Johannes Schildgen

MongoDB kompakt

Was Sie über die NoSQL-Dokumentendatenbank wissen müssen

Mit praktischen Übungsaufgaben

Inhaltsverzeichnis

1. NoSQL-Datenbanken

1.1 Willkommen in der NoSQL-Welt

Der Begriff *NoSQL* darf nicht als Aufschrei „Kein SQL!“ missverstanden werden, sondern er ist vielmehr die Abkürzung für „Not only SQL“. Obwohl sich die Structured Query Language (SQL) und relationale Datenbanksysteme in den meisten Bereichen durchgesetzt haben und obwohl in diesen Systemen jahrzehntelange Forschungen und Weiterentwicklungen gemacht wurden, ist es an der Zeit zu realisieren, dass es nicht nur SQL, sondern auch nützliche Alternativen gibt.

Relationale Datenbanken und die Anfragesprache SQL setzen ein festes Tabellenschema voraus. Die Unterstützung von unbekannten und heterogenen Datenstrukturen, flexiblen Schemata sowie semistrukturierten oder unstrukturierten Daten in SQL ist sehr beschränkt. Das erkennt man bereits daran, dass mit der SQL Data Definition Language (DDL) zuerst Tabellen erstellt und dabei alle ihre Spalten und Datentypen angeben werden müssen, bevor man Daten einfügen und suchen kann. Nachträgliche Änderungen am Schema sind zwar möglich, aber vor allem bei mit vielen Datensätzen gefüllten Tabellen meist sehr teuer. NoSQL-Datenbanken bieten hier den Vorteil der *Schema-Flexibilität.* Die Datenbank muss nicht zwingend wissen, welche Attribute die Daten haben werden und von welchen Datentypen diese sind. Es ist sogar erlaubt, dass alle Datensätze eine unterschiedliche Struktur haben. In relationalen Datenbanken wäre das undenkbar. Dort herrscht horizontale Homogenität, welche besagt, dass in einer Tabelle alle Zeilen die gleichen Spalten haben müssen - Nullwerte sind zwar erlaubt, aber sie belegen trotzdem Platz - sowie vertikale Homogenität, die dafür steht, dass innerhalb einer Tabellenspalte alle Zeilen den gleichen Datentyp haben müssen.

NoSQL-Datenbanken adressieren die Probleme der Speicherung und Verarbeitung von *Big Data.* Gerne charakterisiert man Big Data mit (mindestens) drei Vs. Das erste steht für *Volume,* also enorm große Datenmengen, das zweite für *Velocity,* was für die hohe Geschwindigkeit steht, mit der neue Daten geschrieben werden. Das dritte V haben wir bereits im vorherigen Absatz diskutiert, die *Variety,* also die Heterogenität der Daten.

Neuartige Anwendungen aus den Bereichen Web, Data Mining und Wissenschaft haben andere Anforderungen als klassische Anwendungen, für die relationale Datenbanksysteme weiterhin optimal sind. Im Web wird

beispielsweise das *ACID*-Paradigma nicht so genau genommen. ACID steht für Atomarität, Konsistenz, Isolation und Dauerhaftigkeit und ist beispielsweise in Bankanwendungen unbedingt notwendig, damit Transaktionen korrekt ausgeführt werden und damit parallel laufende Anwendungen sich nicht gegenseitig beeinflussen. Diese Anwendungen laufen meist auf einem einzigen Rechner. Um mit den großen Datenmengen, die beispielsweise in sozialen Netzwerken und Online Shops anfallen, gut arbeiten zu können, setzt man auf verteilte Datenbanksysteme, also solche, die auf mehreren Rechnern laufen und somit sowohl die Speicherung als auch die Berechnung der Daten verteilen. In solch verteilten Systemen lässt es sich nicht vermeiden, dass ein Datenaustausch zwischen Rechnern mal verzögert ausgeführt wird, dass Nachrichten verloren gehen oder Teile des Netzwerks kurzzeitig unerreichbar sind. Ein verteiltes Datenbanksystem muss diese sogenannten Netzwerkpartitionen tolerieren. Das berühmte *CAP-Theorem* (CAP steht für Consistency, Availability und Partition Tolerance) besagt, dass man von den drei Eigenschaften Konsistenz, Verfügbarkeit und Partitionstoleranz nur zwei auf einmal erreichen kann. Da wir wie gerade beschrieben in verteilten Systemen zwingend Partitionstoleranz erfordern, muss man sich also entweder für Verfügbarkeit oder Konsistenz entscheiden. Stellen wir uns eine Reisebuchungswebseite vor, auf der der Preis einer bestimmten Reise um zwanzig Euro erhöht werden soll. In einer verteilten Datenbank ist es vonnöten, dass diese Änderung auf mehreren Rechnern im Netzwerk ausgeführt werden muss. Ist dem Betreiber der Webseite die *starke Konsistenz* so wichtig, dass er es nicht toleriert, dass innerhalb der nächsten Sekunden oder Minuten ein Besucher noch den alten Preis der Reise sieht, muss er bei Netzwerkproblemen damit bezahlen, dass seine Webseite für einige Zeit unerreichbar ist. Denn erst, wenn der neue

Preis auf allen Rechnern angekommen ist, darf der Datensatz wieder gelesen werden. Entscheidet sich der Reisewebsitebetreiber stattdessen für die Verfügbarkeit, kann es beispielsweise bei Nachrichtenverzögerungen im Netzwerk kurzzeitig passieren, dass ein Besucher einen veralteten Wert, also den niedrigen Preis, sieht. Diese Konsistenzstufe wird *Eventual Consistency* genannt, was mit „schließlich konsistent“ zu übersetzen ist. Schließlich, also irgendwann in naher Zukunft, wird die Änderung alle Rechner erreicht haben. Zwischenzeitlich können die Rechner jedoch untereinander in inkonsistenten Zuständen sein; der eine hat den alten Preis, der andere den neuen.

Die Idee von verteilten Datenbanken ist nicht neu. Aber nie hat sich das Verteilen von Daten in relationalen Datenbanksysteme richtig durchgesetzt. Die Replikation, also das Spiegeln des kompletten Datenbestandes auf mehreren Rechnern, wird dagegen sehr wohl eingesetzt und sorgt für Hochverfügbarkeit und verhindert Datenverlust. Möchte man Datensätze jedoch nicht nur spiegeln, sondern tatsächlich aufteilen, leidet die Performanz darunter enorm. Befindet sich die Zeile der Kundentabelle über die Kundin Ulrike auf einem anderen Rechner als die Zeile mit den Reisedaten zur Mittelmeerkreuzfahrt, ist bei einer Anfrage, die einen Verbund dieser beiden Tabellen ausführt, ein Datentransport über das Netzwerk vonnöten. So wie relationale Datenbanksysteme üblicherweise designt werden, also mit normalisierten Tabellen, die über Fremdschlüssel-Primärschlüssel-Beziehungen miteinander in Beziehung stehen, sind ebensolche Verbundabfragen (englisch: Joins) eine der häufigst gestellten Anfragen in gängigen Anwendungen.

In NoSQL-Datenbanksystemen löst man sich nicht nur aufgrund der Schemaflexibilität vom Modell mit Tabellen und Spalten. Ausgenommen von Graphdatenbanken fallen NoSQL-Systeme üblicherweise in die Kategorie der

aggregate-oriented stores. Aggregate steht in dem Fall für so etwas wie die Gesamtheit. Die Idee ist, alles was zusammen gehört, auch zusammen zu speichern. In unserer Reisedatenbank könnte man beispielsweise im Datensatz zu einer Reise die Liste der Kunden speichern, die diese Reise gebucht oder sie auf ihren Merkzettel gesetzt haben. Die Art von Speicherung muss natürlich gut überlegt sein. Man könnte nämlich auch die Liste der gebuchten Reisen in einem Personendatensatz speichern. Wie genau man es modelliert, hängt von der Anwendung ab und welche Anfragen diese üblicherweise an die Datenbank stellt. Der große Vorteil der Speicherung als Gesamtheit ist die Vermeidung von Joins. Dadurch können NoSQL-Datenbanken wunderbar auf mehrere Rechner verteilt und Anfragen schnell beantwortet werden. Gleichzeitig muss man sich jedoch von der Speicherung in Tabellen und Spalten verabschieden und offen sein für neue Datenmodelle und Anfragesprachen.

1.2 Klassifizierung

In den letzten Jahren kamen hunderte NoSQL-Datenbanksysteme auf den Markt. Jedes unterscheidet sich in der Form, in der Datensätze abgespeichert werden und auf welche Weise man als Benutzer Anfragen an das System stellt. Damit nicht jede Firma ihr speziell angefertigtes System NoSQL-Datenbank nennen darf, wird oft das Kriterium genannt, dass NoSQL-Datenbanken *open source* sein müssen.

In diesem Kapitel schauen wir uns die vier Klassen an, in die man fast alle NoSQL-Datenbanksysteme einordnen kann. Die ersten drei sind die Key-Value-Datenbanken, Wide-

Column-Stores und Dokumentendatenbanken. Diese werden, wie oben beschrieben, auch aggregate-oriented stores genannt, da man alles zu einem Datensatz gehörige innerhalb dieses Datensatzes speichert. Die vierte Klasse, die Graphdatenbanken, verfolgen einen anderen Ansatz. Sie verbinden Datensätze untereinander, wie man es aus Knoten und Kanten eines Graphen kennt.

1.2.1 Key-Value-Datenbanken

Man stelle sich eine Tabelle vor, die nur zwei Spalten hat, eine für einen Schlüssel, eine für einen Wert. Genau das sind Key-Value-Datenbanken. Anfragen an diese werden gestellt, wie man es aus Maps in Programmiersprachen kennt: `GET k` liefert den Wert zum Schlüssel `k`, `SET k 5` setzt k auf fünf. Mehr Möglichkeiten bei der Speicherung hat man, wenn man den Schlüssel aus mehreren Elementen zusammensetzt, z. B. könnte der Wert des Schlüssels `pers/12/name` der Name einer Person mit der ID zwölf sein. Viele Key-Value-Stores ermöglichen neben simplen Datentypen wie Zahlen und Zeichenketten die Verwendung komplexer Typen wie Listen und Maps als Werte. Somit kann eine einzige Zeile auch einen kompletten Personendatensatz repräsentieren und nicht nur den Namen einer Person. Zwei berühmte Key-Value-Stores sind Redis und Riak.

1.2.2 Wide-Column-Stores

„A database administrator walks into a NoSQL bar, but he turns and leaves because he couldn’t find a table.“