Vom Hype zum rationalen Einsatz

Was Architekten bei NoSQL-Datenbanken beachten sollten

Halil-Cem Gürsoy

NoSQL-Datenbanken werden in immer mehr Projekten eingesetzt. Architekten möchten mit diesen neuen Datenbanken nicht nur Skalierbarkeitsprobleme lösen, sondern auch wesentlich besser mit komplexen Datenstrukturen umgehen. Aber der Begriff „NoSQL" umfasst ein sehr breites Spektrum unterschiedlichster Datenbanktypen und sie haben die Wahl zwischen extrem schnellen und skalierbaren Key/Value-Datenbanken, die aber nur sehr einfache Datenstrukturen unterstützten, bis hin zu Graphendatenbanken für komplexe Daten und Systemen, die verteilte Transaktionen über mehrere Datensätze hinweg gar nicht unterstützen, und solche, die dies sowohl mit JTA als auch mit ACID tun. Was also ist aus Architektensicht wichtig, wenn der Einsatz einer NoSQL-Datenbank in Betracht gezogen wird? Welche Datenbank ist für welche Aufgabe geeignet? Und wann ist der Einsatz einer NoSQL-Datenbank ratsam?

Von SQL zu NoSQL

Bisher war die Welt für Architekten und Entwickler meist recht simpel gestaltet. Auf der einen Seite der Quelltext und daraus resultierend eine Applikation, auf der anderen Seite die Datenhaltung, in der Regel ein relationales Datenbankprodukt eines namhaften Herstellers. Die Abfragesprache ist mit SQL standardisiert, sodass die Einarbeitung in die Syntax einer anderen relationalen Datenbank in den meisten Fällen recht einfach ist.

Mittlerweile sind objektrelationale Mapper ausgereift. In der idealen Projektwelt kennt ein Java-Entwickler nur noch die „Java Persistence API", um den Rest kümmern sich Implementierungen wie Hibernate; er kommt nur noch in wenigen Fällen mit SQL in Berührung.

In die verwendete Datenbank muss alles hinein, was persistiert werden muss. Egal, ob es um die Netzstruktur eines Telekommunikationsanbieters oder um Katalogdaten eines Versandhändlers geht. Oft sind die Datenbankprodukte zudem vorgegeben. „One size fits all" – eine Lösung für alle infrage kommenden Anforderungen. Und ist der Server irgendwann einmal zu stark ausgelastet, wird vertikal skaliert: mehr CPU, mehr Hauptspeicher usw. Dieser Ansatz muss in Zeiten immer größer werdender Datenmengen einmal an Grenzen stoßen.

Nun aber hat der „Hype Cycle" [GartnerHC] um NoSQL gerade seinen Höhepunkt erreicht [GartnerHCDM] und NoSQL-Datenbanken finden sich in immer mehr Projekten wieder. Häufig weniger bedingt durch Abwägung der Anforderungen als dadurch, dass der Einsatz nunmehr „erlaubt" ist. Aber NoSQL ist nicht gleich NoSQL, sondern es gibt eine große Anzahl unterschiedlicher Ansätze und Datenbanken, laut aktuellen Zahlen über 122 [NoSQLorg].

Klassifizierung von NoSQL-Datenbanken

Im Groben lassen sich die NoSQL-Datenbanken in vier Gruppen (für weitere Klassifizierungen im Detail [Strau]) unterteilen:

- Key/Value-Datenbanken,
- Column Family-Datenbanken,
- Dokumentenorientierte Datenbanken und
- Graphenorientierte Datenbanken.

Bei *Key/Value-Datenbanken* werden Schlüssel/Wert-Paare in einer Datenbank abgelegt. Es existiert kein Schema, wie aus der relationalen Welt bekannt, und es ist egal, was in den Werten gespeichert wird. Datenbanken dieser Gattung sind häufig auf sehr schnelle, viele konkurrierende Lese- und Schreibzugriffe und auf einen hohen Grad an Verteilung in einem Cluster optimiert. Der klassische Anwendungsfall für eine solche Datenbank ist beispielsweise das verteilte Cachen von Daten oder das Speichern von Nachrichten. Tatsächlich dürfte es sehr schwierig sein, klassische Geschäftsobjekte auf einer solchen Datenbank abzubilden. Bekannte Vertreter dieser Gattung sind Redis, Riak und die Amazon SimpleDB.

Die *Column Family-Datenbanken* kommen einem relationalen Datenbanksystem bezüglich der Strukturierung der Daten etwas näher, wobei auch hier Schlüssel/Wert-Paare abgelegt werden, die gemeinsam eine Spalte bilden. Dabei können in den Zeilen einer Tabelle jeweils unterschiedliche Spalten enthalten sein. Diese Datenbanken sind meistens darauf optimiert, mit sehr großen Datenmengen in einem verteilten System zu arbeiten. Ein klassisches Szenario ist das Ablegen von Messdaten: Die ID des Messgerätes als Schlüssel, ein Messdatum sowie das Messergebnis, womöglich noch angereichert um weitere (Meta-)Daten. Der Einsatz solch einer Datenbank, deren Setup unter Umständen recht komplex werden kann, wird aber erst dann richtig sinnvoll, wenn wir tatsächlich von „vielen" Daten sprechen. Einige wenige Millionen Datensätze sind da eher „Peanuts", der Architekt sollte dann zu einer relationalen Datenbank greifen, wenn auch die weiteren Rahmenbedingungen passen.

Möchte nun ein Client z. B. wissen, welche Ergebnisse von einem bestimmten Gerät erfasst wurden, so kann dies mit Hilfe von MapReduce [DeaGh04] optimiert werden. MapReduce

kann vereinfacht als ein „verteiltes Suchen und Aggregieren“ bezeichnet werden. Die Suche auf allen Knoten eines Clusters ist dabei nicht effizient, was den gesamten Ressourcenbedarf betrifft, bietet aber dafür einen hohen Durchsatz. Ein weiteres Beispiel für den Einsatz solch einer Datenbank, zu deren bekanntesten Vertretern Apache Cassandra gehört, wäre Twitter zur Speicherung der Kurznachrichten.

Auch wenn diese Datenbanken über eine erweiterte Möglichkeit der Datenmodellierung gegenüber den Key/Value-Datenbanken verfügen, so ist es mit diesen nicht möglich, analog einem relationalen Datenmodell, Relationen zwischen einzelnen Datensätzen innerhalb der Datenbank herzustellen, es werden meist nur Links unterstützt. Dies führt dazu, dass Joins nicht möglich sind. Google ermöglicht zwar beispielsweise innerhalb der Google App Engine eine Implementierung gegen BigTable mit Hilfe von JPA oder JDO, aber nur unter sehr großen Einschränkungen.

Der „Urvater“ dieser Datenbanken, Google BigTable bzw. dessen Grundideen [Chang06], wurde mehrfach kopiert, so basieren z. B. Apache HBase und Apache Cassandra auf dessen Prinzipien.

Dokumentenorientierte Datenbanken sind darauf spezialisiert, „Dokumente“ zu speichern. Bei Dokumenten kann es sich um die oben erwähnten Messergebnisse eines Messsystems, andere semistrukturierte Daten (s. Listing 1) oder tatsächlich um „richtige“ Dokumente handeln.

```
{"name":"Meier",
  "forename":"Max",
  "adress":{"street":"Deich 7",
            "postcode":28355,
            "city":"Bremen"}
  "comment":"Ok."}
```

Listing 1: Ein Dokument in einer JSON-Repräsentation aus MongoDB

In den meisten Produkten dieser Gattung werden die Daten in JSON bzw. BSON, „Binary JSON“, einem binär serialisierten JSON-Format, gespeichert. Indizes können über verschiedene Attribute eines Dokuments gelegt werden. Auch wenn hier von Dokumenten gesprochen wird, eine Funktionalität wird zumindest bei den bekanntesten Vertretern, MongoDB und CouchDB, vermisst: die Volltextsuche. Hier gibt es verschiedene Ansätze, von einem Tokenizen des Inhaltes innerhalb der Datenbank bis hin zu der Verwendung eines dedizierten Indizierungssystems wie Apache Lucene. Außerdem sind diese Datenbanken, wenn auch gut vorbereitet auf eine horizontale und vertikale Skalierung, nicht unbedingt auf „BigData“-Szenarien ausgelegt. Wobei es für den Begriff „BigData“ keine eindeutige Definition gibt: Was als *Big* gilt, ist abhängig von den Rahmenbedingungen, unter denen mit diesen Daten gearbeitet werden muss.

Eine große Herausforderung bei der Entwicklung mit relationalen Datenbanken war schon immer die Abbildung von komplexen Graphen. Mit der NoSQL-Bewegung haben nun Datenbanken, die auf die Abbildung von Graphen spezialisiert sind, an Sichtbarkeit gewonnen. *Graphenorientierte Datenbanken* speichern Knoten und deren Beziehungen untereinander ab. Zudem sind sie häufig in der Lage, wie die Datenbank Neo4j, ebenfalls zusätzliche Attribute zu den Knoten und Beziehungen abzuspeichern.

Typische Anwendungsfälle sind die Abbildung von Netzwerken, sei es das „Social Network“ und die Beziehungen von Personen untereinander, oder auch die Abbildung eines „Netzes“, wie es z. B. in einer Telekommunikationsinfrastruktur vorzufinden ist. Diese Datenbanken stellen dabei Hilfsmittel zur Verfügung, um zum Beispiel den kürzesten Weg von Knoten A zu Knoten B zu finden. Eine Fragestellung, die in anderen Datenbanktypen nicht trivial zu lösen ist.

Spezialisiert, skalierbar und flexibel

Wie die Klassifikation der NoSQL-Datenbanken auf die genannten vier Gruppen zeigt, sind diese recht stark spezialisiert. Sie erheben nicht den Anspruch, alle Anforderungen abzudecken, die im Zusammenhang mit der Speicherung von Daten zusammenhängen. Vielmehr muss nun eine Auswahl getroffen werden, welche Datenbank tatsächlich geeignet ist. Und es muss entschieden werden, ob der Einsatz einer NoSQL-Datenbank überhaupt sinnvoll ist.

Als eine Gemeinsamkeit lässt sich feststellen, dass diese Datenbanken ohne fest vorgegebene Schemata arbeiten können. Dies ist insbesondere dann interessant, wenn im Lebenszyklus der implementierten Applikation mit häufigen Änderungen der Datenstrukturen gerechnet werden muss, was insbesondere bei „Agile Application Lifecycle Management“ (AALM, s. [Hütt11]) zu erwarten ist. Dies bedeutet aufseiten der Datenbank eine sehr große Flexibilität. Allerdings muss der Architekt im Voraus Vorkehrungen treffen, auch auf der Anwendungsseite auf mögliche Änderungen flexibel und minimalinvasiv reagieren zu können.

Hohe Skalierbarkeit spielt bei den meisten NoSQL-Datenbanken ebenfalls eine große Rolle. Klassische relationale Datenbanksysteme sind eher darauf optimiert, vertikal zu skalieren. Sind die technischen Grenzen eines Systems wie maximaler Hauptspeicher erreicht, müssen aufwendige Maßnahmen zu einer horizontalen Skalierung eingeleitet werden.

Ein wesentlicher Aspekt der Schwierigkeiten bei horizontaler Skalierung ist, dass relationale Datenbanken nach dem ACID-Prinzip transaktional arbeiten*. Dies bedeutet unter anderem, dass zu jeder Zeit die Konsistenz der Daten im Gesamtsystem gewährleistet ist. Eric Brewer hat zu Problemen, die mit einer horizontalen Skalierung einhergehen, im Rahmen seiner PODC-Keynote das CAP-Theorem dargelegt.

NoSQL im CAP-Dreieck

Das CAP-Theorem ([Brew00], s. Abb. 1) besagt, dass ein verteiltes System immer nur höchstens zwei der drei folgenden Bedingungen vollständig erfüllen kann: starke Konsistenz (C), Verfügbarkeit (Availability, A) und Netzwerk-Partitionstoleranz (P).

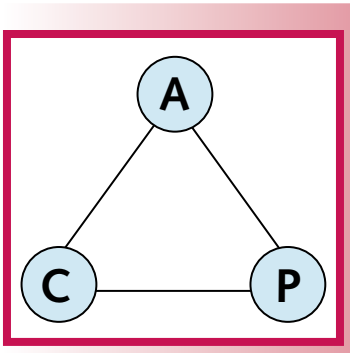

Abb. 1: Das CAP-Dreieck definiert die Freiheitsgrade bezüglich C (starke Konsistenz), A (Availability) und P (Partitionstoleranz) – maximal zwei davon kann ein System vollständig erreichen

Was bedeutet dies für Datenbanken? Ein relationales Datenbanksystem verzichtet auf die Partitionstoleranz und bildet damit ein typisches *CA*-System: Durch (verteilte) Transaktionen wird eine

* Das Akronym steht für die Eigenschaften Atomizität, Konsistenz (Consistency), Isolation und Dauerhaftigkeit einer Transaktion [HäReu83].

starke Konsistenz erreicht, die Systeme stehen, z. B. durch verschiedene Hochverfügbarkeits-Methoden, immer zur Verfügung. Erfolgt aber eine Netzwerkpartitionierung zwischen den an einer Transaktion beteiligten Systemen (oder innerhalb eines Master/Slave-Clusters), so kann das Gesamtsystem die Konsistenz der Daten nicht mehr gewährleisten, die Transaktionen würden demzufolge nicht mehr zum Erfolg führen und abgebrochen werden.

Ein Beispiel für eine NoSQL-Datenbank mit hoher Partitionstoleranz und Verfügbarkeit ist CouchDB. *AP*-Datenbanken verzichten bewusst auf eine starke Konsistenz, um hierdurch eine hohe Skalierbarkeit und Performance zu erreichen. Denn die Gewährleistung einer starken Konsistenz über alle Knoten hinweg kostet Zeit. Viel einfacher ist es dann, wenn die Replizierung der Daten auf die anderen Knoten entkoppelt von dem eigentlichen Schreibvorgang erfolgen kann.

In der Kategorie *CP* befindet sich beispielsweise HBase. In diesem Fall wird durch verschiedene Maßnahmen erreicht, dass ein Client immer auf konsistente Daten zugreift, auch wenn eine Partitionierung stattfindet. Dies erfolgt dadurch, dass gegebenenfalls das System keine Anfrage mehr beantwortet, bis die Partitionierung behoben und eine Konsistenz der Daten wiederhergestellt wurde – auf Kosten der Verfügbarkeit.

Weitere Beispiele für die Anwendung des CAP-Theorems finden sich unter [Wolf12].

BASE: schlussendlich konsistent

Brewers Überlegungen, dass man für *A* und *P* das *C* opfern muss, führten im Zusammenhang mit ACID-Transaktionen zu dem BASE-Prinzip [Prit08] als dessen logisches Gegenstück: *Basically Available, Soft-state, Eventually consistent*. Und hier haben wir schon eine der größeren Stolperfallen im Bezug auf das Verständnis von NoSQL-Datenbanken. Tatsächlich bedeutet „eventually", dass das Gesamtsystem schlussendlich nach einer definierten Zeit konsistent sein wird, nicht bloß „eventuell". Der Verzicht auf starke Konsistenz darf also nicht verwechselt werden mit dem vollständigen Verzicht auf Konsistenz.

Konsistenz

Wichtig in diesem Zusammenhang ist auch, dass im Sinne von CAP und BASE Konsistenz gleichzusetzen ist mit gleichen Daten auf allen Knoten, während in ACID darunter ein widerspruchsfreier Datenbestand zu verstehen ist.

Die NoSQL-Datenbanken gehen unterschiedlich mit dem Thema Konsistenz um: MongoDB beispielsweise erlaubt nur das Lesen aus dem primären Knoten, in den auch immer geschrieben wird. Erst durch einen expliziten Befehl können Daten, die eventuell noch inkonsistent sind, aus den anderen Knoten eines Replika-Sets ausgelesen werden. So ist in der Regel für zugreifende Clients gewährleistet, dass sie einen konsistenten Datenbestand sehen, wobei beispielsweise bei MongoDB beachtet werden muss, dass lesende Zugriffe mit einer READ_UNCOMMITTED-Semantik erfolgen.

Im Hintergrund werden die Daten auf die Replikationsknoten verteilt, so ist das Gesamtsystem nach einer bestimmten Zeit schlussendlich konsistent. MongoDB beispielsweise betrachtet einen schreibenden Zugriff erst als clusterweit committed, wenn die Änderung auf die Mehrzahl der Knoten repliziert wurde.

Kompliziert wird das Thema, wenn horizontal skaliert wird. Dies erreichen die Datenbanken durch das sogenannte Sharding, also das Aufteilen der Daten und deren Verteilen auf mehrere Server. Hierbei werden je nach Datenbank unterschiedliche Verfahren eingesetzt. Nehmen wir wieder MongoDB als Beispiel, werden dort die Daten anhand eines sogenannten Shard-Keys verteilt. Dabei ist es wichtig, dass der Architekt sehr große Sorgfalt bei dessen Wahl walten lässt, denn die Daten sollten, wenn möglich, gleichmäßig auf die einzelnen Shard-Server verteilt werden. Erfolgt eine ungleichmäßige Verteilung, kann dies zu negativen Nebeneffekten führen, da hierdurch die Knoten unterschiedlich belastet werden [Folk10].

Apache Cassandra bietet übrigens eine sehr große Flexibilität bezüglich der Konfigurationsmöglichkeiten. Durch das Anpassen der Anzahl der Replikationsknoten, aus denen gelesen werden darf, sowie der Knoten, auf die für eine Transaktion erfolgreich geschrieben werden muss, kann sich die Datenbank frei im CAP-Dreieck bewegen. Ist die Anzahl sowohl der lesbaren als auch zu beschreibenden Knoten gleich und entspricht diese der Gesamtmenge der Knoten, so liegt ein System mit starker Konsistenz vor.

Transaktionen

Wenn von Konsistenz gesprochen wird, muss dies natürlich auch im Kontext von Transaktionen betrachtet werden. Die bisher näher dargestellten NoSQL-Datenbanken haben als eine Gemeinsamkeit, dass sie keine Transaktionen im ACID-Sinne unterstützen. Die meisten Datenbanken, wie z. B. MongoDB, garantieren die Atomizität eines Vorgangs auf der Ebene eines Datensatzes respektive eines Dokuments. Transaktionen über mehrere Änderungsoperationen hinweg kennen sie nicht**, da sonst die Skalierung nicht gewährleistet werden kann.

Dies bedeutet, dass je nach Anwendungsfall bei einem Fehler innerhalb eines Prozesses Änderungen an Datensätzen manuell rückgängig gemacht werden müssen. Dieses Verfahren ist als „Compensation" insbesondere in verteilten Systemen wie einer SOA-Implementierung eher bekannt als in der klassischen Datenbank-Welt.

Es gibt übrigens in der NoSQL-Gemeinde tatsächlich auch mindestens eine Datenbank, die vollständig ACID und JTA (Java Transaction API) unterstützt: Neo4J.

Reifegrad

Neben den technischen Faktoren gibt es weitere, die bei der Entscheidungsfindung wichtig sind. Während aktuelle relationale Datenbanksysteme eine gewisse Historie und damit einen hohen Reifegrad aufweisen können, sind viele NoSQL-Datenbanken recht jung, bieten aber dafür eine sehr engagierte Community.

Auch sollte bedacht werden, dass viele Entwickler, wenn auch meistens nur rudimentär, SQL-Wissen besitzen, während NoSQL-Datenbanken keine standardisierte Abfragesprache oder API vorweisen können. Oft sind dadurch auch keine Ad-hoc-Abfragen möglich. Die Treiber unterliegen keinem Standard wie JDBC, sodass der Einsatz einer neuen NoSQL-Datenbank mit einer Einarbeitungsphase einhergeht, wobei APIs von Key/Value-Datenbanken eher trivial sind.

Erst jetzt entwickelt sich für die Java-Welt mit „Spring Data" [SpringData] ein Quasi-Standard für die Entwicklung mit NoSQL, sodass Entwickler sich nicht länger mit unterschiedlichsten Treiber-APIs abgeben müssen. Spring Data unterstützt die Datenbanken MongoDB, Neo4J, Riak, Redis, aber auch JPA und damit relationale Datenbanken. Interessant wird der Einsatz von Spring Data, wenn sich der Architekt für den Einsatz ver-

** Unter [MongoDB2PC] wird ein „Two Phase Commit" für dokumentenübergreifende Änderungen mit MongoDB beschrieben.

schiedener Datenbanken innerhalb eines Projektes entscheidet (Stichwort „Multi-Store“): beispielsweise in einem Shopsystem eine relationale Datenbank für Buchungsdaten, eine dokumentenorientierte für die Katalogdaten und Kundenbewertungen, eine Column Family für die Speicherung der Zugriffsdaten. Mit solch einem Ansatz werden die Vorteile der verschiedenen Systeme gebündelt.

Fazit

Für Architektentscheidungen sind mehrere Aspekte wichtig. Neben den verschiedenen Typen von Datenbanken, die für unterschiedliche Anwendungsfälle geeignet sind, müssen besondere Aspekte von verteilten Systemen im Auge behalten werden. CAP und BASE sind hier die Stichwörter.

Die Transaktionsbehandlung wird anders als in den gängigen relationalen Datenbanken gehandhabt, sodass ein Architekt hierauf eingehen sollte. Ebenso wird die Konsistenz der Daten unterschiedlich gehandhabt, starke Konsistenz ist eher die Ausnahme. Die weit verbreitete Assoziation von NoSQL mit *BigData* ist nicht zwingend, auch wenn NoSQL-Datenbanken häufig konzeptionell auf sehr große Datenmengen vorbereitet sind.

Auch sollte ein Architekt nicht zögern, bei Bedarf verschiedene Datenbanksysteme einzusetzen, Spring Data unterstützt diesen Ansatz.

All dies erfordert genaue Kenntnisse über den Anwendungsfall sowie die genutzten Datenbanken.

Literatur

[Brew00] E. A. Brewer, Towards Robust Distributed Systems, PODC-Keynote, 19.7.2000,
`http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf`

[Chang06] F. Chang et. al., Bigtable:
A Distributed Storage System for Structured Data, OSDI'06,
`http://static.usenix.org/event/osdi06/tech/chang/chang_html/`

[DeaGh04] J. Dean, S. Ghemawat, MapReduce:
Simplified Data Processing on large Clusters, OSDI, 2004,
`http://www.cs.toronto.edu/~demke/2227S.12/Papers/mapreduce-osdi04.pdf`

[Folk10] N. Folkman, So, that was a bummer, Foursquare Downtime post-mortem,
`http://blog.foursquare.com/2010/10/05/so-that-was-a-bummer/`

[GartnerHC] Hype Cycles,
`http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp`

[GartnerHCDM] Hype Cycle for Data Management, 2011,
`http://my.gartner.com/portal/server.pt?open=512&objID=249&mode=2&PageID=864059&resId=1751814`

[HäReu83] Th. Härder, A. Reuter, Principles of Transaction-Oriented Database Recovery, in: ACM Comput. Surv. 15(4): 287-317 (1983), s. a. `http://www.minet.uni-jena.de/dbis/lehre/ss2004/dbs2/HaerderReuter83.pdf`

[Hütt11] M. Hüttermann, Agiles Application Lifecycle Management (Agiles ALM), in: JavaSPEKTRUM, 5/2011

[MongoDB2PC]
`http://cookbook.mongodb.org/patterns/perform-two-phase-commits/`

[NoSQLorg] Aktuelle Schätzung von Professor St. Edlich über NoSQL-Datenbanken, `www.nosql-databases.org`

[Prit08] D. Pritchett, BASE: An Acid Alternative, ACM Queue, 1.5.2008, `http://queue.acm.org/detail.cfm?id=1394128`

[SpringData] `http://www.springsource.org/spring-data`

[Strau] Ch. Strauch, NoSQL Databases,
`http://www.christof-strauch.de/nosqldbs.pdf`

[Wolf12] E. Wolff, Architekturen für die Cloud, in: JavaSPEKTRUM, 1/2012

Dr. Halil-Cem Gürsoy ist als Software Architect bei der adesso AG tätig. Sein technologischer Schwerpunkt liegt dabei auf Java Enterprise (JEE, Spring). In diesem Kontext konzentriert er sich auf verteilte Systeme, Cloud-Architekturen sowie Build- und Deploymentprozesse in verteilten Systemen.
E-Mail: Halil-Cem.Guersoy@adesso.de
Twitter: @hgutwit