Von NoSQL nach NewSQL: Welche Speicherstrategie wählen? ▸92

S&S

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 7.2015

JAVA Mag

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

GUI-Entwicklung
Reactive Programming mit JavaFX ▸18

Webentwicklung
Qualität von Websystemen planen ▸86

Web Components

Eigene Web Components erstellen und veröffentlichen ▸52

Anspruchsvolle Single Page Applications ▸68

Polymer: Produktive Entwicklung von Web Components ▸63

DukeScript: Cross-Plattform-Anwendungen mit Java ▸23

Kunden im Cluster: Finden von Ähnlichkeitsstrukturen ▸100

Spring IO: Projektionen in Enterprise-Anwendungen ▸40

Teil 3: NewSQL und Ausblick

Eine kleine Reise durch NoSQL

Bisher haben wir NoSQL-Datenbanken und ihre junge Evolutionsgeschichte betrachtet. Wir haben Parallelen zur Entwicklung der relationalen Systeme aufgezeigt und NoSQL-Systeme gemäß der üblichen Klassifikation diskutiert. Dieser dritte und letzte Teil unserer kleinen Reise greift einige Punkte noch einmal auf, um aktuelle Trends zu beleuchten. Wir schauen insbesondere auf Multi-Modell-Datenbanken und NewSQL-Systeme. Abschließend wagen wir ein paar Thesen über mögliche Entwicklungen in der nahen und mittelfernen Zukunft.

von Christian Mennerich und Joachim Arrasz

In den ersten beiden Teilen unserer kleinen Serie haben wir zunächst die relationalen Datenbanksysteme mit ihren Vor- und Nachteilen im Kontext ihrer Entstehungsgeschichte betrachtet. Zu Recht haben die relationalen Datenbanksysteme ihren Ruf als Universalsysteme eingebüßt.

Artikelserie

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

Wir haben die relationalen Systeme bisher immer als ein Gesamtkonzept betrachtet: Dieses schließt das relationale Datenmodell und ACID-Transaktionalität genauso mit ein wie die Mittel zur technischen Implementierung. Die Techniken stammen allerdings zumeist aus den 1970er Jahren und sind heute oft nicht mehr zeitgemäß. Das Bild eines behäbigen alten Elefanten für diese Systeme erscheint hier also durchaus als passend.

NoSQL-Datenbanken versuchen nun in einigen Bereichen, die Mankos der gemütlichen Riesen auszugleichen. Durch Flexibilität und horizontale Skalierbarkeit unter Verwendung von Standardhardware machen sie diesen den Markt streitig. Egal, ob man der NoSQL-Bewegung positiv oder negativ gegenübersteht, eines ist

auf jeden Fall erreicht worden: Die Diskussion um die Wahl der richtigen Datenbank mit dem richtigen Datenmodell für das vorliegende Problem ist wieder erlaubt und gewünscht.

Ein bekennender Kritiker sowohl der alten relationalen Datenbanken aber auch der NoSQL-Bewegung ist Michael Stonebraker, der uns auf unserer kleinen Reise bereits mehrfach begegnet ist. Stonebraker plädiert seit jeher für die richtige Wahl des Datenspeichers, ist jedoch auch ein Befürworter von ACID-Transaktionen [1]. Seine Hauptkritik an relationalen Datenbanksystemen ist nicht das relationale Datenmodell an sich, sondern die nicht zeitgemäße technische Implementierung der Systeme. Diese beruhen sämtlich auf Ideen aus System R und orientieren sich an in den 1970er Jahren vorherrschenden Techniken [2]. Gemäß dem Motto „Stand on the shoulders of those who came before you, not on their toes." [1] finden die Konzepte der relationalen Welt in VoltDB eine technisch aktuelle Umsetzung.

Anwendungsfelder von Datenbanken

In den ersten beiden Teilen unserer Serie haben wir im Wesentlichen zwischen SQL- und NoSQL-Systemen unterschieden und Datenbanksysteme gemäß des unterliegenden Datenmodells in fünf Kategorien eingeteilt: in relationale Systeme und die üblichen vier NoSQL-Kategorien Dokumenten-, Key-Value und Wide-Column Stores sowie Graphdatenbanken.

An dieser Stelle schauen wir auf die Unterscheidung nach Anwendungsbereichen. Die wohl gewichtigsten Anwendungsbereiche sind

- Online Transaction Processing (OLTP)
- Data Warehouses
- Textmanagement
- Stream Processing

Viele NoSQL-Datenbanken lassen sich den oben genannten Bereichen zuordnen. So sind Elasticsearch oder Solr der Textindizierung zuzuordnen. Anwendungen im Data Warehousing können mit Column Stores wie HBase oder Cassandra umgesetzt werden. Auf dem Hadoop-Stack basierende Lösungen wie Storm oder Kafka sind für Streamverarbeitung verschiedenster Art optimiert.

In diesen drei Bereichen gibt es also nicht relationale Lösungen, die relationalen Lösungen um Größenordnungen den Rang ablaufen. Der Bereich des OLTP ist die originäre und nun letzte große Domäne der relationalen Datenbanksysteme, und dieser wird ihnen im Moment von NoSQL streitig gemacht.

Wir schauen uns zwei Arten von Datenbanksystemen im Bereich OLTP genauer an: zum einen Multi-Modell-Datenbanken, die verschiedene Datenmodelle im Sinne der NoSQL-Klassifikation vereinen. Und zum anderen NewSQL-Systeme, die zumindest teilweise das relationale Datenmodell beibehalten, die Mechanismen zur Erfüllung der gewünschten Eigenschaften aber zeitgemäß implementieren.

Multi-Modell-Datenbanken

Multi-Modell-Datenbanken sind NoSQL-Datenbanken, die unterschiedliche Datenmodelle vereinen, beispielsweise Eigenschaften von dokumentenorientierten Datenbanken und Graphdatenbanken. Denn gelegentlich lassen sich die Entitäten der Anwendungsdomäne gut mit Dokumenten beschreiben, allerdings bestehen zwischen diesen ja auch Beziehungen.

Mitarbeiterdaten, Abteilungen und Projekte können gut als Dokument erfasst werden. Den Ähnlichkeiten und kleinen Unterschieden der Daten (wie mehrere Adress- oder Telefonnummernarten) kann so gut Rechnung getragen werden. Die Beziehungen der Mitarbeiter zu Abteilungen und Projekten kann über Datenredundanz modelliert werden, wenn Daten über Mitarbeiter und Projekte atomar zur Verfügung gestellt werden sollen. Wird normalisiert modelliert, so sind für das Zugreifen auf Mitarbeiter- und Abteilungsdaten mitunter mehrere Anfragen notwendig.

Multi-Modell-Datenbanken erlauben nun unterschiedliche Sichtweisen auf die gespeicherten Daten und vereinen in ihren Anfragesprachen Elemente aus verschiedenen der vier NoSQL-Kategorien. Die Beziehungen zwischen Mitarbeitern, Abteilungen und Projekten werden dann als ein Graph beschrieben, dessen Knoten die Dokumente sind. Die Datenbank unterstützt nativ

Info

Michael Stonebraker ist Professor am MIT und ein alter Hase im Datenbankgeschäft. Mit Sequel hat er eine der ersten, auf Edgar F. Codds Arbeit zum relationalen Datenmodell beruhenden, relationalen Anfragesprachen implementiert. Er ist Mitbegründer des Systems Ingres, dem Vorläufer von PostgreSQL, sowie Mitinitiator zahlreicher weiterer Datenspeicherungstechnologien. Aus den Forschungsprojekten Aurora, C-Store und H-Store, an denen er beteiligt war, sind die Datenbanksysteme StreamBase (eine Streamverarbeitungsanwendung), Vertica (eine spaltenorientierte Datenbank) und VoltDB (ein NewSQL-Datenbanksystem) hervorgegangen [3]. Von Stonebraker stammt der zu unseren vorangehenden Serienteilen passende Ausspruch: „Those who do not understand the lessons from previous generation systems are doomed to repeat their mistakes."[1].

Info

Online-Transaction-Processing-(OLTP-)Datenbanken sind im Bereich der NoSQL-Datenbanken die am häufigsten anzutreffende Art von Datenbanken. OLTP-Systeme sind darauf ausgelegt, wiederkehrende Arten von Anfragen zu beantworten, beispielsweise Anfragen nach Benutzerdaten, Artikelnummern oder auch Empfehlungen in Onlineshops.
Im Gegensatz zu Data Warehouses werden in OLTP-Systemen meist keine, gegebenenfalls langlaufenden, Ad-hoc-Anfragen abgesetzt. Anfragen in OLTP-Systemen sind meist von wiederkehrender Struktur sowie so konzipiert und optimiert, dass sie auf heutigen Maschinen im Millisekundenbereich beantwortet werden können.

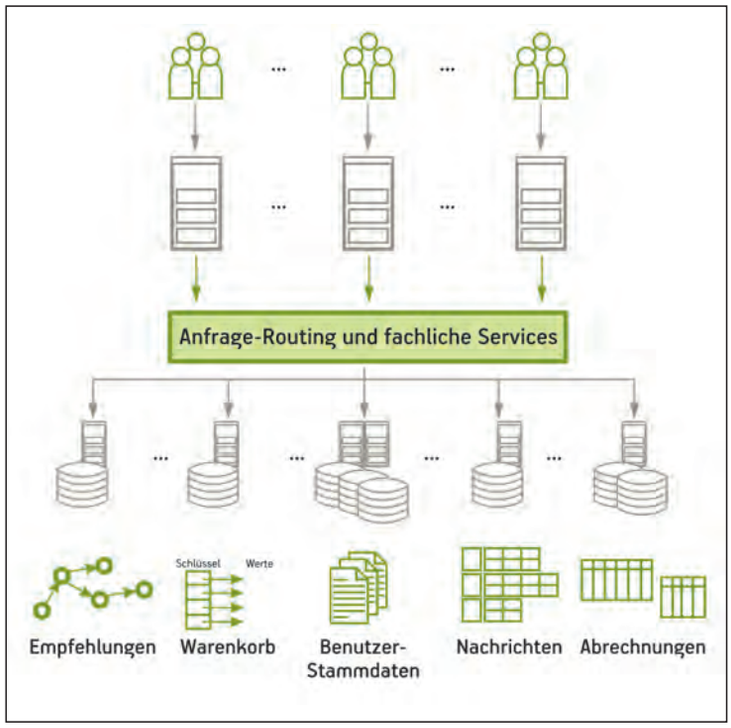

Abb. 1: Der Webshop aus [6]; mit einer Multi-Modell-Datenbank könnten verschiedene Datenmodelle mit derselben Technologie polyglott verknüpft werden

sowohl Anfragen und Auswertungen über die Dokumente als auch Fragen nach Beziehungen und Pfaden. Die Anzahl der Multi-Modell-Datenbanken im NoSQL-Archiv ist überschaubar [4], die bekanntesten Vertreter sind wohl ArangoDB, OrientDB und FoundationDB. Generell ist bei Verwendung einer Multi-Modell-Datenbank darauf zu achten, nicht versehentlich einen unpassenden monolithischen Ansatz zu wählen.

Die ArangoDB ist eine „Multi-Purpose-Datenbank", deren Basismodell Dokumente sind [5]. Sie bietet aber von Beginn an auch die Vorzüge einer Graphdatenbank und unterstützt die ACID-Eigenschaften sowie das atomare „Joinen" über mehrere Datensätze hinweg. Beziehungen zwischen Daten können als Graph modelliert werden. Da Dokumente die Basiseinheit sind, sind auch Beziehungen Dokumente und können selbst wieder als Knoten interpretiert werden, was das Abbilden komplizierter Szenarien erlaubt.

Listing 1

```
// Persons und Cities sind Dokumente
FOR p IN Persons
  // Filtern von Dokumenten
  FILTER myfunctions::mustPayTax(p)
  LET distance = LENGTH(
    // Graphalgorithmus kürzester Weg
    SHORTEST_PATH(Persons, Friends, p, 'Persons/godfather', 'any'))
FOR c IN Cities
  FILTER p.zip == c.zip
  // atomare Aktualisierung mehrerer Personendaten
  UPDATE p WITH {taxPaid: p.taxPaid * c.discount * distance}
    IN Persons
```

Die ArangoDB Query Language AQL ist die Schnittstelle zum System. Das Beispiel von der ArangoDB-Homepage [5] zeigt in Listing 1, wie in einer einzigen Anfrage Gebrauch von den Konzepten Dokument, Graph und Transaktionalität gemacht werden kann.

Das Verwenden mehrerer Datenmodelle innerhalb einer Anwendung und Anfrage ist aber nur einer der Vorteile, die Multi-Modell-Datenbanken anpreisen. Schauen wir noch einmal zurück auf unser Beispiel eines Webshops aus dem zweiten Artikel unserer Serie (**Abb. 1**). Wir haben für die verschiedenen Teile des Webshops unterschiedliche Persistierungstechnologien, zugeschnitten auf die Verwendung, identifiziert und im Sinne einer polyglotten Persistenzstruktur verschiedene Stores vorgeschlagen. Dies lässt sich nun auch mit verschiedenen, unabhängigen Instanzen einer Multi-Modell-Datenbank realisieren, wobei jede Instanz nur ein Datenmodell (also nur Dokument oder nur Graph) nutzt. Der Vorteil: Es handelt sich um nur eine Technologie; Expertenwissen und Support lassen sich also bei einem Anbieter anfragen.

Einen ähnlichen Weg geht die FoundationDB, ein Key-Value Store, der ebenfalls ACID-Transaktionen als Grundbausteine solider Applikationen verwendet. Die FoundationDB bietet Schichten, die sie nach außen hin anderen Kategorien zugehörig erscheinen lassen. Neben einer Graphschicht gibt es auch eine SQL-Schicht, was die FoundationDB schon in die Nähe der NewSQL-Datenbanken rückt.

NewSQL-Datenbanken

NewSQL-Systeme versuchen die Vorzüge von SQL und ACID beizubehalten. Ein Kritikpunkt der NewSQL-Anbieter lautet: In den letzten Jahren sind die sich auf dem Markt befindlichen relationalen Systeme zwar angepasst worden und bieten mit Sharding, Kompression, Bitmap-Indexen, benutzerdefinierten Typen und vielem mehr neue, zeitgemäße Eigenschaften. Kein System ist allerdings von Grund auf neu geschrieben worden [2].

Das Erweitern vorhandener Systeme führt allerdings dazu, dass im Kern Altcode immer noch aktiv ist, der grundlegende Mechanismen verwendet, die für heutige Hardwarearchitekturen, insbesondere die gängige Shared-Nothing-Architektur von Rechnerclustern auf Standardhardware, nicht mehr aktuell ist. Die technologischen Grundideen der meisten großen, auch heute noch im Markt aktiven, relationalen Systeme sind bereits in IBMs System R aus den späten 1970er Jahren vorhanden [2].

Durch die Verbreitung des Webs hat die Maschine-zu-Maschine-Kommunikation zugenommen. Längst ist die Mehrzahl der Nutzer einer Datenbank nicht mehr ausschließlich menschlich. Dies führt zu anderen Bedürfnissen und Anforderungen, als sie eine Konsole für menschliche Endbenutzer fordert.

Wir bewegen uns heutzutage auch in ganz anderen Dimensionen für Haupt- und Persistenzspeicher, de-

ren Preise stark gefallen sind. Der Trend geht weg von vertikal hochskalierten und hochspezialisierten Einmaschinensystemen und hin zu horizontal verteilten Systemen, die auf Standardhardware betrieben werden können.

Stonebraker identifiziert für die gängigen relationalen Systeme vier, auf das Design von System R und die Umstände der Zeit seiner Entwicklung zurückgehende Eigenschaften, die massiven Overhead erzeugen [9], [10]:

- Führen von Logdateien (Logging)
- Sperren von Datensätzen (Locking)
- Sperren von Indexen (Latching)
- Puffermanagement (Buffer Management)

Die traditionellen Datenbanksysteme sind gezwungen, jeden Datensatz wenigstens zweimal zu schreiben: Einmal in die Datenbank selbst und einmal in eine Logdatei für Backup und Recovery. Die Logdatei muss darüber hinaus auf der Festplatte gespeichert werden, um die Dauerhaftigkeit (das D in ACID) zu gewährleisten.

Bevor eine Transaktion einen Datensatz ändern darf, muss eine Sperre angefordert werden, um exklusiven Zugriff auf den Datensatz zu erhalten. Die Koordination der Sperren geschieht über eine Relation in der Datenbank, deren Verwaltung kostspielig ist.

Info

Eine neue Klasse von Datenbanken sind die NewSQL-Systeme, die sich nicht so recht dem NoSQL-Spektrum zuordnen lassen. Eine sehr lesenswerte Übersicht über NewSQL-Systeme und deren Eigenschaften im Vergleich zu NoSQL bietet der Artikel [7].
NewSQL-Systeme versuchen die Vorzüge der relationalen Systeme mit modernen und zeitgemäßen Mitteln umzusetzen und somit die Vorteile als auch das Wissen um den Umgang mit relationalen Datenbanken und ACID-Transaktionalität zu erhalten und zu nutzen. Die NoSQL-Welt präferiert BASE und Eventual Consistency, wann immer es möglich ist, um maximale Performanz zu gewährleisten (vgl. Teil 2 dieser Serie). Häufig sind Transaktionen, die mehrere Datensätze umspannen, aber gewünscht. Google setzt nach BigTable nun auf die verteilte Datenbank Spanner und führt folgende Begründung an: „We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.“ [8].

Der Zugriff auf von mehreren Transaktionen gemeinsam genutzte Strukturen wie B-Bäume oder andere Indexe muss ebenfalls koordiniert werden. Diese

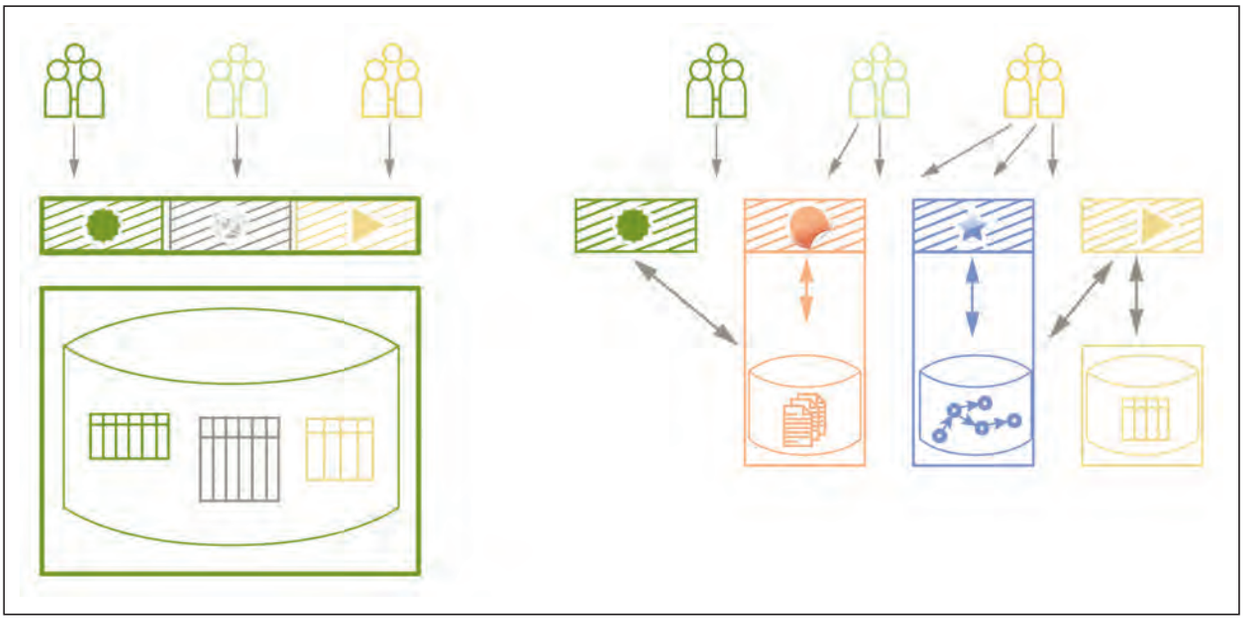

Abb. 2: Nebeneinanderstellung eines monolithischen Systemansatzes und einer polyglotten Systemarchitektur

Sperren sind in der Regel sehr kurzlebig, ihr Verwaltungsaufwand allerdings ist nicht zu vernachlässigen.

Ausgefeilte Pufferstrategien sorgen dafür, dass möglichst viele der benötigten und auf der Festplatte gespeicherten Daten schnell zugreifbar im Cache der Datenbank vorliegen. Der Austausch von Daten zwischen Puffer und Festplatte erfolgt immer seitenweise. Ein Puffermanager muss zum einen dafür sorgen, dass die benötigten Daten im Hauptspeicher residieren und zum anderen den Datensatz innerhalb der Pufferseite identifizieren.

Den Verwaltungsoverhead, den jeder dieser vier Punkte veranschlagt, liegt bei ca. 25 Prozent [10]. Das heißt, wenn alle vier Punkte eliminiert werden können, kann eine Verdoppelung der Performanz des Systems erreicht werden. Aus diesen Überlegungen ist die NewSQL-Datenbank H-Store entstanden, die mit der VoltDB ihre kommerzielle Variante gefunden hat, siehe hierzu auch den schönen Artikel von Jan Stamer [11].

Info

Hardwarearchitekturen haben sich stark gewandelt. In den 1970er gab es vorwiegend Shared-Memory-Multi-Prozessorarchitekturen, erweitert um Shared-Disk-Support in den 1980er Jahren. Diese sind darauf ausgelegt, vertikal zu skalieren und Multi-Threading auf einer einzigen Maschine optimal zu unterstützen.

Die heute, auch in der NoSQL-Welt vorherrschende Architektur ist eine Shared-Nothing-Architektur. Hierbei werden viele unabhängige Rechnerknoten über ein Netzwerk vernetzt, die Rechner selbst teilen aber nichts, weder Speicher noch CPU-Zeit. Der Ansatz lässt sich noch weiter treiben, sodass auch die Speicherbereiche der einzelnen CPUs eines Rechners nichts teilen. Der Austausch von Daten erfolgt ausschließlich über das Netzwerk oder den Systembus eines Rechners.

Die Grundüberlegungen zur Überwindung der oben diskutierten Defizite für OLTP-Systeme können folgendermaßen zusammengefasst werden: Eine verteilte Architektur und Redundanz der Daten machen das Persistieren von Undo-Logs unnötig. Diese können nach dem erfolgreichen Commit oder Rollback einer Transaktion verworfen werden. Redo-Logs werden gar nicht mehr benötigt, die Wiederherstellung nach Datenverlust auf einem Rechnerknoten geschieht über das Netzwerk aus redundanten Daten im Cluster.

Langlaufende Anfragen sind selten und durch gutes Programmdesign oft vermeidbar. Transaktionslaufzeiten im Millisekundenbereich sind durchaus normal. Diese kurzen Transaktionen können in einer Single-Threading-Umgebung linearisiert ausgeführt werden. Durch geschicktes Sharding können Anfragen auch echt parallel auf verschiedenen Knoten des Clusters ausgeführt werden. Das macht knotenübergreifende Sperrverfahren unnötig.

Wenn alle Daten im Hauptspeicher residieren, kann auf Datensätze direkt, ohne Indirektion über den Puffermanager, zugegriffen werden. Das macht diesen obsolet, der Verwaltungsaufwand und eine Indirektionsschicht fallen weg.

Echte langlaufende Anfragen, wie sie für Reporting oder Ad-hoc-Statistiken nötig sind, sollten ohnehin in ein für diese Zwecke bereitgestelltes und optimiertes Data Warehouse verlagert werden, dem gegebenenfalls ein anderes Datenmodell zugrunde liegt. Hauptspeicher ist heutzutage so günstig, dass viele Daten In-Memory gehalten werden können. Die Verteilung der Daten in einem Cluster verändert die Backup- und Recovery-Strategien. Sharding der Daten nach Zugriff ist eine Strategie, um aufwändige Sperrverfahren wie Zwei-Phasen-Com-

mit-Protokolle zu vermeiden. Diese kosten Netzlatenzen von mehreren Millisekunden und sind damit oft länger als die Ausführungsdauer einer Transaktion selbst.

Polyglotte Persistenz: Fluch oder Segen?

NoSQL (die Multi-Modell-Datenbanken eingeschlossen) und NewSQL bieten die Möglichkeit, flexible, polyglotte Systeme zu bauen. Hierbei muss allerdings wiederholt betont werden, dass man die Komplexität durch einen polyglotten Ansatz nicht einfach „wegzaubern" kann. Die Komplexität, die man durch Abbildung verschiedener Domainteile in einer relationalen Struktur erhält, lässt sich in der Modellierung vielleicht auflösen, beispielsweise durch die Aufteilung in einen Graphen und eine relationale Struktur, die die Domäne natürlicher abbilden. Diese Vereinfachung verschiebt allerdings Aufwand in den Betrieb, die Wartung, die Auswertung sowie das Monitoring des gesamten Systems. Ein Blick auf die Konsequenzen aus dem CAP-Theorem lohnt sich hier (vgl. auch den vorangehenden Teil dieser Serie).

```
@NodeEntity
public class Actor {
        @RelatedToVia
        Collection<Role> roles;

        public Actor() {
        }

        public Iterable<Role> getRoles() {
            return roles;
        }
}
public interface MovieRepository extends GraphRepository<Movie>,
RelationshipOperationsRepository<Movie> {

        Movie findById(String id);

 Page<Movie> findByTitleLike(String title, Pageable page);

 @Query( "(user=node({0}) match user-[r:RATED]->movie<-[r2:RATED]"
      + "-other-[r3:RATED]->otherMovie where r.stars >= 3 and"
      + " r2.stars >= r.stars and r3.stars >= r.starsreturnotherMovie,"
      + " avg(r3.stars) as rating, count(*) as cnt" )
 List<MovieRecommendation> getRecommendations( User user );

}
```

```
@Entity
@NamedQuery(name = "User.findByTheUsersName", query = "from User u
where u.username = ?1")
public class User extends AbstractPersistable<Long> {
        @Column(unique = true)
        public String username;

        public String firstname;
        public String lastname;

        public User() {
        this(null);
 }
 /**
 * Creates a new user instance.
 */
  public User(Long id) {
   this.setId(id);
  }
}
public interface SimpleUserRepository extends CrudRepository<User, Long> {
 /**
 * Find the user with the given username.
 */
 User findByTheUsersName(String username);
 /**
 * Find all users with the given lastname.
 */
 List<User> findByLastname(String lastname);
 /**
 * Returns all users with the given firstname.
 */

 @Query("select u from User u where u.firstname = ?")
 List<User> findByFirstname(String firstname);
```

Abb. 3: Nebeneinanderstellung der Entwicklungsmodelle in Spring Data JPA und Spring Data Neo4j

Vergleichen wir einmal die Entwicklung gegen ein verteiltes polyglottes System mit der Entwicklung einer herkömmlichen Datenhaltung in einem einzigen relationalen System (**Abb. 2**). Wir sehen, dass wir das Wissen über das System breiter streuen müssen. Dies beginnt bei der Konfiguration der Datasources und setzt sich über die genutzten Annotationen innerhalb des Sourcecodes fort bis in die Funktionsweise des Systems selbst. Abläufe und Abhängigkeiten innerhalb des Systems müssen sehr gut verstanden sein, um ein verteiltes polyglottes System zum einen wartbar und am Leben erhalten zu können, zum anderen aber auch die Zugänglichkeit für weitere Entwicklungen zu gewährleisten. Ebenso wächst der Kommunikationsaufwand, wenn Schnittstellen verändert werden sollen, welche nun häufig in mehreren Versionen genutzt werden können. Monitoring gewinnt in solchen Systemen eine noch größere Bedeutung, da auch die Nutzung von Schnittstellen und Services durch andere Teile des Systems protokolliert werden muss, um gegebenenfalls über deren Abschaltbarkeit entscheiden zu können.

Am Beispiel von Spring Data kann man schön sehen, wie versucht wird, zum Umgang mit relationalen Systemen äquivalente Entwicklungsmodelle bereitzustellen, um die Komplexität von polyglotten Systemen zumindest innerhalb der Entwicklung einigermaßen in den Griff zu bekommen. Natürlich könnte man auch ohne Spring Data entwickeln und mit dem nativen API von Neo4j und klassischem JPA mit Hibernate als Provider arbeiten, jedoch ist dann auch die Entwicklungskomplexität ungleich höher. **Abbildung 3** zeigt die Ähnlichkeiten zwischen Spring Data für JPA und Neo4j.

Wie man im Vergleich in **Abbildung 3** schön sehen kann, ist das Entwicklungsmodell sehr ähnlich, ob wir nun gegen eine relationale DB mit JPA oder gegen eine Graphdatenbank wie Neo4j [12] mit der Erweiterung Spring Data Neo4j [13] arbeiten. Auch die Möglichkeiten, speziellere Anfragen innerhalb der Repository-Implementierungen zu nutzen, sind in beiden Fällen sehr ähnlich. Auf der einen Seite wird klassisches SQL, auf der anderen Seite Cypher [14] verwendet.

Durch Hinzufügen von Neo4j in die Systemlandschaft müssen wir jedoch Datenbank-Connections, -Firewalls und -Management für mindestens ein weiteres Datenbanksystem, das sogar aus mehreren Clusterknoten bestehen kann, pflegen, dokumentieren und überwachen.

Wenn man hier nun Systeme der Multi-Modell-Datenbank und der NewSQL-Welt, wie beispielsweise die ArangoDB oder auch VoltDB, mit zum Vergleich heranzieht, sehen wir, dass hier weiteres Spezialwissen zu erarbeiten ist. Die Datenbanktreiber der ArangoDB [15] sind nicht gegen einen Standard wie beispielsweise JPA implementiert, das entstandene API ist proprietär. Das ist nicht notwendigerweise schlecht. Jedoch zeigt uns die Erfahrung der Vergangenheit, dass hier Risiken entstehen können.

Artenvielfalt

Der Datenbankmarkt ist in Bewegung und wird immer vielfältiger. Nach dem Rückgang der relationalen Systeme und den Verwirrungen der NoSQL-Welt komplettieren nun die NewSQL-Systeme die Unübersichtlichkeit bei der Wahl des richtigen Speichersystems. Das richtige System zu wählen, ist jedenfalls weniger denn je eine triviale Aufgabe, und die Gefahr, dass man das für sein Problem optimale System gar nicht kennt, ist bei der gegebenen Vielzahl an erhältlichen Datenbanksystemen natürlich groß.

Polyglotte oder eher monolithische Ansätze lassen sich mit allen Systemen, seien sie OldSQL, NoSQL oder NewSQL, erreichen. Vertreter wie die Multi-Modell-NoSQL-Systeme sind hier sorgsam zu erlernen, da sie einen vielleicht eher dazu verführen, sein System versehentlich monolithischer und enger zu gestalten als beabsichtigt, da unterschiedliche Modelle nebeneinander vereint existieren. Dies allerdings ist ja auch ihr großer Vorteil.

Abzuwarten ist, ob der Druck der Märkte nicht wiederum dazu führt, dass Systeme zu Universallösungen werden wollen. Dies ist zum Scheitern verurteilt und hat das langsame Aussterben der OldSQL-Systeme zumindest eingeleitet. Diese Art der Entwicklung sollte vermieden werden, um die Artenvielfalt zu erhalten. Doch auch Spezialisten müssen sich erst etablieren. Viele neue Systeme werden noch eine Weile brauchen, um sich durchzusetzen. Denn in puncto Tuning und Betriebsstabilisierung sowie generell zur Risikominimierung bei der Wahl eines Systems stehen momentan Stores wie MariaDB oder Neo4j noch deutlich besser da als beispielsweise die VoltDB, für die zur Zeit nur schwer Experten zu finden sein dürften.

Mit den neuen Stores müssen sich auch die Programmiermodelle zur Unterstützung der Entwickler ausbilden und wachsen. Dabei kann aus den Erfahrungen der bestehenden Werkzeuge gelernt werden, Spring Data macht es mit der Unterstützung für viele NoSQL-Datenbanken vor.

Fazit

Aus unserer Sicht bleibt es dabei: Nie zuvor war das Angebot an Datenbanken so groß wie heute, und nie war die Wahl so spannend. Alles in allem sollte aber klar sein, dass es die Geschäftsdomäne ist, die letztendlich bestimmt, wie ein System architektonisch zu designen ist und welche Speicherstrategien dazu optimal geeignet sind. Disziplin und Wissen sind hier – wie so oft – der Schlüssel zum Erfolg.

Wer seine Geschäftsdomäne kennt, zusammen mit den Anforderungen und Beschränkungen, die sie stellt, kann maximale Performanz erzielen. Und im Falle weicher Anforderungen, oder wenn die Domäne sich stark ändern darf, sind schemafreie Systeme vorhanden, die Migrationsrisiken klein halten. Welche der Systeme sich letztlich wirklich durchsetzen werden, bleibt nur abzuwarten. Die Wahrscheinlichkeit aber, dass ACID und strengere Programmierparadigmen ihren festen Platz in der Entwicklung behalten, ist anzunehmen. Deutlich sind die Zeichen, die NewSQL-Systeme setzen. Und der Wunsch nach Transaktionen über mehrere Datensätze hinweg steht auch bei einigen NoSQL-Systemen auf der Roadmap, zum Beispiel bei MongoDB [16].

Wir hoffen, dass die Anbieter der Systeme der neuen Welt nicht den Blick zurück scheuen und den Weg der Spezialisierung und Stabilisierung der Systeme weitergehen. So kann spezielles Wissen aus der Domäne optimal unterstützt werden, und Performanz geht nicht verloren, weil das System als Universallösung alles will und nur wenig noch richtig gut kann. Aus der Erfahrung der Vergangenheit kann gelernt werden, denn früher war ja nicht alles schlecht.

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 Code Clinic tätig. Darüber hinaus twittert und bloggt er gerne.

@arrasz **http://blog.synyx.de/**

Links & Literatur

[1] Stonebraker, M.: „Stonebraker on NoSQL and Enterprises“, Communications of the ACM, 54(8), 2011

[2] Stonebraker, M. et al.: „The End of an Architectural Era. (It’s Time for a Complete Rewrite)“, Proc. 2007 VLDB Conference, 2007

[3] http://en.wikipedia.org/wiki/Michael_Stonebraker

[4] nosql-database.org/

[5] https://www.arangodb.com/

[6] Mennerich, C.; Arrasz, J.: „Ein Reise durch NoSQL – Teil 2“, in Java Magazin 6.2015

[7] Grolinger, K. et al.: „Data management in cloud environments: NoSQL and NewSQL data stores“, Journal of Cloud Computing 2:22, 2013

[8] Corbett, J. C. et al.: „Spanner: Google’s Globally-Distributed Database, Proceedings of OSDI’12“, Tenth Symposium on Operating System Design and Implementation, 2012

[9] Harizopoulos, S. et al.: „OLTP Through the Looking Glass, and What We Found There“, SIGMOD’08, 2008

[10] Stonebraker, M.: „SQL Databases v. NoSQL Databases“, Communications of the ACM, 53(4), 2010

[11] Stamer, J.: „Elefant unter Strom“, in Java Magazin 1.2015

[12] http://neo4j.com

[13] https://github.com/spring-projects/spring-data-neo4j/

[14] http://neo4j.com/docs/stable/cypher-query-lang.html

[15] https://github.com/arangodb/arangodb-java-driver

[16] https://www.careerdean.com/q/mongo-supprt-multi-documentcollection-acid-compliant-transactions