwei einfache Spezialfälle verteilter Datenbanken durchgesetzt, Replikation und Sharding.

Bei der Replikation werden die gleichen Daten auf mehrere Datenbanken dupliziert, beim Sharding werden die Daten anhand eines Kriteriums auf mehrere Datenbanken aufgeteilt.



Master-Slave-Replikation

Der einfachere Fall von Replikation ist die Master-Slave-Replikation. Hier werden Schreibzugriffe nur auf dem Master durchgeführt, die geänderten Daten werden an die Slaves weitergegeben, technisch in der Regel dadurch realisiert, dass die Log-Dateien des Masters kopiert werden, die Slaves befinden sich ständig im Recovery-Modus. Das ist kein Problem, da auf den Slaves nur Lesezugriffe ausgeführt werden.

[1] Siehe etwa Dadam, Verteilte Datenbanken und Client/Server-Systeme, Springer, 1996.

[2] Siehe etwa G. Hohpe, Your Coffe Shop Doesn’t Use Two-Phase Commit, IEEE Software, Volume 22 Issue 2, March 2005 Page 64-66

Diese Art der Replikation bietet einige Vorteile: Schreibzugriffe auf den Master sind genau so schnell wie bei einer einzelnen Datenbank, Lesezugriffe können durch Zugriffe auf die Replikate beschleunigt werden. Dies ist insbesondere interessant, wenn nur wenige Schreibzugriffe erfolgen, die Nutzer aber weltweit verteilt sind. Bei einer einzelnen Datenbank wären hier die Latenzen zum (Lese-) Zugriff auf den zentralen Datenbankserver langsam, werden die Replikate geschickt verteilt, kann dieses Problem gelöst werden [3].

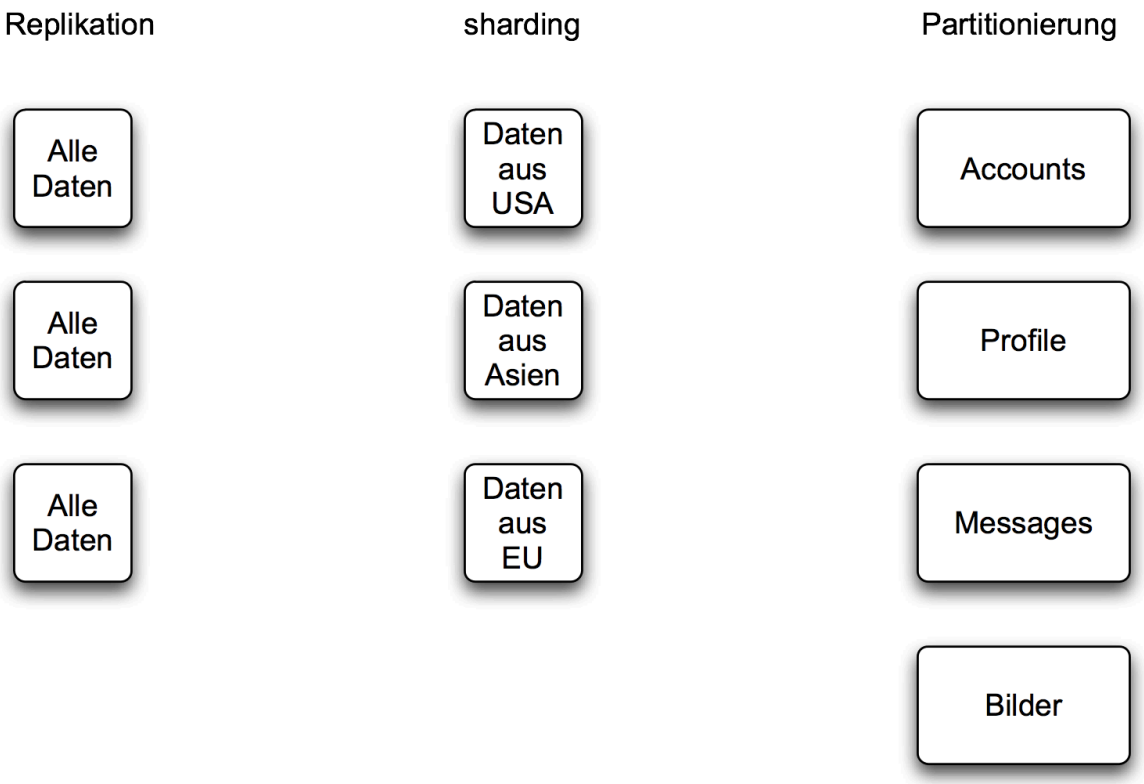
Problematisch ist diese Art der Replikation dann, wenn entweder viele, verteilte Schreibzugriffe stattfinden oder zwar nur wenige Schreibzugriffe erfolgen, diese aber auch möglich sein müssen, wenn die Verbindung zum zentralen Datenbankserver (kurzzeitig) ausfällt („always writeable“). Der erste Fall kann in vielen Fällen durch Sharding gelöst werden, der zweite Fall ist schwierig.

Schwierig bedeutet hier, dass die ACID-Eigenschaften nicht eingehalten werden können, dies ist die Aussage des CAP-Theorems. Das CAP-Theorem besagt, dass bei verteilten Datenbanken entweder Konsistenz oder hohe Verfügbarkeit erreicht werden kann, nicht beides gleichzeitig [4].

Theoretisch gibt es auch noch die Multi-Master-Replikation, d.h. Schreibzugriffe sind auf jeder Datenbank möglich und werden auf die anderen Knoten repliziert, dies ist aber unter Beibehaltung der ACID-Eigenschaften nur in Spezialfällen mit akzeptabler Performance und Verfügbarkeit möglich.

[3] Ganz so unproblematisch ist dieses Verfahren aber auch nicht: Liest etwa ein Client von einem der Slaves und schreibt dann auf den Master und liest dann unmittelbar darauf folgend wieder vom Slave, erhält er unter Umständen die alten, also aus seiner Sicht falsche Werte. Um dies zu vermeiden, müsste nach einem Schreibzugriff vom Master gelesen werden, um „read your writes“-Konsistenz zu erhalten. Gängige Datenbank-Schnittstellen wie ORM-Mapping-Tools bieten diese Funktionen nicht von Haus aus. Die Implementierung von Konsistenz in verteilten Datenbanken muss in der Anwendung selber realisiert werden. Selbst dann lesen verschiedene Clients von verschiedenen Slaves unter Umständen unterschiedliche Daten.

[4] Genau genommen besagt das CAP-Theorem, dass von den drei Eigenschaften Konsistenz (Consistency), Verfügbarkeit (Availability) und Unempfindlichkeit gegenüber Unterbrechungen der Netzwerkverbindung (Partition Tolerance) nur zwei gleichzeitig realisierbar sind. Zumindest gelegentliche, kurzzeitige Unterbrechungen der Netzwerkverbindung sind aber bei verteilten Systemen unvermeidbar, deshalb die einfachere Formulierung im Text. Man darf sich das aber nicht so vorstellen, dass es hier nur zwei Möglichkeiten (hohe Verfügbarkeit/keine Konsistenz und Konsistenz/schlechte Verfügbarkeit gäbe), hier ist ein Spektrum an Möglichkeiten realisierbar. Für jede Anwendung muss deshalb definiert werden, welches Maß an Konsistenz notwendig ist und welche Verfügbarkeit dadurch erreicht werden kann, oder umgekehrt, welche Verfügbarkeit gefordert ist und welche Konsistenzprobleme deshalb gelöst werden müssen. Die zweite Form ist in der Praxis häufiger als die erste. In der Regel ist diese Entscheidung auch für verschiedene Daten innerhalb einer Anwendung unterschiedlich zu treffen: Bei amazon ist beispielsweise der Warenkorb immer schreibbar, beim Auslösen einer Bestellung wird dann allerdings auf Konsistenz geachtet.



Verschiedene Arten der Verteilung

Beim Sharding werden gleichartige Daten auf mehrere Datenbanken aufgeteilt. Die Idee dabei ist, dass Lese- und Schreibzugriffe gemeinsam auf zusammengehörigen Datensätzen stattfinden, sich die Zugriffe aber annähernd gleichmäßig über die Gesamtdatenmenge verteilen. Insbesondere ist dies der Fall, wenn die einzelnen Nutzer jeweils auf unterschiedliche Daten, also insbesondere auf ihre eigenen, zugreifen. Ein Beispiel hierfür ist ein Messaging-System: Werden die Nutzerdaten nach Regionen auf mehrere Datenbanken verteilt, bleiben die meisten Zugriffe lokal. So kann sowohl eine im Mittel gute Performance, als auch eine hohe Verfügbarkeit erreicht werden.

Die beiden Verfahren können auch kombiniert werden, d.h. die einzelnen shards werden zur Erhöhung der Verfügbarkeit und der Lese-Performance repliziert. Gerade Web 2.0 Anwendungen haben ein dieser Architektur entsprechendes Zugriffsmuster. Problematisch hierbei sind die sogenannten Hotspots, also einzelne Datensätze, die von sehr vielen Nutzern gelesen werden. Hier müssen geeignete, anwendungsspezifische Lösungen gefunden werden, eine allgemeine Lösung existiert nicht [5].

Zur Verbesserung der Performance wird häufig zusätzlich zu Replikation und Sharding partitioniert, es wird eine Aufteilung nach der Art der Daten vorgenommen. Bei einem Messaging-System werden etwa in der einen Datenbank die Nutzerprofile, in einer anderen die Accounts, wieder in einer anderen die Nachrichten usw. gespeichert. Wird diese Aufteilung geschickt gewählt, kann in vielen Fällen erreicht werden, dass sich Zugriffe nicht über mehrere Datenbanken erstrecken,

[5] So werden bei Twitter etwa Tweets von Nutzern nicht (nur) bei den Nutzern selber gespeichert, sondern auch bei den Followern. Ansonsten würde bei Nutzern mit sehr vielen Followern ein Performance-Problem entstehen, bei fast jedem Zugriff auf eine Timeline müssten diese durchsucht werden.

sondern lokal in einer Datenbank realisiert werden können [6].

Auch hier ist eine Kombination mit den anderen Verfahren möglich: Zunächst werden die Daten nach Aggregates aufgeteilt, diese dann, etwa regional, durch Sharding verteilt und die einzelnen shards schließlich repliziert. Diese Architektur ist bei Web (2.0) Anwendungen üblich.

Problematisch ist hierbei, dass bei dieser Architektur die meisten Vorteile relationaler Datenbanken nicht genutzt werden können: Keine Transaktionen mehr, keine Gewährleistung von semantischer oder referentieller Integrität, keine flexiblen Abfragen durch Joins (effizient nur noch in den Aggregates) und so weiter. Da die Daten grade zur Verbesserung der Verfügbarkeit und Performance verteilt werden, würde die Verwendung verteilter Zugriffe und Transaktionen diese Architektur konterkarieren.

Letztlich wird in jeder Datenbank nur noch eine einzige Tabelle (ggf. mit einer komplexen, eher objektrelationalen Struktur, möglicherweise aufgeteilt auf einige wenige Tabellen) gespeichert. Da stellt sich schon die Frage, warum man dann noch eine relationale Datenbank braucht.

Zusammenfassung

Daten können über mehrere Rechner verteilt werden. Dies wird meistens zur Erhöhung der Verfügbarkeit (mehrere Kopien der Daten auf mehreren Servern), der Performance (kleinere Latenz beim Zugriff auf näher gelegene Server) oder beidem durchgeführt. Problematisch sind verteilt realisierte Schreibzugriffe, hier können leicht Inkonsistenzen entstehen, wenn die einzelnen Datenbanken nicht mittels Transaktionen synchron gehalten werden, was aber wiederum der Performance und Verfügbarkeit sehr abträglich ist.

[6] Dies ist genau dann der Fall, wenn nicht nach einzelnen Tabellen, sondern nach Aggregates partitioniert wird. Diese Beobachtung stammt von Eric Evans, Domain Driven Design: In der Anwendung wird meistens nicht auf einzelne Datensätze, sondern etwa bei Bestellungen fast immer gemeinsam auf den Bestellkopf, die Bestellpositionen und die Artikeldetails zugegriffen. Diese sollten also auch zusammen verwaltet und gespeichert werden.