Neue Serie: Geschichte von NoSQL ▸ 82

S&S

JAVA Mag

Deutschland €9,80 Österreich € 10,80 Schweiz sFr 19,50 Luxemburg € 11,15 5.2015

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

JNI, Rootbeer, CUDA
Parallelisierung auf der Grafikkarte ▸20

Apache Tamaya
Konfiguration in Java ▸91

RESILIENCE

Einführung in Resilient Software Design ▸ 33

Spring Cloud: Resilience für Microservices ▸ 53

Chaos-Engineering bei **Netflix** ▸ 44

Michael Nygard über sein Circuit Breaker Pattern ▸ 58

4 194586 709801 05

Sustainable Service Design: Nachhaltig – oder nur langlebig? ▸ 62

gulp: Frontend-Build-System auf Node.js-Basis ▸ 96

Android 5 Lollipop: Ein Blick auf die neuen Nexus-Geräte ▸ 122

NEUE SERIE

Teil 1: Relationale Datenbanksysteme

Eine kleine Reise durch NoSQL

Not only SQL, kurz NoSQL, ist eine noch relativ junge Disziplin der Informatik, die sich mit Datenspeicherungstechnologien beschäftigt. NoSQL befindet sich in beständigem Wandel, was es schwer macht, eine strenge Definition zu geben. In dieser dreiteiligen Artikelserie wollen wir versuchen, die bisherige Entwicklung in groben Zügen nachzuvollziehen. Wir beginnen mit einem Rückblick auf die relationalen Datenbanksysteme.

jaxSpeaker

von Christian Mennerich und Joachim Arrasz

NoSQL ist momentan in aller Munde und erfährt eine Namensgebung und -prägung, die es sehr schwer macht, eine strenge Definition dafür zu geben, was sich genau hinter diesem Akronym verbirgt. Das ursprünglich erklärte Ziel von NoSQL war die Entwicklung von so genannten Web-Scale-Datenbanken [1], jedoch gibt es viele verschiedene Auffassungen und Meinungen über das, was unter NoSQL zu verstehen ist.

Wir wollen die Entstehung von NoSQL näher beleuchten und beginnen unsere dreiteilige Serie über NoSQL mit einem für das Verständnis notwendigen Rückblick auf die relationalen Datenbanken und die Probleme, die das relationale Datenmodell zu lösen versuchte. Im nächsten Artikel werden wir uns dann intensiver den NoSQL-Datenbanken selbst widmen, um im dritten Teil neue Trends zu prognostizieren und zu diskutieren.

Die Evolution der Speicherungstechnologien seit den 1970er-Jahren machen wir an drei Arbeiten fest, die uns durch unsere Serie begleiten werden.

Artikelserie

Teil 1: Relationale Datenbanksysteme
Teil 2: NoSQL-Datenbanken
Teil 3: Ausblick

Der Ausgangspunkt ist Codds Arbeit zu relationalen Datenbanksystemen [2] aus dem Jahr 1970, in der die Grundsteine des relationalen Datenbankmodells gelegt werden. Den Konsequenzen der zunehmenden Verteilung von Daten trägt der Beweis der Brewer'schen Vermutung Rechnung, der 2002 durch Gilbert und Lynch präsentiert wurde. Er ist als CAP-Theorem bekannt und stellt, streng interpretiert, ein „Wähle zwei aus drei"-Ergebnis dar, mit gewichtigen Implikationen für die verteilte Datenhaltung. Die dritte Arbeit ist die 2005 verfasste Absage an relationale Datenbanken als Universallösungen durch Stonebraker und Çetintemel [3]. Sie versetzte dem Vorgehen, für jedes Problem eine relationale Datenbank zu verwenden, den etablierten Todesstoß.

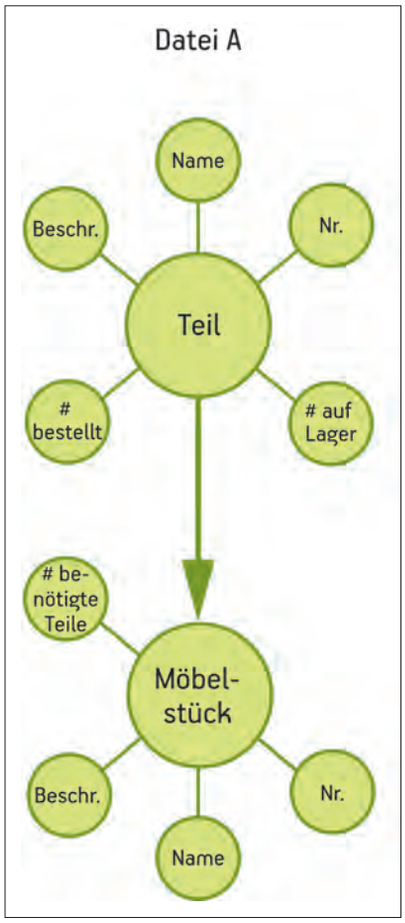

Abb. 1: Modell, das Möbelstücke den Teilen unterordnet

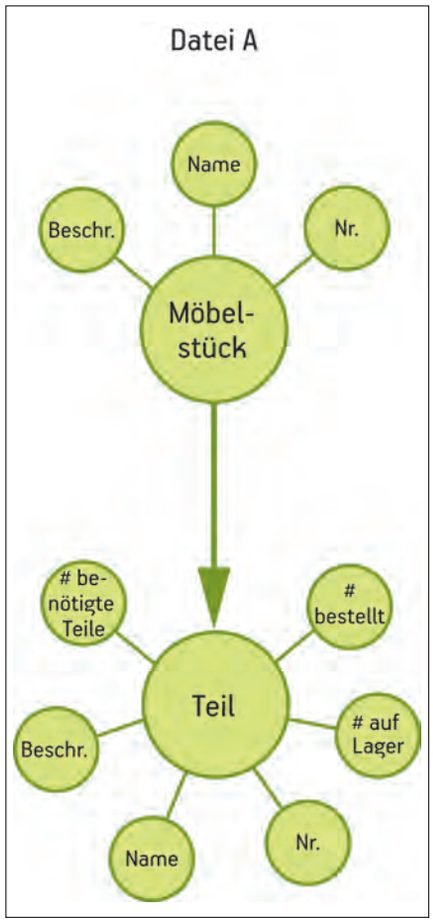

Abb. 2: Modell, das Teile den Möbelstücken unterordnet

Codds Kritik und das relationale Datenmodell

Die in den 1960er-Jahren vorherrschenden Datenspeichertechnologien waren Systeme wie IBMs IMS oder IDS von General Electric, in denen Daten hierarchisch oder in Netzwerken organisiert sind. Codd formulierte drei wesentliche Kritikpunkte [2]:

1. Eine Applikation hängt direkt davon ab, wie die Daten auf dem Persistenzspeicher angeordnet sind (Ordering Dependence).
2. Eine Anfrage an ein Datenbanksystem hängt direkt davon ab, welche Indexstrukturen definiert sind (Indexing Dependence).
3. Eine Applikation hängt direkt davon ab, über welche Zugriffspfade die auf dem Speichersystem persistierten Daten untereinander vernetzt sind (Access Path Dependence).

Diese Abhängigkeiten erlegen dem Anwendungsentwickler Bürden auf. Er muss wissen, wie die Daten auf dem Speichersystem abgelegt sind, ob Indizes existieren, wie diese benannt sind und wie von einem Datensatz zu einem anderen zu navigieren ist, um die für seine Anwendung relevanten Informationen zu finden. Jede Änderung an der Anordnung der Daten, den Indexstrukturen oder den Zugriffspfaden kann dazu führen, dass eine Applikation nicht mehr wie erwartet funktioniert oder gar vollständig auseinanderbricht.

Zur Verdeutlichung der Abhängigkeiten nehmen wir an, wir hätten eine Anwendung für einen Möbelhandel zu schreiben, der seine Möbelstücke in Einzelteilen zum eigenständigen Zusammenbauen ausliefert. Dazu benötigt er ein Verwaltungssystem, in dem festgehalten ist, wie viele Teile welcher Art vorhanden sowie bestellt sind und wie viele Teile für den Zusammenbau eines bestimmten Möbelstücks benötigt werden.

Die vielleicht naheliegendste Modellierung ist, die Möbelstücke den Teilen strukturell unterzuordnen und die Beziehung hierarchisch in einer Datei abzulegen (**Abb. 1**).

Eine zweite Möglichkeit geht umgekehrt an den Sachverhalt heran. Die Teile werden den Möbelstücken untergeordnet, für die sie gebraucht werden. Diese hierarchische Beziehung wird wieder in einer einzigen Datei abgelegt (**Abb. 2**).

Eine dritte Möglichkeit betrachtet Möbelstücke und Teile einander gleichgestellt und speichert sie auch in verschiedenen Dateien. Die Beziehungen zwischen den Möbelstücken und den Teilen wird über eine zusätzliche Struktur modelliert, die Möbelstücke und Teile verknüpft (**Abb. 3**).

Weitere Möglichkeiten der Datenhaltung sind denkbar, eine ausführlichere Diskussion kann bei Codd [2] nachgelesen werden. Im Allgemeinen ist es für eine Anwendung nicht realistisch zu prüfen, welche der möglichen Darstellungen gewählt worden ist. Der Anwendungsentwickler muss die gewählte Speicherart wissen, um Bestandsabfragen und -pflege implementieren zu können. Im Falle einer Änderung müssen alle Anwendungsprogramme, die sich auf die alten Strukturen verlassen, nachgepflegt und neu ausgerollt werden.

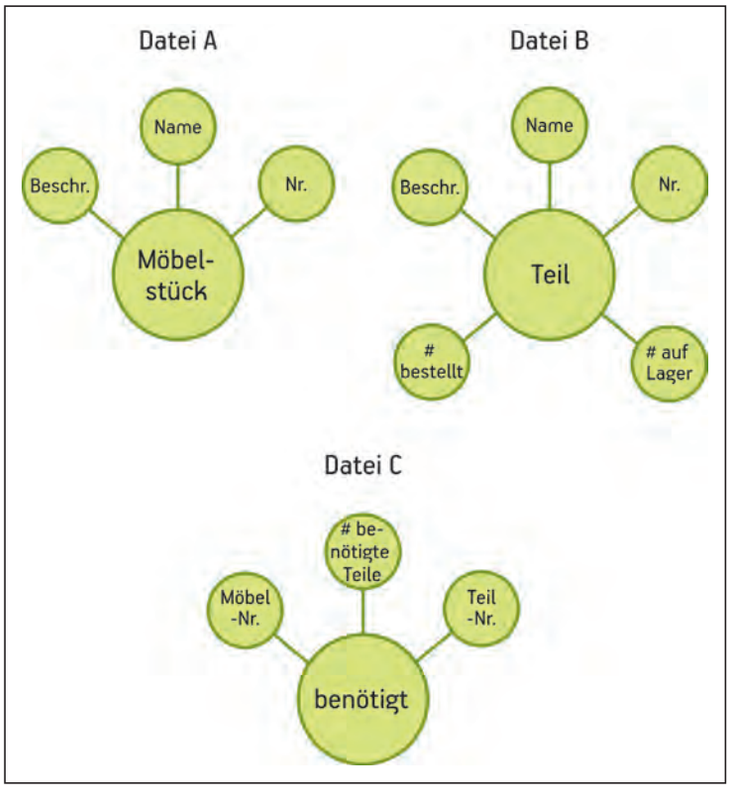

Abb. 3: Modell, das Teile und Möbelstücke gleichstellt

Das relationale Datenmodell nach Codd

Das relationale Datenmodell wie es Codd vorschlägt [4], bietet einen möglichen Ausweg aus dieser Bredouille. Es schlägt die Separation von Datenmodell und Datenspeicherung vor, die Trennung in ein logisches und ein physikalisches Datenmodell.

Hinweis 1

Strukturierte Werte in den Attributen sind im relationalen Datenmodell erst einmal nicht erlaubt. Dies ist als erste Normalform (1NF) bekannt und hat wichtige Konsequenzen für die Modellierung. Es gibt aber Datenbanken, in denen diese Bedingung abgeschwächt wurde. Erweiterungen des relationalen Modells und Non-First-Normal-Form-(NF^2-)Systeme erlauben auch das Speichern strukturierter Werte oder geschachtelter Relationen.

Hinweis 2

Der simple Ansatz der Speicherung von Tupeln in Tabellen kann zu unerwünschten Anomalien führen, die durch Datenredundanz verursacht werden. Relationale Datenbankschemata werden deshalb normalisiert und von Redundanz befreit. In der Literatur finden sich viele unterschiedliche Normalformen. In der Regel sind Schemata bis zur dritten Normalform (3NF) [5] mit intuitiver Modellierung erreichbar, wenn auf das Vermeiden von Redundanz geachtet wird. Allerdings wird in der Praxis häufig auf das Normalisieren verzichtet und Datenredundanz zugunsten von Performance der Datenabfragen in Kauf genommen.

Basierend auf der mathematischen Mengentheorie werden Daten als Mengen geordneter Tupel, so genannte *Relationen*, beschrieben. Eine wichtige Bedingung ist, dass die zugrunde liegenden Mengen nur atomare Wertebereiche haben dürfen, die Tupel sind also flach. Die Struktur der Tupel ist ein Datenbankschema, eine Menge von Relationen ist eine Datenbankinstanz. Häufig wird von Tabellen statt von Relationen gesprochen. Die Tupel sind deren Zeilen, die Namen der Spalten heißen Attribute.

Die gespeicherten Daten werden mittels Anfragesprachen, die gegen das Datenbankschema formuliert werden, wieder aus der Datenbank herausgeholt. Codd schlug für das relationale Datenmodell die relationale Algebra als Anfragesprache vor, deren Operatoren Relationen als Eingabe erhalten und neue Relationen als Ergebnis erzeugen.

Die Structured Query Language, SQL, ist heute die am weitesten verbreitete Anfragesprache für relationale Datenbanksysteme. Als ein Nachfolger der Sprache SEQUEL [6] beruhen die Implementierungen von SQL auf der relationalen Algebra.

Allerdings hat fast jede relationale Datenbank einen eigenen SQL-Dialekt umgesetzt, was die Austauschbarkeit des Datenbanksystems erschwert oder sogar verhindert. Auch die Unterstützung von Erweiterungen oder NF^2-Eigenschaften ist weder überall vorhanden noch standardisiert. Die Datenbank PostgreSQL [7] beispielsweise erlaubt das direkte Auswerten von in Tupeln gespeicherten JSON-Dokumenten.

Semantik und Konsistenz

Der innere Zusammenhang zwischen den gespeicherten Daten wird durch referenzielle Integrität beschrieben. Hierzu werden Primärschlüssel und Fremdschlüsselbeziehungen definiert, die Eindeutigkeit von Werten in

Hinweis 3

Die algebraisch abgeschlossene (prozedurale) relationale Algebra ist logisch äquivalent zum (deklarativen) Tupelkalkül (Tuple-orientied Calculus, TRC), einer prädikatenlogischen Sprache erster Stufe. Damit kann erwartet werden, Fragen, die natürlichsprachlich formulierbar sind, auch als SQL-Anfragen stellen zu können.

Datenbankoptimierer machen starken Gebrauch von den Eigenschaften des TRC. SQL-Anfragen werden in TRC-Ausdrücke transformiert und dann mittels logisch äquivalenter Umformungen in performante auszuwertende Ausdrücke umgewandelt.

Weiter dient die relationale Algebra oft als Referenz für die Aussagekraft von Anfragesprachen. Mit ihr müssen sich andere Sprachen vergleichen lassen. Dabei ist immer zu beachten, wie die Rahmenbedingungen zur Interpretation der relationalen Algebra festgelegt sind. Ausführliche Diskussionen zu diesen Themen finden sich z. B. in [5] oder [8].

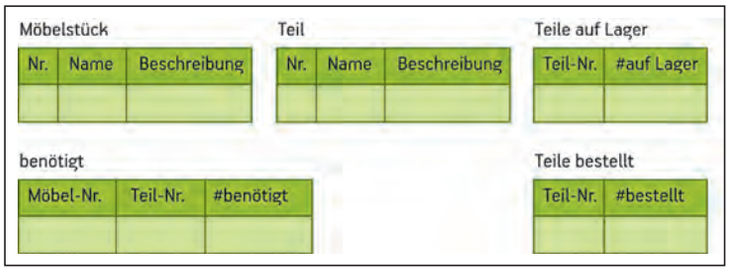

Abb. 4: Normalisiertes relationales Schema

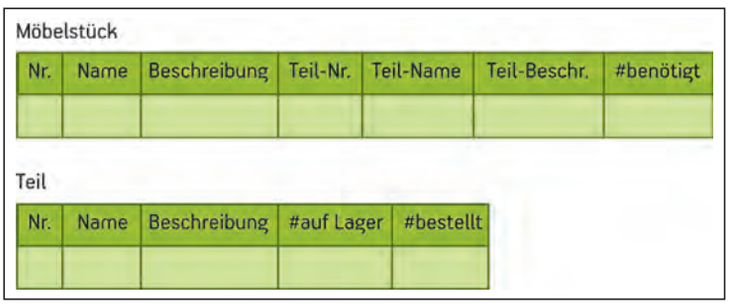

Abb. 5: Denormalisiertes relationales Schema

Attributen oder die Gültigkeit anderer funktionaler Abhängigkeiten gefordert.

Transaktionen sind Gruppierungen von Operationen in der Datenbank, die dafür sorgen, dass die referenzielle Integrität im Betrieb erhalten bleibt. Kernkonzept ist das ACID-Paradigma: Eine Transaktion

- wird entweder vollständig oder gar nicht ausgeführt (Atomicity)
- überführt die Datenbank von einem konsistenten Zustand in einen weiteren konsistenten Zustand (Consistency)
- sieht die Datenbank so, als wäre sie die einzige momentan laufende Transaktion (Isolation)
- hat nach ihrem Beenden alle ihre Änderungen dauerhaft gespeichert (Durability)

Die Operationen einer Transaktionen werden also

- vollständig ausgeführt (Commit),
- oder gar nicht (Rollback).

Inkonsistente Zwischenzustände sind vorübergehend erlaubt, diese sehen andere Transaktionen aber nicht. Um die ACID-Eigenschaften zu gewährleisten, werden in der Regel sperrende Verfahren (Locks) und blockierende Protokolle wie ein Zwei-Phasen-Commit-Protokoll verwendet. Die Granularität der Sperren kann variieren. In vielen relationalen Datenbanksystemen ist der Konsistenzlevel in gewissem Rahmen veränderbar.

Hinweis 4

Für Primärschlüssel wird vom System automatisch ein Index angelegt. Indizes sind auch für kombinierte Attribute möglich und nicht auf ein einzelnes Attribut beschränkt.

Jeder Index wird vom Datenbanksystem bei Änderungen am Datenbestand aktualisiert. Es ist also darauf zu achten, es mit der Anzahl der Indizes auf einer Relation nicht zu übertreiben, da sich eine hohe Anzahl negativ auf die Performance von Einfüge- und Updateoperationen auswirken wird. Einen Index für jede mögliche Kombination von Attributen anzulegen, ist in Systemen, die nicht nur lesen, absolut zu vermeiden.

In Data Warehouses ist es häufig so, dass selten (große Mengen) Daten eingefügt werden. Diese Daten werden in der Regel nicht aktualisiert; zum Zwecke des Reporting werden (komplexe) Leseoperationen ausgeführt. Hier kann ggf. großzügiger mit Indizes umgegangen werden.

Normalerweise sind es auch bei großen Tabellen nur wenige Attribute, über die häufig zugegriffen wird. Hier ist es sinnvoll, typische Anfragen zu identifizieren, um die benötigten Indizes festzulegen.

Bei Indizes, die mehrere Attribute beinhalten, ist es oft sinnvoll, diese absteigend nach Selektivität der einzelnen Attribute anzuordnen, damit das System den Index auch bei Anfragen, die nur eine Teilmenge der Attribute des Indexes beinhalten, nutzen kann.

Zugriffsstrukturen

Anfragen werden vom Datenbanksystem transparent in Zugriffspfade auf dem Persistenzspeicher übersetzt. Indexstrukturen werden neben den eigentlichen Daten gepflegt, ihre Existenz hat keinen Einfluss auf den Informationsgehalt der gespeicherten Daten. Auch muss der Anwendungsentwickler von ihnen keine Kenntnis haben, um Anfragen formulieren zu können, da in Anfragen keine Indizes auftauchen.

Dem Datenbanksystem obliegt es, die Indizes auch tatsächlich zu nutzen. Ein Index kann die Performance einer Anfrage stark verbessern, ohne dass diese selbst angepasst werden muss. Das macht Anfragen robust gegen Änderungen der Indexstrukturen. Anfragen und Indizes können unabhängig voneinander erstellt und gepflegt werden.

Das relationale Modell: Theorie und Praxis

In der Theorie liefern die oben diskutierten Konzepte die Basis für ein universell-aussagekräftiges Datenmodellierungs- und Datenspeicherwerkzeug mit standardisierter Anfragesprache. So wird es möglich, Daten dauerhaft zu speichern und wieder auszulesen und sich gleichzeitig über den Zustand der Daten jederzeit sicher sein zu können. **Abbildung 4** zeigt ein normalisiertes relationales Schema zum eingangs diskutierten Möbelhandelbeispiel.

Die Praxis sieht, wie so oft, anders aus. Die Erfahrung hat hier gezeigt, dass Schemata im Nachhinein häufig denormalisiert werden müssen, um den Anforderungen an das Anfrageverhalten zu genügen. Auf die Normalisierung wird von daher, oft bewusst, von vorne herein verzichtet, um Performanceproblemen vorzubeugen. Durch Redundanz hervorgerufene Anomalien werden dabei wissentlich in Kauf genommen und deren Behandlung in die Anwendung verlegt.

Performanceeinbußen können daher rühren, dass das Datenbanksystem die Zugriffspfade der Anfragen zwar vor dem Entwickler versteckt, die durch die Nor-

malisierung hervorgerufene Verteilung der Daten aber dennoch durchscheinen und sich negativ bemerkbar machen kann. (Das Phänomen, dass Eigenschaften durch alle Abstraktionsschichten durchscheinen können, hat Spolsky bereits vor mehr als zehn Jahren treffend beschrieben [9].)

Schauen wir wieder auf unser Beispiel von oben. Es ist wahrscheinlich, dass typische Anfragen zu Möbelstücken auch Informationen zu den Teilen, aus denen sie zusammengebaut werden, beinhalten. Dies legt es nahe, einen Teil der Daten redundant vorzuhalten, um das Joinen der Daten zu vermeiden. In der Anwendung zur Pflege der Daten muss jetzt allerdings darauf geachtet werden, Änderungen der Eigenschaften eines Teils in unterschiedlichen Tabellen zu pflegen, um eine konsistente Repräsentation der Daten zu gewährleisten (**Abb. 5**).

Die relationalen Datenbanken haben seit den 1970er-Jahren einen fulminanten Siegeszug als Universaldatenspeicher angetreten. Die Verwendung eines Datenspeichers ist oft eine fast nicht funktionale Anforderung: In vielen Unternehmen stand (und steht) ein, gegebenenfalls teuer betriebenes, relationales Datenbanksystem als universeller Speicher für die Daten einer Vielzahl von Anwendungen. Dies birgt nicht nur die Gefahr, sich an einen Hersteller zu binden (Vendor Lock-in) und abhängig von dessen Betriebserfahrung und Supportleistungen zu werden, es hängen damit auch viele Applikationen von den Wartungszyklen der Datenbank ab. Verfügbarkeit, Ausfallsicherheit und Datensicherheit des zentralen Mollochs müssen kostspielig gewährleistet werden.

Die Hersteller relationaler Datenbanksysteme haben sich diese Situation zunutze gemacht und versuchen jedes Problem relational erscheinen zu lassen. Oft wurden (und werden) bei den Anstrengungen, jedes Problem in eine relationale Struktur zu pressen, große Geldsummen in Unternehmen verschwendet.

Programmieren gegen relationale Datenbanken

ACID bietet dem Entwickler eine sichere Ausgangsbasis für das Schreiben zuverlässiger Anwendungen. Transaktionen bilden solide Grundbausteine für Abstraktionen, aus denen eine Anwendung zusammengesetzt werden kann. Eine Anwendung muss sich nicht um inkonsistente Zwischenzustände aufgrund fehlgeschlagener Transaktionen kümmern.

Zahlreiche Hilfsmittel unterstützen den Anwendungsentwickler bei seiner täglichen Arbeit mit relationalen Datenbanksystemen. Das Java Persistence API (JPA) [10] bietet ein Programmierinterface zur Persistierung von Daten in relationalen Datenbanksystemen.

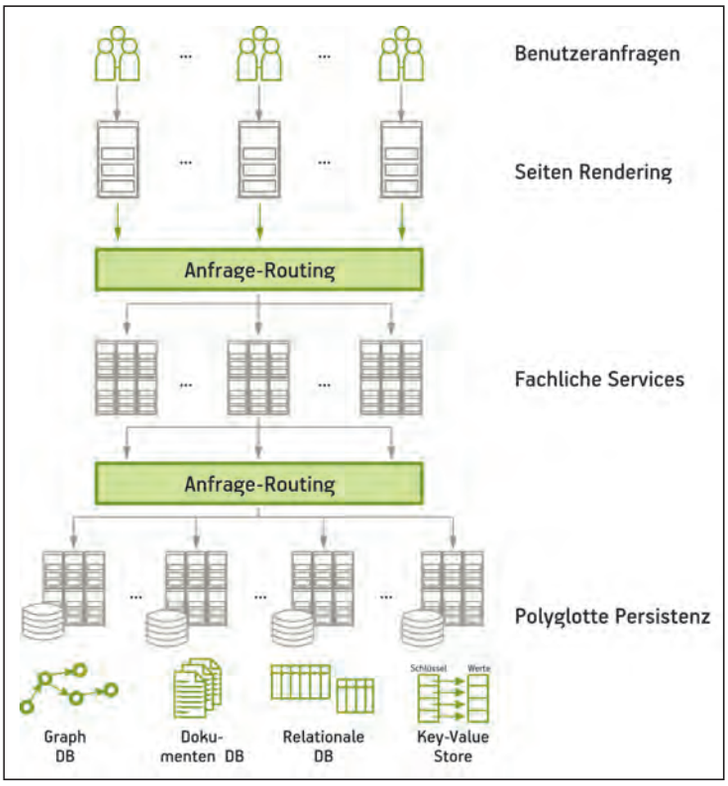

Abb. 6: Architektur mit polyglotter Persistenzschicht

Object-relational Mapper (ORM) übernehmen dann die Verwaltung der Korrespondenz zwischen den Objekten der Anwendungsschicht und ihrer Repräsentation und Persistierung im relationalen Datenbanksystem. Diese Korrespondenz ist für den Entwickler transparent und abstrahiert von der Verteilung der Daten auf unterschiedliche Tabellen. Mit Hibernate ORM [11] steht eine freie Implementierung von JPA zur Verfügung. Allerdings bietet Hibernate ORM auch eigene Funktionalitäten an, die über den für JPA definierten Standard hinausgehen. Hier ist Vorsicht bei der Verwendung geboten: Greift die Anwendung auf diese Funktionen zurück, so ist die Austauschbarkeit des ORM nicht mehr gewährleistet.

Technologien wie z. B. Spring Data JPA [12] oder das Spring-JDBC-Template [13] stellen Mechanismen zur Verfügung, die den Entwickler weiter unterstützen und das Entwickeln robuster Anwendungen, die auf relationale Datenbanken als Speichertechnologie setzen, sichern.

Mit *PreparedStatements* [14] steht ein Mittel zur Verfügung, um das Datenbanksystem bei der optimierten Ausführung zu unterstützen und Böswilligkeiten wie SQL Injection vorzubeugen.

Tipp

Die Autoren Joachim Arrasz und Christian Mennerich werden auf der diesjährigen JAX 2015 einen Vortrag halten. Das Thema lautet „Erfahrungsbericht: Umgekehrte Architekturanalyse – Datastore Evaluation". Der Vortrag findet am Dienstag, 21.04.2015 von 11:45 bis 12:45 statt.

Auf Basis der Möglichkeiten, die Frameworks wie Spring Data bieten, können einige der im nachfolgenden Abschnitt besprochenen Kritikpunkte abgeschwächt werden. Beispielsweise kann die Komplexität von SQL-Anfragen hinter geschickten Mappings versteckt werden. Das macht aber die Konfiguration von Transaktionen und kaskadierenden Interaktionen kompliziert. Hier gilt wie so oft, dass sich Komplexität zwar verschieben, nicht aber eliminieren lässt. Allerdings bringen diese Techniken deutlich sichtbare Vorteile für den Alltag des Anwendungsentwicklers. Trotz allem darf die Komplexität bei der Fehlersuche, welche mit jeder weiteren Abstraktion steigt, aber nicht unterschätzt werden.

Relationale Systeme in der Kritik

Aufgrund der Normalisierung sind komplexe Objekte der Geschäftsdomäne zumeist auf mehrere Relationen verteilt. Diese als Impedance Mismatch [15] bekannte Tatsache macht SQL-Anfragen schnell kompliziert und unübersichtlich, was zu Lasten der Wartbarkeit der Anfragen geht.

Das Zusammensuchen der auf dem Speichersystem verteilten Daten zum Befüllen der Objekte kostet Zeit und Rechenleistung. Hier bieten Systeme ausgefeilte Caching- und Optimierungsmechanismen. Bei Anfragen mit vielen Join-Operationen sind die I/O-Zeiten aber deutlich spürbar. Eine weitere Maßnahme zur Performancesteigerung ist die vertikale Skalierung, der Einbau von mehr RAM, CPUs oder schnelleren I/O-Subsystemen, gemäß der Maxime „viel hilft viel". Denormalisierung des Schemas kann die Performance steigern, bringt aber wieder mehr Verantwortung in die Anwendung zurück.

Die Performance einer Anfrage ist weiterhin abhängig von der Abarbeitung durch das relationale Datenbanksystem. Äquivalente Umformulierungen der Anfragen durch den Entwickler können zu einer signifikanten Verbesserung führen, wenn dieser die Daten kennt. Eine Daumenregel ist, Anfragen so zu formulieren, dass die Relationen möglichst früh möglichst wenige Tupel enthalten. Weiter kann der Anfrageoptimierer gezwungen werden, gewisse Zugriffpfade und Indizes zu verwenden. Dies bringt aber eine Abhängigkeit vom Zustand des Systems mit sich und weicht die Trennung von Anfrage- und Datenspeicherungsebene auf. Die Arbeitsweise des Optimierers ist darüber hinaus von der Version des Datenbanksystems abhängig, eine performante Anfrage kann nach einem Update hoffnungslos ineffizient sein.

Weitere Schwächen offenbaren relationale Datenbanken bei der Skalierung im Umfeld der fortschreitenden Verteilung der Datenwelt. Der vertikalen Skalierung sind physikalische und monetäre Grenzen gesetzt. Sperren und blockierende Verfahren, die die Transaktionalität

des Systems sichern, skalieren horizontal nur schlecht. Eine Verteilung der relationalen Datenbank kommt mit zum Teil sehr deutlichen Performanceeinbußen daher, da Netzwerklatenzen sich dann bis in die Sperrverfahren durchschlagen. In der Konsequenz laufen Transaktionen länger und die Antwortzeiten der Applikation erhöhen sich, der Durchsatz sinkt.

Verfahren für den Umgang mit Hochverfügbarkeitsanforderungen (High-Availability) und Lastverteilung beruhen meist darauf, die relationale Datenbank weiterhin als ein System erscheinen zu lassen. Verteilung und Replikation der Daten werden transparent gehandhabt, das Gewährleisten der strikten Konsistenzanforderungen kostet wertvolle Antwortzeit.

In einigen Bereichen haben sich relationale Datenbanken als nicht einsetzbar erwiesen. Ein Beispiel kann ein Webshop mit Anforderungen und SLAs sein, die ein Antwortverhalten fordern, das mit dem Overhead eines relationalen Systems schlicht nicht erreichbar ist. Eine gute Beschreibung der Defizite relationaler Systeme in diesem Bereich findet sich in [16].

Fazit

Die genannten Schwächen, insbesondere im Bereich der horizontalen Skalierung und dem Antwortverhalten, sind in den letzten Jahren mit zunehmender Verteilung der Daten und geänderten Anforderungen an die Datenkonsistenz schwerwiegender geworden. Im Zuge der Diskussionen um Big Data und Web Scale ist eine Vielzahl neuer Systeme entstanden, die versuchen, das relational geprägte Denken zu revolutionieren und den Fokus bei der Wahl der Speichertechnologie wieder verstärkt auf die Anforderungen der Anwendungsdomäne zu lenken. In den Vordergrund rücken die Wahl des richtigen Werkzeugs für das vorliegende Problem und der richtigen Datenbank für das gewählte Datenmodell. Die Universalität der relationalen Datenbanken ist zur Marketingillusion erklärt worden [3]. Im Beispiel des Webshops kann auf neue Technologien zurückgegriffen werden. **Abbildung 6** zeigt eine mögliche Architektur, die von verschiedenen Speichertechnologien Gebrauch macht, um speziellen Anforderungen gerecht zu werden (vgl. auch [16]). Auf die genannten Datenbankarten gehen wir im nächsten Teil der Serie gezielter ein.

Nicht mehr alles in ein relationales Schema pressen zu müssen, wird als Befreiung empfunden. Domänenmodelle können natürlicher erstellt werden, wenn die Transformation in ein relationales Modell wegfällt. Manchmal **ist** die Domäne ein Graph oder eine schlichte Menge von Schlüssel-Werte-Paaren. Wenn die gewählte Art der Modellierung direkt von der Datenbank unterstützt wird, so fühlt sich das nicht nur besser an, es gibt auch mehr Sicherheit und Vertrauen in das Modell. Dies beschleunigt die Entwicklung und macht die Evolution des Modells weniger fehleranfällig.

Geminderte Konsistenz (sofern die Anwendungsdomäne es erlaubt) ist nur eine Technik, Performance und Datendurchsatz zu steigern. Allerdings kehrt auch wieder mehr Verantwortung zum Anwendungsentwickler zurück. Die Regeln der verteilten Datenhaltung sind strikt, die Implikationen des CAP-Theorems [3] spielen eine zentrale Rolle. Die NoSQL-Bewegung befindet sich hier teilweise in einer Situation, die mit der in den 1970er vergleichbar ist. Und vielleicht kann aus den Entwicklungen in der Vergangenheit gelernt werden.

NoSQL-Datenbanken und Analogien zwischen der aktuellen Entwicklung im Bereich NoSQL und der Situation der Anwendungsentwickler zu Codds Zeiten werden wir im nächsten Teil dieser Serie ausführlicher diskutieren.

Christian Mennerich hat Diplom-Informatik studiert, einer seiner Schwerpunkte lag auf Theorie und Implementation von Datenbanksystemen. Er arbeitet als Entwickler bei der synyx GmbH & Co. KG in Karlsruhe, wo er sich unter anderem mit NoSQL beschäftigt.

@cmennerich

Joachim Arrasz ist als Software- und Systemarchitekt in Karlsruhe bei der synyx GmbH & Co. KG als Leiter der CodeClinic tätig. Darüber hinaus twittert (@arrasz) und bloggt er gerne (http://blog.synyx.de/).

Links & Literatur

[1] Edlich, S. et al.: „NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken“, Carl Hanser Verlag, 2011

[2] Codd, E. F.: „A Relational Model of Data for Large Shared Data Banks“, Communications of the ACM, Vol. 13:6, 1970

[3] Stonebraker, M.; Çetintemel, U.: „One Size Fits All: An Idea Whose Time Has Come and Gone“, Proceedings of the 21st International Conference on Data Engineering, 2005

[4] Gilbert, S.; Lynch, N.: „Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services“, ACM SIGACT News, Volume 33 Issue 2 (2002)

[5] Kemper, A.; Eickler, A.: „Datenbanksysteme – Eine Einführung“, 8. Auflage, Oldenbourg Wissenschaftsverlag, 2011

[6] Chamberlin, D. C.; Boyce, R. F.: „SEQUEL: A structured English query language“, in „Proceedings of the 1974 ACM SIGFIDET workshop on Data description, access and control“, 1974

[7] http://www.postgresql.org/docs/9.4/static/functions-json.html

[8] Kandzia, P.; Klein, H.-J.: „Theoretische Grundlagen relationaler Datenbanksysteme“, BI Wissenschaftsverlag, 1993

[9] Spolsky, J.: „The Law of Leaky Abstractions“, 2002: http://www.joelonsoftware.com/articles/LeakyAbstractions.html

[10] http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[11] http://hibernate.org/orm/

[12] http://projects.spring.io/spring-data-jpa/

[13] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/jdbc.html

[14] http://de.wikipedia.org/wiki/Prepared_Statement

[15] http://de.wikipedia.org/wiki/Object-relational_impedance_mismatch

[16] DeCandia, G. et al.: „Dynamo: Amazon's Highly Available Key-value Store“, in: „SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles“, ACM, 2007