Martin Schönert
(m.schoenert@triagens.de)
ist einer der Geschäftsführer von triAGENS in Köln. Seine Fokusthemen sind Enterprise-Architektur und -Entwicklung sowie der Einsatz von speicherbasierten Datenbanken. Aktuell kann er als Architekt der Open-Source-Datenbank ArangoDB seine Idealvorstellung einer universellen Multi-Model-Datenbank verwirklichen. Als Sprecher findet man ihn bevorzugt auf Datenbank-Konferenzen im In- und Ausland.

Neue Wege der Datenpersistierung – wie NoSQL-Datenbanken Architektur und Business verändern

Nicht-relationale Datenbanken, häufig etwas irreführend als NoSQL-Datenbanken bezeichnet, stehen an der Schwelle zum Mainstream. Immer mehr Projekte – insbesondere im Webumfeld – setzen nicht-relationale Datenbanken, wie z. B. MongoDB, CouchDB oder Redis, ein. Dafür gibt es gute Gründe, denn durch diese Datenbanken und die dahinterstehenden Technologien ergeben sich neue Möglichkeiten für die Architektur, z. B. für die Skalierbarkeit von Systemen, wenn man die notwendigen Architekturentscheidungen konsequent etabliert. Dieser Artikel beschreibt die neuen Möglichkeiten und die Konsequenzen für Entwicklung und Business.

Während relationale Datenbanksysteme, wie Oracle, DB/2, MySQL und PostgreSQL, sich in vieler Hinsicht gleichen, unterscheiden sich die „neuen" nicht-relationalen Datenbanken teilweise deutlich voneinander. Grob lassen sich die aktuellen Ansätze vier Konzepten zuordnen:

- In der Gruppe der *Key-Value Stores* ist die Funktionalität auf ein Minimum reduziert: sie kann zu Schlüsseln (Keys) jeweils Werte (Values) speichern. Man kann lediglich den Wert zu einem Schlüssel abfragen, aber nicht nach allen Werten mit einer bestimmten Eigenschaft suchen. Ein Beispiel hierfür ist Riak.
- *Document Stores* speichern Dokumente, d. h. strukturierte Objekte mit mehreren Attributen, die jeweils aus einem Attributnamen und einem möglicherweise zusammengesetzten Attributwert bestehen. Man kann z. B. nach allen Dokumenten suchen, die für ein bestimmtes Attribut einen bestimmten Wert haben. Beispiele hierfür sind die Open-Source-Datenbanken MongoDB und CouchDB.
- *Extended-Column Stores* legen die Daten als Datensätze (Rows) mit Attributen (Columns) ab. Anders als bei relationalen Datenbanksystemen kann die Anzahl der Attribute sehr groß und auch zwischen den Datensätzen unterschiedlich sein. Das Apache-Projekt Cassandra ist ein Beispiel für dieses Konzept.
- *Graphendatenbanken* schließlich speichern einerseits Knoten und andererseits Kanten zwischen diesen Knoten. Diese können z. B. Personen und Beziehungen zwischen diesen Personen repräsentieren. Die Datenbank kann dann in diesem Graphen nach Verbindungen (z. B. „über wie viele Zwischenschritte sind zwei Personen miteinander bekannt") wie auch nach Mustern (z. B. „wer ist mit vielen Personen bekannt, die sonst niemanden kennen") suchen. Neo4J ist ein bekannter Open-Source-Vertreter in dieser Gruppe.

Polyglotte Persistenz – „the right tool for the job"

Aus der Unterschiedlichkeit der verschiedenen nicht-relationalen Datenbanksysteme ergibt sich die erste architekturelle Konsequenz mit Auswirkungen auf die Gesamtarchitektur – die sogenannte „polyglotte Persistenz". Hinter diesem Begriff verbirgt sich die Idee, in einem umfangreicheren System nicht nur eine, sondern mehrere unterschiedliche Datenbanken einzusetzen, jeweils „the right (database) tool for the job" [Fow12].

Das könnte zum Beispiel in einem Webshop zu folgender Verteilung führen:

- die Daten, die zu einer Benutzersession gehören (z. B. Benutzer-ID und die gewählten Produkte im Einkaufswagen) werden in einem Key-Value Store gespeichert,
- Informationen über getätigte Einkäufe und Wunschlisten werden in einer Graphendatenbank gespeichert, in der nach Verbindungen und Zusammenhängen gesucht werden kann, z. B. für Kaufempfehlungen,
- die Kundenstammdaten sowie die Abrechnungsdaten werden in einem klassischen relationalen Datenbanksystem gehalten.

Der Vorteil ist, dass hierbei jedes dieser Systeme seine Stärken ausspielen kann: Der *Key-Value Store* kann mit der großen Menge an Daten und der hohen Frequenz an Abfragen und Änderungen am besten umgehen. Dass er keine komplexen Abfragen erlaubt, ist kein Problem, solche werden für Sessions nicht gebraucht. *Graphendatenbanken* können mit den komplexen Beziehungen (wer hat was gekauft, was angeschaut, was bewertet, welche Nutzer verhalten sich ähnlich) am besten umgehen. Dass manche Abfragen länger dauern, ist kein Problem, da Empfehlungen asynchron berechnet werden. Für die Verarbeitung der Stamm- und Abrechnungsdaten gibt es fertige Finanzbuchhaltungssysteme, die auf *relationalen Datenbanken* arbeiten.

Komponentenkoppelung und polyglotte Persistenz

Voraussetzung für ein System mit polyglotter Persistenz ist, dass die verschiedenen Systemkomponenten über Services miteinander gekoppelt sind und nicht über die Datenbank als verbindendes Element. Das kann entweder sehr „enterprise-mäßig“ realisiert sein – mit Web-Services, SOAP und einem Enterprise-Service-Bus. Oder aber eher „leichtgewichtig“ – mit REST-Verbindungen zwischen den Komponenten, JSON und HTTP.

Natürlich hat die polyglotte Persistenz aber nicht nur Vorteile.

Wie immer steigt mit der Anzahl der eingesetzten Technologien die Komplexität. So braucht man für jede der eingesetzten Datenbanken entsprechendes Knowhow, um sie effektiv und effizient einsetzen zu können, und muss insbesondere auch wissen, unter welchen Umständen sie ausfallen kann, um eben diese Situationen zu vermeiden.

Hinzu kommt, dass jede der eingesetzten Datenbanken bestimmte Einschränkungen besitzt. Dies kann problematisch werden, wenn ein System bzw. eine Komponente weiterentwickelt wird und diese Weiterentwicklung Datenbankeigenschaften benötigt, die diese nicht bietet.

Skalierung

Eine häufig angeführte Begründung für den Einsatz von nicht-relationalen Datenbanken ist der Wunsch nach maximaler horizontaler Skalierbarkeit. Wird mehr Durchsatz benötigt oder steigt die Datenmenge, soll es im Idealfall ausreichen, weitere Server in den Cluster zu stellen, den Rest erledigt die Datenbank automatisch.

Aus der NoSQL-Familie können die Key-Value Stores hier punkten. Systeme wie Dynamo von Amazon [Vog07] und Riak von Basho sind auf den Einsatzzweck als verteilte Datenbank in großen Clustern optimiert. Die Skalierbarkeit erkauft man sich zum Preis einer beschränkten Funktionalität. Die Key Value Stores bieten nur wenige einfache Funktionen zum Anlegen, Lesen und Löschen von Schlüssel-Wert-Paaren, komplexe Abfragen wie „finde alle Benutzer < 40 Jahre“ sind nicht möglich.

Automatisiertes Verteilen über Consistent Hashing

Systeme wie Dynamo und Riak können die Daten eigenständig auf die Server im Cluster verteilen. Dies geschieht zum einen automatisch, was besonders wichtig ist, wenn auf viele Rechner skaliert werden muss. Denn in diesem Fall sind andere Verfahren mit hohen administrativem Aufwand belastet.

Es geschieht zum anderen sehr effizient, weil beim Hinzufügen eines weiteren (n-ten) Servers möglichst wenig Daten neuverteilt (also kopiert) werden müssen. Von jedem alten Server wird im Mittel 1/n der Daten auf den neuen Server kopiert, es müssen somit keinerlei Daten zwischen den alten Servern kopiert werden.

Erreicht wird dies durch einen besonderen Algorithmus – das Consistent Hashing (vgl. [Kra97]).

Ausfallsicherheit durch Replikation

In erster unmittelbarer Konsequenz aus einem solchen Setup ergibt sich die Notwendigkeit, automatisch mit dem Ausfall einzelner Server fertigzuwerden, denn mit der Anzahl der Server sinkt die mittlere Zeit zwischen den Ausfällen. Dazu wird jedes KeyValuePair (Schlüssel-Wert-Paar) von dem primären Server auf mehrere weitere Server repliziert.

Dazu kann man den Consistent Hashing-Ansatz so erweitern, dass dies auch wieder gleichverteilt passiert. Die auf die Server verteilten Schlüssel-Wert-Paare können auf mehrere Rechenzentren aufgeteilt werden, damit nicht ein Stromausfall in einem Rechenzentrum alles lahmlegt. Irgendwann wird dann die Verbindung zwischen zwei Rechenzentren ausfallen – eine Partition, auch „Split-Brain“ genannt, entsteht.

Wenn jetzt die Applikation in den zwei Rechenzentren ganz normal weiterläuft, wird es zu inkonsistenten replizierten Daten kommen – Daten also, bei denen die jeweiligen Repliken in beiden Rechenzentren geändert wurden, wobei die Änderungen eben nicht wieder zwischen den Rechenzentren repliziert werden konnten.

Architekturaufgabe „Konfliktbewältigung“

Der Architekt muss sich beim Design des Systems entscheiden, ob er diese Inkonsistenzen zulässt oder Änderungen an den Daten während einer Partition in einem der beiden Rechenzentren verbietet, sodass die Datenbank und in Folge die Applikation in dem anderen Rechenzentrum währenddessen nicht verfügbar ist.

Dass ein System nicht gleichzeitig Konsistenz (Consistency), Verfügbarkeit (Availability) und Toleranz (Tolerance) gegenüber Unterteilung (Partition) zur Verfügung stellen kann, wird durch das CAP-Theorem sogar theoretisch bewiesen (siehe [Bre00] und [Bre12]). In der Praxis werden die betriebswirtschaftlich Verantwortlichen dem Architekten hier häufig die klare Vorgabe machen, die ständige Verfügbarkeit von Applikation inklusive Datenbank sicherzustellen.

Der Architekt muss in einem System, das horizontal skalierbar sein soll, also die Anwendung so konzipieren, dass sie mit Inkonsistenzen umgehen kann. Von dem Key-Value Store bekommt die Applikation in diesem Fall mehrere Versionen eines Key-Value Paars geliefert und muss dann eine Konfliktauflösung durchführen. Diese kann z. B. darin bestehen, nur die letzte Version zu verwenden („Last-Write-Wins“) oder eine inhaltliche Konsolidierung der Werte durchzuführen. Bei Amazon werden zwei Versionen des Einkaufswagens dadurch konsolidiert, dass die Vereinigungsmenge genommen wird.

Tschüss Objekt-Relational-Mapping (ORM)?

Apropos Konsolidierung: Betrachten wir im nächsten Schritt das Zusammenspiel von Anwendung und Datenbank am Beispiel der Document Stores, die als besonders universell einsetzbare Gattung der nicht-relationalen Datenbanken gelten.

In Document Stores werden – wie der Name impliziert – Dokumente gespeichert. Jedes Dokument ist dabei eine Sammlung von Attributen. Jedes Attribut besteht aus einem Namen und einem Wert. Der Wert kann dabei nicht nur skalar sein – also eine Zahl, ein String oder ein Binärblock –, sondern auch ein Aggregat, also eine Liste oder ein eingebettetes Teildokument (siehe Abbildung).

Die Dokumente in Document Stores werden meistens in JSON zwischen dem Document Store und dem Applikations-Code ausgetauscht. Die Kopplung zwischen Applikations-Code und dem Document Store ist daher deutlich enger als bei relationalen Datenbanken. So ist bei Systemen mit relationalen Datenbanken häufig zwischen Applikations-Code und Datenbank noch ein Objekt-Relationaler-Mapper (ORM). Einen solchen brauchen Document Stores nicht.

Objekt-Relationaler-Mappers bringen ihre ganz eigene Menge an Problemen [New06]. Typischerweise ist der Einsatz von ORMs am Anfang sehr einfach, mit der Zeit wird der Aufwand, den man investieren muss, damit der ORM inhaltlich richtig und effizient zwischen Applikations-Code und relationaler Datenbank mittelt, immer höher – und man hat typischerweise keinen Ausweg, sondern muss diesen immer weiter steigenden Aufwand tragen.

Diese Probleme rühren bei Applikationen, die dem OOP (objektorientierte Programmierung) -Paradigma folgen, letztlich daher, dass der Applikations-Code mit seiner Objektorientierung und die relationale Datenbank eine fundamental unterschiedliche Sicht auf Objekte bzw. Datensätze haben. Man sagt, dass zwischen Applikations-Code und relationaler Datenbank ein „Impedance Mismatch“ (eine objektrelationale Unverträglichkeit) existiert. Dieser entfällt bei Document Stores bzw. ist dort deutlich kleiner.

Strukturlosigkeit planen

Eine weitere verwandte Eigenschaft von nicht-relationalen Datenbanken ist die sogenannte Schemalosigkeit. In Document Stores können zu jedem Zeitpunkt beliebige Dokumente gespeichert werden mit beliebigen Attributen und beliebigen Werten für diese Attribute.

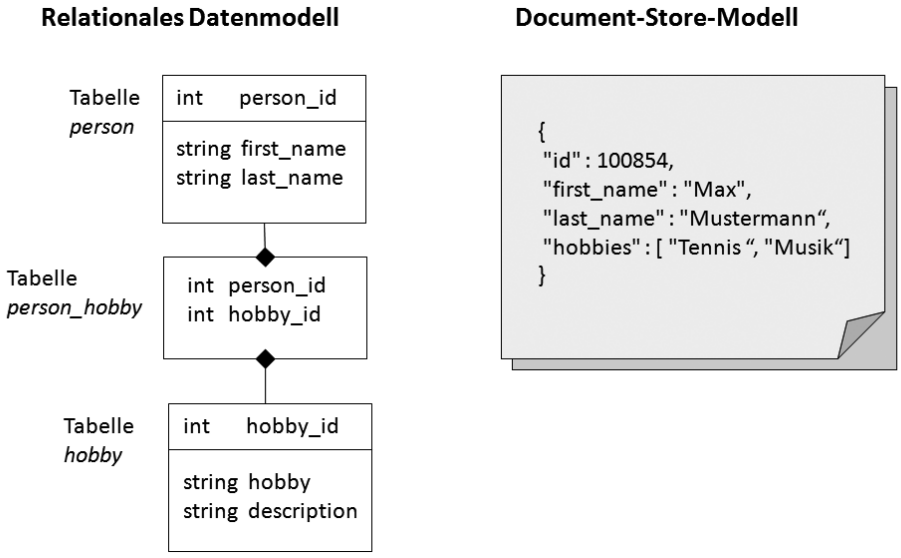

Abb.: Modellierung mit Relationen vs. Dokumenten

Bei relationalen Datenbanken muss man, bevor man einen Datensatz speichern kann, zuerst die Struktur einer Tabelle definieren. Wenn sich die Anforderungen an die Struktur der zu speichernden Daten ändern, so muss die Struktur der Tabelle mit einem „ALTER TABLE-Befehl“ geändert werden. Die Ausführung dieses Befehls kann – je nachdem wie gefüllt die Tabelle ist – lange Zeit in Anspruch nehmen, während dieser Zeit ist die Tabelle typischerweise gesperrt und kann nicht verwendet werden.

Im Gegensatz dazu muss bei den nicht-relationalen Document Stores vordergründig nichts geändert werden, denn es gibt ja tatsächlich scheinbar gar kein Schema, das man ändern könnte, geschweige denn ändern müsste.

Tatsächlich ist dies aber nicht ganz so. Denn in der Applikation gibt es Code, der davon ausgeht, dass die Dokumente in einer Sammlung bestimmte Attribute haben und dass diese Attribute mit bestimmten Werten gefüllt sind. In aller Regel kann dieser Code nicht mit beliebigen Dokumenten – die der Document Store problemlos speichern kann – umgehen.

Wenn man keine Sorgfalt walten lässt, wird die Anwendung immer häufiger auf Fehler stoßen, weil sie in Dokumenten auf unerwartete Attribute bzw. Werte trifft. Natürlich ist es ebenso unerwünscht, dass ein immer größer werdender Teil der Applikation lediglich mit den unterschiedlichen Möglichkeiten umgeht, mit denen Dokumente Attribute bzw. Werte enthalten können.

Durch sorgfältige Planung, welche Dokumente welche Attribute bzw. Werte enthalten und wie sich diese langsam und am besten jederzeit aufwärtskompatibel mit der Zeit verändern, lässt sich dieser Effekt vermeiden. Es muss also nach wie vor ein Schema bzw. eine Schemaevolution geplant werden. Diese Planung manifestiert sich jetzt nicht mehr in einer Schemadefinition in der Datenbank und Wartungsfenstern für „ALTER TABLE-Befehlen“, sondern findet im Code statt, der typischerweise logisch zwischen Applikations-Code und Document Store liegt und eine einheitliche Struktur der Dokumente sicherstellt bzw. für ältere Dokumente eine automatische Anpassung durchführt.

Database-as-a-Service

In letzter Zeit etablieren sich immer mehr Angebote, Datenbanken als Service einzusetzen, häufig im Cloud-Kontext, z. B. Cloudant Data Layer (mit CouchDB als zugrunde liegender Datenbank), SimpleDB von Amazon, DataStax (mit Cassandra als Datenbank) und diverse Angebote für MongoDB. Seit Neuestem hat Amazon mit DynamoDB ein weiteres Angebot, dabei ist DynamoDB eine Neuentwicklung auf Basis der Erfahrungen, die Amazon mit Dynamo als Datenbank für ihre eigenen Dienste gemacht hat.

Wir wollen DynamoDB [Ama12] exemplarisch etwas näher betrachten.
In DynamoDB kann der Kunde beliebig viele Tables (Tabellen) anlegen. In jedem

Table kann er Einträge, in Dynamo-Terminologie „Items“, speichern. Jedes Item besteht aus beliebig vielen Attributen, die Zahlen, Strings, Binärdaten oder Mengen solcher Werte sein können.

Die Kosten setzen sich dabei aus drei Faktoren zusammen:

- Man reserviert und bezahlt für jeden Table eine Menge an Lese- und Schreiboperationen pro Stunde. Diese Reservierung geschieht über einen programmatischen Aufruf und kann kurzfristig erhöht oder verringert werden. Diese Kosten fallen an, selbst wenn man weniger als die reservierten Operationen durchführt. Falls man versucht, mehr als die reservierten Operationen durchzuführen, führt dies zu einem Fehler.
- Zweitens muss man für den belegten Platz bezahlen. Diese Kosten ergeben sich jeweils aus dem aktuell belegten Platz für Items und Index, eine Reservierung ist hier also nicht nötig.
- Schließlich muss man für Daten, die von der Datenbank an Klienten übergeben und die nicht ebenfalls in der Amazon Cloud betrieben werden, bezahlen. Auch hier werden nur die tatsächlich anfallenden Kosten berechnet, eine Reservierung ist hier ebenfalls nicht nötig.

Mit dem Zugriff auf ein derartiges System entstehen für die architekturelle Realisierung von Systemen ganz neue Möglichkeiten. Man kann aus der eigenen Anwendung einfach auf diesen Datenbankservice zugreifen. Man muss lediglich sicherstellen, dass man zu jedem Zeitpunkt hinreichend viele Lese- und Schreiboperationen reserviert. Man muss sich nicht darum kümmern, wie viele Server dafür nötig sind, wann man mehr Server braucht, wie die Daten über diese Server zu verteilen sind, was passiert, wenn einer dieser Server ausfällt und so weiter. Man kann sich also mehr als je zuvor auf den Applikations-Code konzentrieren.

Wie in jedem Fall muss man allerdings genau die Kosten kalkulieren, denn der Aufwand für all diese Tätigkeiten ist ja nicht verschwunden. Er wird jetzt nur von Amazon übernommen und letztlich über die Kosten wieder zurückgegeben.

@tt:Es gilt festzuhalten, dass ein solches Preismodell in der relationalen Welt so nicht direkt funktionieren würde. Das liegt daran, dass in der relationalen Welt zwischen den Operationen (SELECT bzw. UPDATE Statements) und den dafür nötigen „low-level Operationen“ (in erster Nähe Block-Input/Output auf den Sekundärspeicher) stark unterschiedliche Faktoren liegen.

Wie viele Block-IO Operationen für ein SELECT nötig sind hängt z. B. davon ab, wie viele Tabellen „zusammengejoint“ werden, welche Indizes es gibt, ob die Zugriffsmuster effizientes Caching ermöglichen oder nicht, wie voll die Tabellen sind und vielem mehr. Das Preismodell, das der Provider dem Kunden also anbieten kann, macht es für den Kunden nötig, sich mit all diesen Faktoren zu beschäftigen, sodass es sich – aus Kundensicht – doch wieder eher um Infrastruktur-as-a-Service als Platform-as-a-Service handelt.

Fazit

Die vorangegangenen Ausführungen haben einige Aspekte von nicht-relationalen Datenbanken beleuchtet, die explizit oder implizit Auswirkungen auf die Arbeit von Architekten haben. Die „neuen“ Datenbanken bieten neue Optionen, hochskalierbare Anwendungen mit überschaubarem Aufwand und Risiko zu realisieren, flexibel und agil zu entwickeln und Features wie Empfehlungsmanagement performant umzusetzen. Wer möchte, kann den Betrieb der Datenbank an spezialisierte Anbieter outsourcen zu transparent kalkulierbaren Konditionen.

Grundvoraussetzung für den dauerhaft erfolgreichen Betrieb von Anwendungen mit (heterogenen) nicht-relationalen Datenbanken ist eine sorgfältige Planung auf Gesamt- und Detailebene, eine saubere Komponentenkapselung und wie bei relationalen Datenbanken ein sorgfältiges Schemamanagement. Ebenfalls darf die Architektur nicht ignorieren, dass Verfügbarkeit und Konsistenz in verteilten Systemen, in denen Partitionen mit an Sicherheit grenzender Wahrscheinlichkeit passieren, nicht gleichzeitig zu erzielen sind. ■

Referenzen

[Fow12] Martin Fowler, „NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence“, Addison-Wesley Longman, 2012

[Vog07] Werner Vogels et.al., „Dynamo: Amazon‘s Highly Available Key-Value Store“, Proceedings of the 21st ACM Symposium on Operating Systems Principles, 2007, http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[Kra97] David Karger et.al., „Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web“, Proceedings of the 29th Annual ACM Symposium on Theory of Computing., 1997, http://thor.cs.ucsb.edu/~ravenben/papers/coreos/KLL+97.pdf

[Bre00] Dr. Eric A. Brewer, „Toward Robust Distributed Systems“, PODC Keynote, 2000, http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[Bre12] Dr. Eric A. Brewer, „CAP Twelve Years Later“, IEEE Computer, Feb 2012, http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

[New06] Ted Neward, „ORM - The Vietnam of Computer Science“, 2006, http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx