SQL vs. „NoSQL“

W. Mark Kubacki

kubacki@hurrikane.de

Kurzbeschreibung: Viele Relationale Datenbank Management-Systeme (RDBMS) gelten als horizontal nicht skalierbar, als unnötig schwer zu benutzen und bieten eine eingeschränkte Zahl an Datenstrukturen. Je angefragter Webanwendungen sind, in denen solche DBMS eingesetzt werden, desto eher führe dies an ihre Leistungsgrenzen. Systemarchitekten beginnen sich mit alternativen Lösungen zu beschäftigen. Dies führte zur ‚SQL vs. „NoSQL“‘ Bewegung, welche die Vorteile des Einsatzes von nicht-relationaler DBMS („NoSQL“) bei Webanwendungen preist und für Ihren Einsatz vom Projektbeginn an eintritt. Im Nachfolgenden setze ich mich mit den Argumenten der Bewegung auseinander, führe gleichzeitig an ihre Hintergründe heran und gebe eine Übersicht der verschiedenen „SQL“ Datenbank-Arten sowie NoSQL Datenbanken und werde erläutern, bei welchen Einsatzzwecken sie besonders geeignet sind.

1 Einführung

Vor der Einführung von Relationalen Datenbank Management-Systemen wurden bereits nicht-relationale in Programmen zur Datenhaltung eingesetzt. Obwohl es deutlich früher schon ähnliche Datenbanken (DB) gab, ist ab dem Jahr 1979 „dbm“ besonders bekannt geworden [ATT79], später mit „sdbm“, „ndbm“, „BerkeleyDB“ und vielen anderen auch benutzt als Synonym für eine Familie von Datenbanken, welche einem Primärschlüssel die eigentlichen Nutzdaten zuordnen. Während die Nutzdaten ein Binärstrom sind, der nicht weiter von der Datenbank selbst interpretiert wird, dient der Primärschlüssel zur ihrer schnellen Adressierung. Heute ist diese Art von Speicher auch als „Key/Value Store“ (KVS, zu Deutsch: „Schlüssel/Wert Speicher“) bekannt.

Im Unterschied dazu konnte man im von Edgar F. Codd [Codd70] 1970 vorgeschlagenen Datenmodell, dem „Relationalen Model“, mehr als ein Element pro Schlüssel speichern (und mittels relationaler Algebra verarbeiten). In der Tat muss der Primärschlüssel nicht zwingend vorhanden sein. Die Daten werden in Tupeln organisiert, deren Datentyp, Name und Position zuvor eindeutig zu definieren sind. Man kann sich dies als eine Matrix oder Tabelle veranschaulichen, bei der die Spaltenzahl feststeht und neue Einträge zeilenweise eingefügt werden.

SQL wurde die heute am weitesten verbreitete Sprache für dieses Datenmodell und populärster Mittler in der Entwicklung von datenbankgestützten Programmen für das Internet. Zwar wird an dieser Sprache selbst nur ihre Komplexität und bisweilen Mehrdeutigkeit bemängelt, die eigentliche Kritik richtet sich heute[1] jedoch gegen die RDBMS, in denen sie eingesetzt wird. Genauer, die Datenorganisation und Speicherung (im Englischen „data storage“) sowie ihre Grenzen hinsichtlich „Performance“, als Geschwindigkeit von Lese- und/oder Schreibzugriffen; Flexibilität der „Tabellendefinition“ sowie Skalierbarkeit mit der Rechnerzahl.

Die Bewegung der Kritiker und allgemein derjenigen, die sich mit den Unterschieden auseinandersetzen, wurde unter ‚SQL vs. „NoSQL“‘ um das Jahr 2009 [RS09] zusammengefasst, wobei ‚SQL‘ stellvertretend für die RDBMS steht und gemeinhin unter ‚NoSQL‘ nicht-relationale DBMS sowie Datenspeicher subsumiert werden.

NoSQL Datenbanken werden oft die Eigenschaften zugeschrieben, die bei RDBMS vermisst werden. Allerdings sind unter erstem Begriff viele verschiedene Datenbank-Arten zu finden, etwa KVS, Graphen-Datenbanken, Multidimensionale DB und viele andere mehr. Deshalb muss immer auf das konkrete Produkt geschaut und abgewogen werden, inwieweit sich die Optimierung auf einen Einsatzzweck negativ auf sonstige Nutzungen auswirkt und wie der Vergleich zu RDBMS gezogen wird.

Beispielsweise mögen sich KVS und verteilte KVS („distributed KVS“, DKVS) exzellent dazu eignen, ganze Internetseiten über eine Adresse abzufragen, können aber in Ermangelung einer Algebra oder eingebauten Abfragesprache nicht filtern oder ihnen mangelt es komplett an der Durchsetzung von Zugriffs- oder Benutzerrechten.

[1] In den neunziger Jahren gab es eine ähnliche Strömung zu Lasten von RDBMS und Gunsten von objektorientierten DBMS.

Während ein Tupel in einer Relation eines RDBMS zehn Elemente enthalten könnte und 1000 Tupel[2] in einer Sekunde lesend geliefert würden, wäre hier der Performance-Vergleich mit einem DKVS verfälscht, wenn nur festgestellt wird, dass letzteres 6000 Einträge liefern kann[3]: Weder ist bekannt, wie viele „Felder“ im DKVS leer waren, noch entspricht die (volle) Elementzahl des RDBMS der Eintragszahl des DKVS – mögliche Transformationen im RDBMS schon nicht berücksichtigt.

Es existieren nahezu keine Benchmarks, welche die Performance von NoSQL Datenbanken vergleichbar machen. Eines der wenigen Beispiele ist Yahoos „Cloud Serving Benchmark“ [YCSB]. Aber auch er trägt den Spezialisierungen der Datenbanken keine Rechnung – anders als diese Ausarbeitung, bei der kein Performance-Vergleich mangels Raum dargestellt werden kann, aber besonders auf die Eigenheiten eingegangen wird.

2 Grundlagen

Den Kontext zu kennen, in der Datenbanken im gesteckten Rahmen eingesetzt werden, ist unerlässlich zur Beurteilung, ob sie sich für einen Zweck besonders eignen. Zudem kann viel leichter ein Bezug zu den vielleicht dem Leser schon bekannten und implementierten Datenmodellen hergestellt werden, wenn man einige grundlegende Transformationsregeln kennen gelernt hat. Deshalb gehe ich im Nachfolgenden auf die Entwicklungsstufen einer typischen Architektur von Webanwendungen ein und leite zum Kapitel ‚Datenbanken‘ mit jenen Transformationen über:

2.1 Typische Architektur von Webanwendungen und ihre Entwicklung

Wenn nicht gerade die Datenhaltung in einfachen Datenstrukturen erfolgt die nicht serialisiert werden müssen, wie etwa bei einem Maßeinheiten-Umrechner, dann wählen viele Entwickler eine Datenbank, die entweder direkt im Prozess der Anwendung läuft („in-process database“ genannt) oder installieren eine separate auf dem gleichen Rechner. Populäre Produkte sind „SQLite“ für den ersten und „MySQL“ oder „PostgreSQL“ für den zweiten Fall.

Nun werde die Webanwendung beliebter und erhalte mehr Zugriffe. Es folgt, dass das DBMS auf einen zweiten Rechner verlagert wird, damit der eigentlichen Anwendung die volle Leistungsfähigkeit ihres Servers zur Verfügung steht. An dieser Stelle kann damit begonnen werden Daten in der Anwendung zwischenzuspeichern („Caching“), um die Latenz zu vermeiden die entsteht, wenn übers Netzwerk der Datenbankserver kontaktiert werden muss.

Die Vervielfachung der Anwendung auf mehrere Rechner ist der nächste Wachstumsschritt und nicht trivial: Setzt die Anwendung Sessions ein oder ist im Allgemeinen nicht zustandslos, dann muss sichergestellt werden, dass ein Zustand Abfragen auf seinen speichernden Server zur Folge hat, etwa indem ein Load-Balancer vorgeschaltet wird, oder alle diesen Zustand beschreibenden Daten in die Datenhaltungsschicht bzw. Datenbank externalisiert werden.

Im nächsten Schritt reicht die Performance der Datenbank nicht mehr für die viele Anwendungsserver. Unabhängig davon, was das im Einzelnen bewirkt hat. Entweder kann die Datenmenge auf mehrere Rechner aufgeteilt werden, oder der gesamte Datenbestand, vollständig oder auszugsweise, auf zusätzliche Rechner gespiegelt werden. Ich gehe hier vom letzteren aus. Es wird „Mirroring“ genannt und entlastet zwar den Rechner mit dem zu spiegelnden Datenbestand – „Master“ oder „Authoritative Server“ genannt – bei lesenden Zugriffen, findet aber seine Grenzen bei Schreibzugriffen: Diese gehen noch immer an ihn, bei zusätzlicher Belastung durch Anfragen der spiegelnden Server.

[2] Man sagt auch: „Aus einer (manifestierten) Tabelle mit zehn Feldern könnten 1000 Zeilen…“
[3] Der Autor benutzte bisher die Faustformel, ein RDBMS kann 8.000 Zeilen zu vier Spalten mit 100b und ein KVS in der gleichen Sekunde 200.000 Einträge zu 150b liefern. Sie stellt auf Beobachtungen von MySQL im Vergleich zu Tokyo Cabinet und Redis ab.

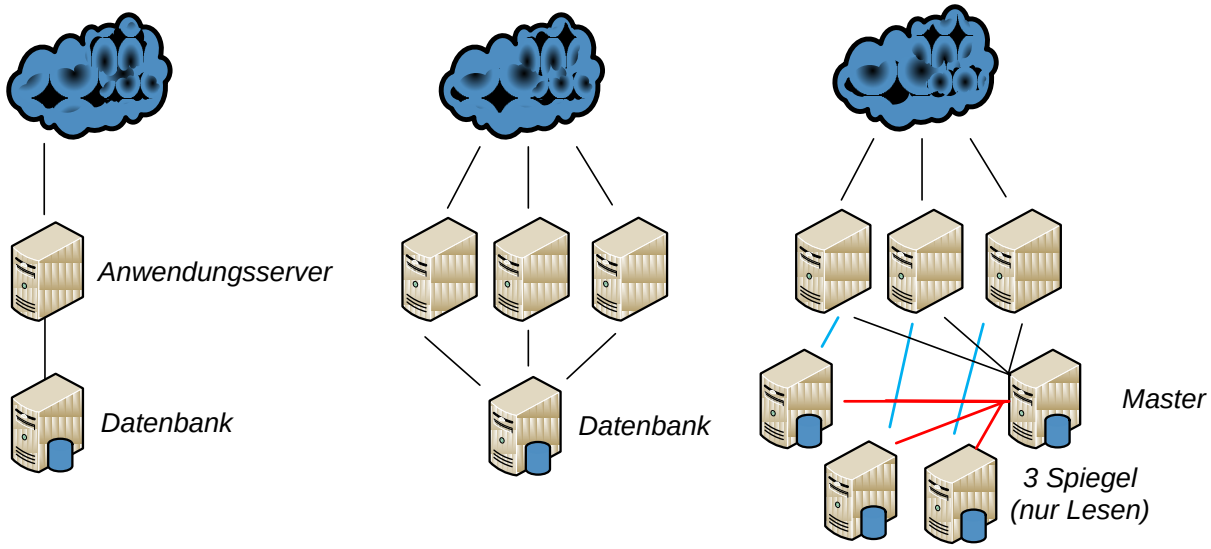
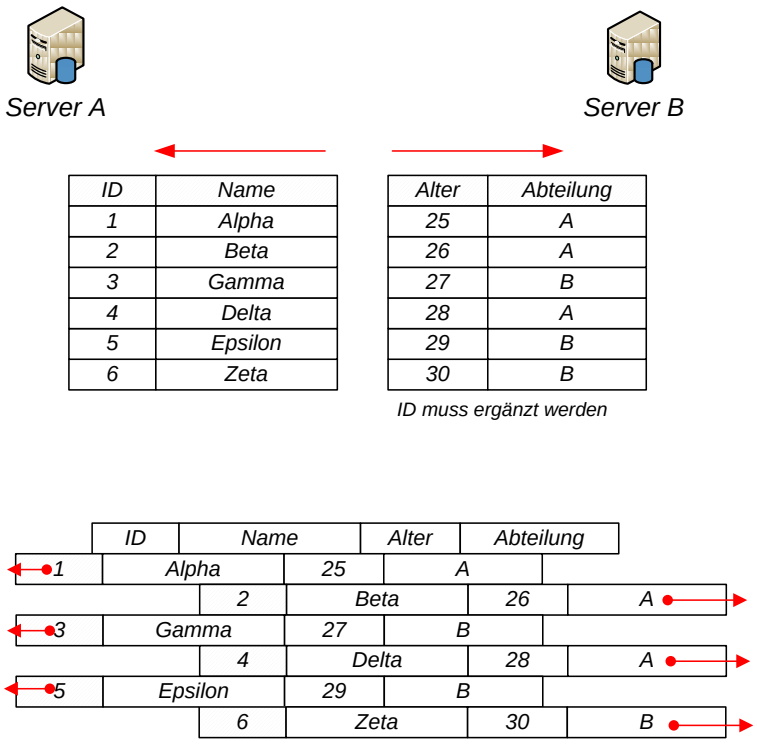


Abbildung 1 - links: Server und Datenbank; daneben: Anwendungsserver vervielfacht (ohne Loadbalancer); rechts: Master und drei Mirrors der Datenbank

Stößt die Architektur an diese Grenze, wird der Datenbestand auf eine von mehreren Arten auf gleichberechtigte Datenbankserver aufgeteilt. Diesen Vorgang nennt man auch „Partitionierung“ und unterscheidet zwischen der ‚vertikalen‘ und der ‚horizontalen‘: Bei ersterer werden ganze Relationen ausgegliedert, bei letzterem Einträge (abwechselnd) unter Beibehaltung aller Relationen an unterschiedliche Server geschickt.

Typischerweise wird sich hier des sogenannten „Shardings“ bedient. Dabei wird, wenn nicht schon vorhanden, jeder Relation ein eindeutiger Schlüssel zugeordnet. Darüber hinaus wird jedem solchen Eintragstupel über eine Funktion eine Zahl zugewiesen, die eindeutig einem Datenbankserver zugeordnet werden kann (in Abbildung 2 geschieht dies darüber, ob eine Zahl gerade ist). An diesen gehen dann sowohl Lese- als auch Schreibzugriffe. Sharding hat zur Folge, dass die Last im Allgemeinen von einem Datenbankserver sinkt, verhindert aber meist, dass vorhandene SQL Abfragen weiterbenutzt werden können: In den populären RDBMS sind keine „Join-Operationen“ über geshardete Datenbanken hinweg möglich.



ID	Name
1	Alpha
2	Beta
3	Gamma
4	Delta
5	Epsilon
6	Zeta

Alter	Abteilung
25	A
26	A
27	B
28	A
29	B
30	B

ID	Name	Alter	Abteilung
1	Alpha	25	A
2	Beta	26	A
3	Gamma	27	B
4	Delta	28	A
5	Epsilon	29	B
6	Zeta	30	B

Abbildung 2 - oben: vertikale Partitionierung; unten: horizontale Partitionierung

Die Datenzugriffs-Steuerung verlagert sich vom DBMS in die Anwendung. Es kann hier schon argumentiert werden, dass wenn SQL von RDBMS nicht mehr so umfassend benutzt werden kann, man gleich eine andere Datenbankart benutzen könne.

Eine Alternative zur vertikalen Partitionierung und Sharding ist es, den Datenbankserver durch einen leistungsfähigeren zu ersetzen. Das kann zeitaufwendige Neuprogrammierungen und komplexe Datenbankinstallationen vermeiden und wird tatsächlich von finanzkräftigen Betreibern gemacht (vgl. [Forbes10]).

Das Vermögen Leistung von Funktionsträgern durch Erhöhung der Leistungsfähigkeit einzelner Rechner steigern zu können nennt man dabei „vertikale Skalierbarkeit“, dies durch zusätzliche Rechner - oder seltener geschicktere Aufteilung der Daten – tun zu können nennt man „horizontale Skalierbarkeit“.

2.2 Alternative Architekturentscheidungen

Um nicht nur Last von den DBMS zu nehmen, sondern auch von den „Frontend“ bzw. Anwendungs-Servern, können beispielsweise Elemente der Webanwendung zwischengespeichert werden, die sich nicht häufig ändern. Bestenfalls können Anfragen mit ganzen Seiten direkt aus einem solchen Zwischenspeicher bedient werden. Kapitel 3 vorweg genommen, erfreute sich der KVS „Memcached“ hierfür einiger Beliebtheit bei seinem prominenten Vertreter, der Danga Interactive [Danga03]. (Siehe auch Kapitel 4.1.)

Wird die Anwendung in mindestens zwei eigenständige Funktionseinheiten getrennt und auf separaten Server installiert, dann kann darüber auch skaliert werden: Möglicherweise lassen sich Zugriffe schneller abhandeln als die Daten verarbeiten. In dem Fall kann zwischen die Anwendungsteile eine Warteschlange („Queue“) platziert werden und steigender Last nun unabhängig voneinander die Datenverarbeitung oder/und das Frontend skaliert werden. Die Amazon Inc. etwa bietet eine solche Warteschlange als „Simple Queue Service“ an [AMZN-SQS], es kann auch eine Lösung, die das „Advanced Message Queueing Protocol“ [AMQP] implementiert, oder eine Datenbank wie Redis [Redis] verwendet werden. (Siehe auch Kapitel 4.2.)

Die Zahl der Rechner keiner Funktionseinheit muss konstant sein. Denkbar ist, dass bei steigender Last in einem Bereich zusätzliche Rechenkapazitäten eingesetzt werden („upscaling“), die bei geringer Last wieder abgebaut werden („downscaling“). „Elastic Computing“, wie man dieses Prinzip bei einem hohen Grad an Automatisierung nennt, bedingt aber, dass nicht nur die Anwendung, sondern auch die eingesetzten Datenbanken so dynamisch skalierbar sind. Insbesondere müssen Daten auf die neuen Knoten verschoben und genauso beim Downscaling gerettet werden. Selbstverständlich eignet sich eine verteilte DB hierzu nicht, wenn vorher alle Knoten durch manuellen Eingriff bekannt gemacht werden müssen – eine durchaus gängige Fehlentscheidung bei der Konzeption jener Datenbanken.

Sobald eine größere Zahl von Anwendungs- und Datenbankservern kommunizieren muss, wird ein einzelner Server viele Verbindung zu anderen Servern offen halten müssen. Nicht nur kann das auf größten Installationen dazu führen, dass sich der Verwaltungsaufwand hierfür negativ bemerkbar macht, es kann auch zu einer hohen Auslastung des zugrunde liegenden Netzwerks bewirken. Mögliche Lösungen hierfür sind, entweder auf verbindungslose Kommunikation im Sinne von UDP umzustellen, oder ein Gruppenkommunikationsframework (etwa das Spread Toolkit [Spread]) einzusetzen. Diese können Nachrichten auch zusammenfassen, die Zustellung garantieren oder gar über Rechenzentren hinweg erst ermöglichen, zusätzlich den Anwendungsentwickler bei der Anbindung der Datenbanken entlasten. Dies setzen leider keine mir bisher bekannte (verteilte) Datenbank um bzw. solcherlei ein. Lediglich von Redis ist mir bekannt, dass mit Version 2.0 ein UDP Protokoll implementieren wird [Redis-UDP].

2.3 Transformation von Relationen zu Tripeln und Schlüssel/Wert Paaren

In RDBMS werden Relationen als Tabellen mit Zeilen dargestellt. Bei „Triple-Stores“, wie sie im Semantic Web auftreten, und KVS sieht das anders aus. Man kann jedoch eine solche Tabelle mittels folgendem einfachen Algorithmus in Tripel (Subjekt, Prädikat, Objekt) in diesem Sinne umformen:

1. Der Tabellenname ist eindeutig und wird zum Präfix A.
2. Für jede Zeile aus der Tabelle:
    1. Der eindeutige Primärschlüssel einer Zeile wird zusammen mit dem Präfix A zum ‚Subjekt'.
    2. Für jede Spalte aus der Tabelle:
        1. Ist das Feld leer, überspringe nachfolgende Schritte.
        2. Der Spaltenname zum ‚Prädikat'.
        3. Der Feldinhalt wird das ‚Objekt'.
        4. (Der Spaltentyp wird zum Objekttyp)
        5. Speichere das Tripel.

Fremdschlüsselbeziehungen gehen dabei verloren. Gegebenenfalls muss das Datenmodell vorher denormalisiert werden. Das Präfix kann, je nach Datenmodell, sinnvoller vor das Prädikat gesetzt werden.

Ausgehend von den Tripeln kann zu Schlüssel/Wert Paaren wie folgt umgeformt werden:

3. Für jedes Tripel:
    1. Das ‚Prädikat' samt Trennzeichen wird mit dem ‚Subjekt' zum ‚Schlüssel'.
    2. Das ‚Objekt' wird zum ‚Wert'.

Im Speziellen kann durchaus eine effizientere Umformung erfolgen. Beispielsweise könnten mehrere Spalten zu einem Objekt zusammengefasst werden um insgesamt die Zahl von neuen Einträgen zu verringern.

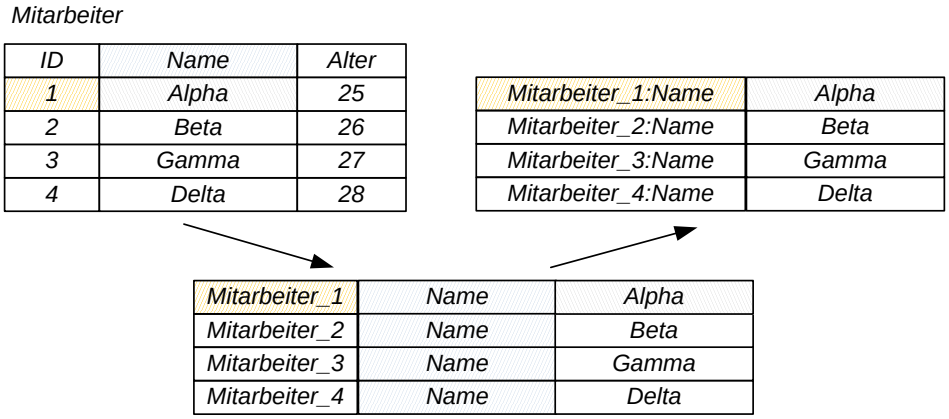


Abbildung 3 - Transformation nach einer Spalte aus einer RDBMS Tabelle (oben links) über Tripel (unten) zu Schlüssel/Wert Paaren (rechts).

3 Datenbanken

Obgleich es bei ‚SQL vs. „NoSQL"' nicht um die Konkurrenz unter Abfragesprachen, sondern um die Auseinandersetzung zwischen den Vor- und Nachteilen von RDBMS im Gegensatz zu nicht-relationalen geht, werde ich beide Begriffe als Stellvertreter für die Gegensätze beibehalten.

3.1 SQL

Wenn man in diesem Zusammenhang von SQL Datenbanken bei Webanwendungen spricht, dann denkt man an Produkte wie MySQL, PostgreSQL oder Oracle 10g. Die ‚Data Stores' dieser RDBMS speichern im Grunde genommen ihre Relationen als Tabellen zeilenweise. Felder können zu Schlüsseln, und Indices zum schnellen Zugriff definiert werden.

Eine effiziente Nutzung liegt vor, wenn ein Großteil der Felder einer Zeile genutzt wird und der Zugriff auf diese über die Indices erfolgen kann. Braucht es nicht alle Felder einer Zeile, per „SELECT" in SQL ausgewählt oder nicht, dann werden viele Daten unnötig gelesen. Die Verarbeitungsgeschwindigkeit sinkt.

Neben diesen zeilenbasierten Speichern gibt es auch spaltenbasierte. Ein kommerzieller Vertreter ist etwa Vertica [Vertica], eine „open-source“ Lösung ist MonetDB [MonetDB]. Sie speichern Felder einer Tabelle intern als Abfolge von Schlüssel/Wert Paaren in sogenannten „Binary Association Tables“, wobei in besonderen Fällen der Schlüssel gar entfallen kann. Zugriffe auf einzelne Felder sind entsprechend schnell; die DBMS ist besonders auf JOIN Operationen optimiert. Vorteilhaft sind also diese RDBMS, wenn entweder viele Felder einer Tabelle leer sind oder die Tabelle selbst viele Spalten hat bzw. dahingehend erweitert werden soll: Anders als bei den zeilenbasierten RDBMS führt das Anfügen einer (leeren oder mit gleichen Werten belegten) Spalte nicht zum zeitaufwendigen Kopieren der alten Daten in die neue Tabelle und ist somit sehr schnell.

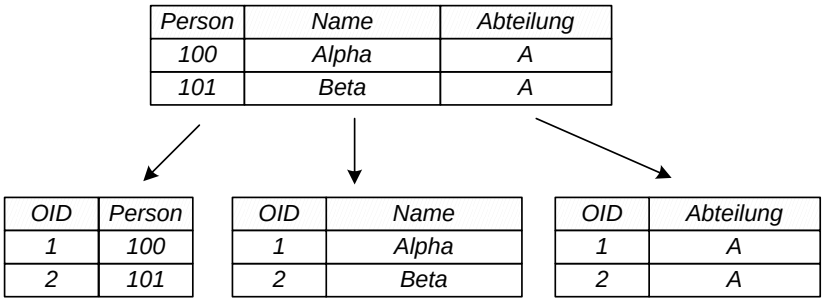


Abbildung 4 - oben: zeilenbasierte Speicherung; unten: spaltenbasierte Speicherung, in drei BAT

„Correlation Databases“ speichern Daten ähnlich den zeilenbasierten Speichern. Es werden abhängig vom konkreten Produkt entweder alle Felder einer Tabelle in einer einzigen BAT abgelegt, oder einzelne Spalten. Gleiche Feldinhalte werden dabei zusammengefasst.

Allen Lösungen haben gemeinsam, dass sie einen Dialekt von ANSI SQL [SQL] zum Zugriff und zur Verwaltung von Daten verwenden. Entsprechend muss der Anwendungsentwickler mit einer weiteren Sprache umgehen, kann sich aber auch sogenannter „Object/Relational-Mapper“ bedienen, die es erlauben, SQL in seine primäre Programmiersprache abzubilden.

In Java kann das beispielsweise Hibernate [Hibernate] oder EclipseLink Expressions [ELUG] sein, bei Python etwa SQLObject [SO].

Insgesamt wird als Nachteil von RDBMS im Allgemeinen angeführt, dass

- ein festes Schema vorliegen muss.
- die Datenbank eine Zeit lang nicht nutzbar ist, während das Schema verändert wird.
- Verteilung der Daten nicht möglich oder insofern teuer ist, als es in der Wartung und/oder Beschaffung zusätzlicher Lizenzen erheblich mehr als u.g. Alternativen kostet.
- Transaktionen bzw. Zeilen/Tabellenweise Sperren („Locking“) nicht immer nötig sind, die Verarbeitungsgeschwindigkeit aber mindern.
- mit SQL eine zusätzliche Sprache eingeführt wird.
- SQL im Allgemeinen unelegant und im Speziellen zu komplex ist.
- RDBMS „langsam“ sind.

Dem gegenüber steht:

- SQL nimmt einem Programmierer Arbeit ab. Er kann bewährte Algorithmen weiterverwenden.
- RDBMS unterstützen Mirroring, Replikation sowie „Federation“, den transparenten Zugriff auf eine entfernte Relation.
- In RDBMS wie MySQL können zusätzliche Datenspeicher implementiert werden, womit man weiter mit SQL KVS oder andere Speicherarten benutzen kann (beispielsweise [BlitzDB] für Drizzle; ein Ableger von MySQL 6.0).
- Eine teure Hardware zu kaufen sei günstiger als Programmieraufwand (vgl. auch [Forbes10]).
- Bei der Betrachtung von RDBMS wird selten zwischen zeilenbasierten und spaltenbasierten und Anderen unterschieden. Diese haben aber erheblich andere Eigenschaften, machen Kritikpunkte zum Teil gegenstandslos.

3.2 NoSQL

Unter den vielen Taxonomien (siehe auch [Yen09] via [HiScal09]) für die Vertreter der NoSQL Kategorie zeichnet sich ein Konsens zu Gunsten folgender Unterkategorien für Datenbank-Arten ab:

Kategorie	**Beispiele für Vertreter**
Key/Value Cache	Memcached, Repcached, Velocity, Terracoqa
Key/Value Store	die „dbm“ Familie, Memcachedb, Tokyo Tyrant, Keyspace[4]
Distributed Key/Value Store	Scalaris, KAI, Dynomite
Distributed Hash Table	Bamboo, Chord, Hazelcast
Data Structures Server	Redis
Tuple Store	Gigaspaces, Coord, Apache River
Object Database[5]	ZopeDB, db40, Shoal
Document Store	Scalaris, Riak, Jackrabbit, CouchDB, MongoDB
Wide Columnar Store	BigTable, Hbase, Cassandra, Hypertable
Multi-Dimensional Store	GT.M, SciDB
Graph Databases	Neo4j

‚Key/Value Caches‘ sowie Redis eignen sich beispielsweise besonders dazu, ganze Seiten zwischen zu speichern. Insbesondere bietet es sich an, einen solchen Zwischenspeicher mehreren Webservern zugänglich zu machen, damit der ausgelieferte Inhalt nur ein Mal erzeugt werden muss.

Viele KVS werden zu ‚Document Stores‘ oder ‚Wide Columnar Stores‘ weiterentwickelt. Erstere sind dergestalt, dass ein Wert wiederum eine Liste von Schlüssel/Wert-Paaren ist, die direkt vom DBMS als solche verarbeitet werden können. Riak [Riak] ist ein solches Beispiel, mit dem sogar Daten nach dem Map-Reduce-Verfahren [Google04] mit einer Vorfilterung entnommen werden können [Riak-MR]. Im Allgemeinen verarbeiten Document Stores zumindest einige Wert- bzw. Dokumenten-Formate.

‚Wide Columnar Stores‘ sowie ‚Multi-Dimensional Stores‘ haben gemeinsam, dass einem Schlüssel mehrere Werte beigeordnet werden können. Wobei erstere nicht notwendigerweise einen expliziten Primärschlüssel führen und insofern vergleichbar mit RDBMS sind, als sich die Daten als Tabellen mit fester Spaltenzahl abbilden lassen. Letztere wiederum weisen einem Schlüssel eine beliebige Anzahl von Werten zu.

Redis unterstützt viele verschiedene Datenstrukturen, beispielsweise Schlüssel/Wert Paare. Sofern der Wert ein Integer ist, kann man ihn inkrementieren, dekrementieren oder zerstörend auslesend - ursprünglich wollte der Autor eine Datenbank zum Zählen von Zugriffen auf Webseiten schreiben. Darüberhinaus lassen sich Queues über Listen, geordnete und ungeordnete Sets[6] sowie Schlüssel/Wert Paare („Map“ oder „Hash“ genannt) als Wert einem Schlüssel beiordnen und auch verarbeiten [Redis10]. Ein viel beachtetes Tutorial zu den Einsatzmöglichkeiten ist [Willison10].

3.3 Verteilte Datenbanken

Die Beobachtung, bei verteilten Systemen müssen immer an irgendeiner Stelle bei Eigenschaften wie Verfügbarkeit oder Partitionstoleranz Abstriche zu Gunsten anderer Eigenschaften gemacht werden, ist nicht neu (vgl. auch die Referenzen auf [Browne09]). Jedoch hat sie erst im Jahre 2000 Dr. Eric Brewer [Brewer00] öffentlichkeitswirksam mit seinem dann so genannten „Brewer’s CAP Theorem“ auf den Punkt gebracht:

> „Consistency, Availability, Tolerance to network Partitions.
> Theorem: You can have at most two of these properties for any shared-data system.”
> (ebenda)

[4] Verteilung erfolgt über Replikation.
[5] Siehe auch [Obasanjo01] sowie [Edlich09].
[6] Ein Set ist eine Liste ohne Duplikate.

Dieser Aussage muss unterstellt werden, dass das „shared-data system“ so verstanden wird, es sei pro System nur einen Algorithmus zur Verteilung von Daten implementiert. Das ‚Wide Columnar Store‘ Cassandra [Cassandra] zeigt, wie durch Implementierung mehrerer dem Benutzer schon in einem System die Wahl gelassen werden kann, welche Eigenschaften er bevorzugt [CassArch].

Das Theorem ist nicht unumstritten; Daniel Abadi [Abadi10] gehört zu den prominentesten Kritikern. Die Diskussion seiner Argumente würde den Umfang dieser Ausarbeitung sprengen, weshalb ich mich auf das CAP Theorem selbst beschränken werde.

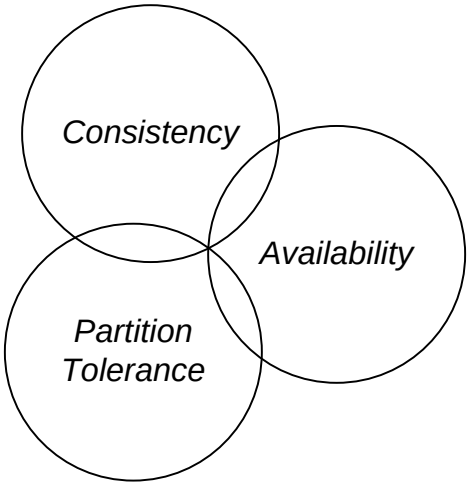


Abbildung 5 - CAP Theorem und seine Schnittmengen

„Consistency“ steht dabei für die Eigenschaft eines verteilten Systems, dass alle Knoten zu einem bestimmten Zeitpunkt bei Abfragen ein Datum, sofern sie es führen und nicht etwa eine horizontale Partitionierung vorliegt, mit dem gleichen Wert liefern würden. „Availability“ meint, das bei Ausfall von Knoten das Gesamtsystem weiter funktionieren kann. „Tolerance to network Partitions“ wiederum bedeutet, wenn die Kommunikation zwischen den Knoten so zusammenbricht, dass es von den Kommunikationsmöglichkeiten her in mindestens zwei Teile zerfiele, das System weiter operieren kann. Eine andere Betrachtungsweise der letzten Eigenschaft beinhaltet auch den Verlust von beliebigen Kommunikationspaketen.

Eine gute Erläuterung der obigen Kompromisse ist in [Browne09] sowie [Abadi10] zu finden. Letztendlich muss für den konkreten Anwendungszweck entschieden werden, welche Eigenschaften unabdingbar sind. Beispielsweise ist in einer Finanzanwendung Verfügbarkeit nicht so wichtig wie Konsistenz, in einem Zwischenspeicher für Internetseiten eines Magazins kann hingegen auf Konsistenz zu Gunsten von Verfügbarkeit und Partitionstoleranz verzichtet werden.

NoSQL Lösungen wird nachgesagt, dass:

- die unkomplizierter und flexibler zu nutzen sind.
- die Datenverarbeitung vom DBMS in die Anwendung verlagert wird.
- sie schneller sind.
- wesentlich besser und einfacher horizontal skalieren.

Aus meinen Beobachtung kann ich dem gegenüberstellen, dass:

- besonders auf die Qualität der Dokumentation geachtet werden sollte und eine Beurteilung der Quellcode-Qualität unerlässlich ist.
- die versprochene Performance sehr stark davon abhängt, ob man die NoSQL Datenbank in ihrem Spezialgebiet einsetzt.
- die beworbene Geschwindigkeit um Größenordnungen geringer ist, wenn die Datenbank nicht im Anwendungsprozess läuft. Bei Riak beispielsweise hatte ich statt 35.000 Einfügeoperationen pro Sekunde nie mehr als 500. Die gleiche Betrachtung, jedoch nicht mehr so extrem, konnte ich bei Cassandra, Scalaris, Keyspace, KAI und Dynomite (i.A. in Java oder Erlang geschriebenen Lösungen) machen.

4 Beispiele

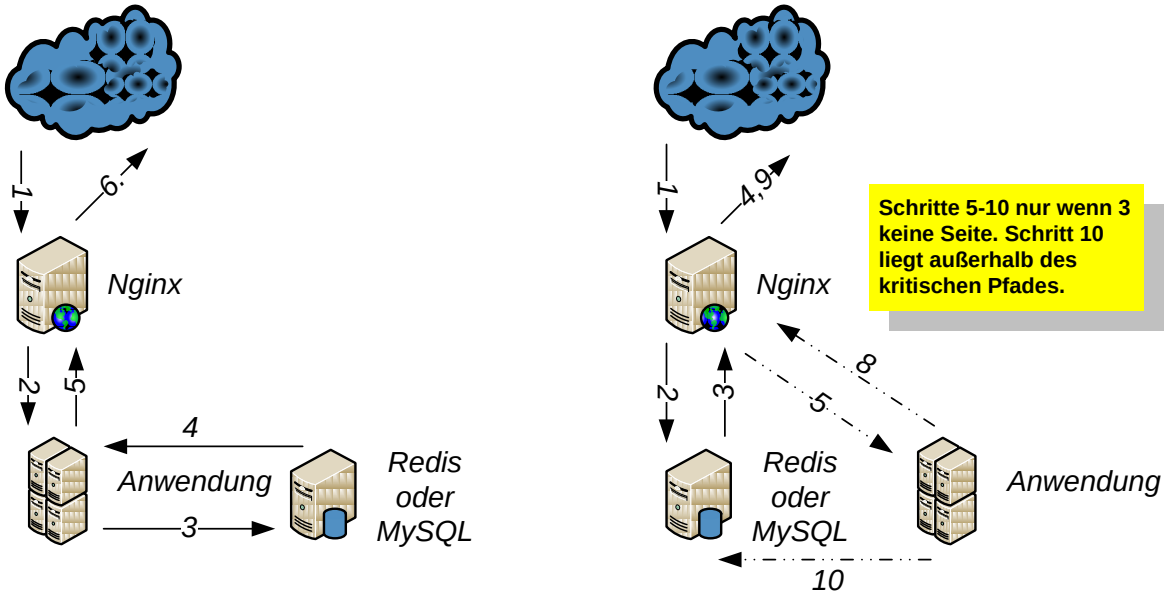
Anhand eines Wikis mit Diskussionsmöglichkeit werde ich zeigen, dass durch Einsatz geeigneter NoSQL Datenbanken nicht nur die Last pro Abfrage auf dem Server signifikant gesenkt werden kann, sondern auch dass eine Lösung per SQL umständlicher und mit einem RDBMS wie MySQL weniger performant ist.

In diesem Beispiel werde ich mich auf das Zusammenwirken der Komponenten beschränken und muss auf die Darstellung von Quellcode mit dem Verweis auf die Demo-Anwendung verzichten. Programmiersprache ist wegen der besseren Lesbarkeit Python, das Framework ist Anzu 0.3 [Anzu] und der Frontend-Server Nginx 0.8.40 [Nginx] mit Modulen [Nginx-M]. Die Anwendung läuft auf einem SheevaPlug [Sheeva]. Um das Beispiel kurz zu halten, werde ich nur MySQL 5.1.40 mit Redis 2.0.0-rc1 vergleichen.

Weitere Anwendungsbeispiele für Redis führt [Meyer10].

4.1 Cache für Seitenabrufe

Jeder Abruf einer Wiki-Seite könnte zur Folge haben, dass die Anwendung erst aus ihrer Datenbank Rohdaten liest und diese zu einer HTML Seite transformiert (auch „rendern“ genannt). Viele populäre Webanwendungen funktionieren nach diesem Prinzip, darunter auch Wordpress und Mediawiki. Es ist sehr ineffizient, weil sich die Seite nicht bei jedem Abruf ändert. Der Webserver könnte diese Seite auch gleich aus der Datenbank liefern.



1 – Anfrage
2 – Anfrage weitergereicht
3 – Anwendung an DB
4 – Rohdaten
5 – gerenderte Seite
6 – Antwort auf Anfrage

1 – Anfrage
2 – Nginx liest Redis
3 – gerenderte Seite direkt aus Redis
4 – Antwort auf Anfrage

5 – Anfrage weitergereicht
6-7 – (nicht abgebildet) Anwendung kommuniziert mit Datenbank
8 – gerenderte Seite
9 – Antwort auf Anfrage
10 – gerenderte Seite wird in Redis gespeichert

Abbildung 6 - links klassische einfache Architektur, rechts mit Redis als Zwischenspeicher

Mit Redis und Mysql ergeben sich vier Varianten der beiden Architekturen:

1. (links) Die Seitenabfrage wird über den Proxy Nginx direkt von der Anwendung beantwortet. Aus MySQL wird bei jedem Zugriff gelesen und gerendert.
2. Wie oben. Die Datenbank ist jetzt Redis.
3. (rechts) Hier wird die gerenderte Seite in MySQL abgelegt und über libdrizzle direkt von Nginx gelesen, also weder wird die Anwendung kontaktiert noch die Seite neu gerendert.

4. Wie oben. Die Datenbank ist Redis, Nginx kommuniziert mit Redis über das „HTTP Redis Module“ 0.3.1.

Nur bei der ersten Abfrage werden in Variante 3 und 4 alle zehn Schritte genommen.

Als Maß der Performance sei hier die Zeit zwischen Anfrage und Antwort (TPR) sowie die Anzahl an Seiten pro Sekunde genommen und im LAN per „*ab*“ [ApacheAB] gemessen[7]:

Tabelle 1 - Performance der vier Varianten

Variante	TPR in [ms]	Seiten/s
1	12	40
2	8	120
3	4,3	520
4	2,3	906

Entscheidend sind die Größenverhältnisse. Das Rendern der Seite brauchte exakt 2ms, die Abweichung von der TPR lag bei unter 0,1ms. Der erste Abruf wurde nicht mit gemessen und auf HTTPs „Keep-Alive“ verzichtet, damit Nginx auch immer die Datenbank kontaktiert und nicht selbst zwischenspeichert.

```
server {
        listen 127.0.0.1:80;
        server_name wiki.local;

        location /static/ {
                expires          7d;
                root             /.../wiki/static/;
        }
        location / {
                expires          8h;
                set $redis_db    "0";
                set $redis_key   "$uri?$args";
                default_type     "text/html; charset=UTF-8";
                redis_pass       127.0.0.1:6379;
                error_page       404 502 504 = @fallback;
        }
        location @fallback {
                expires          1d;
                proxy_pass http://127.0.0.1:8080;
        }
}
```

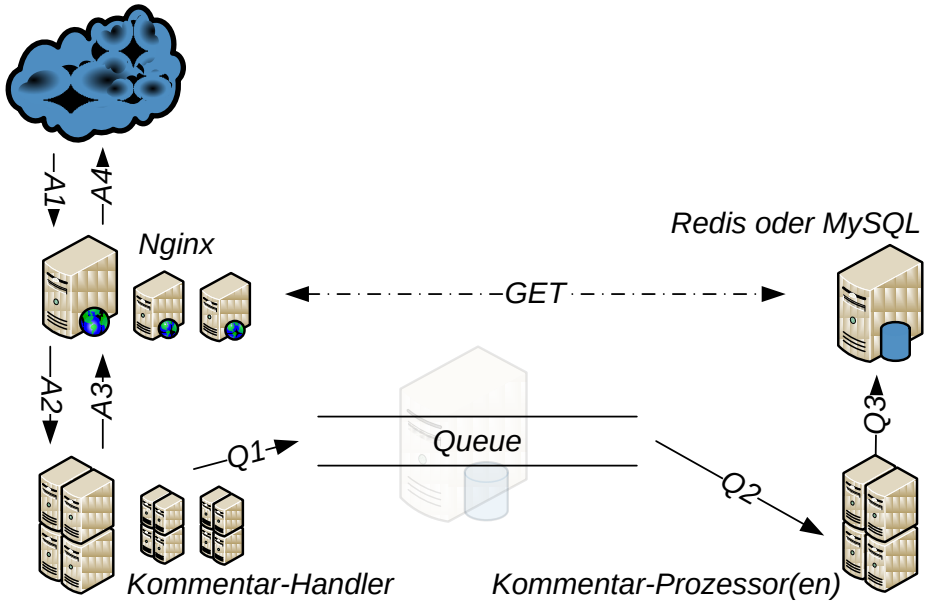
Abbildung 7 - Konfiguration von Nginx für Redis. Analog hierzu für MySQL/libdrizzle.

Ist eine angefragte Seite noch nicht in Redis oder MySQL gespeichert oder führt die Abfrage zu einem Fehler, wird die Anwendung kraft „@fallback“ aufgerufen. Der Schlüssel wird bereits in den entsprechenden Modulen soweit verändert, dass es zu keinen SQL-Injections [SQLInjection] oder dergleichen kommt.

4.2 Queue zur Entkopplung von Teilkomponenten

Angenommen, wir wollen Kommentare zu den Wiki-Seiten erlauben und fürchten, unsere beliebte Seite wird innerhalb von Sekunden tausende davon bekommen. Dann ist es zweckmäßig, ihre Entgegennahme von der Verarbeitung und Veröffentlichung durch eine Warteschlange (Queue) zu trennen:

[7] ab –c 5 –n 1000 http://127.0.0.1/ ; die gerenderte Seite war 2kb groß



A – Kommentare werden mit einem konstanten Text beantwortet.
Q – Sie werden in die Queue geschrieben, aus der sich die Prozessoren bedienen.
Q3 – Im Erfolgsfall wird die Seite ergänzt, gerendert und in die DB geschrieben.
GET – Nginx bedient sich bei Lesezugriffen direkt aus der Datenbank.

Abbildung 8 - eine Queue zur Trennung von Funktionseinheiten

Bei MySQL muss eine Tabelle die Aufgabe der Queue erfüllen. Das Auslesen eines Eintrages muss dann zwingend in einer Transaktion erfolgen, damit ein Kommentar nicht doppelt verarbeitet wird. Das könnte wie folgt aussehen:

```
BEGIN;
SELECT * FROM comments ORDER BY entry_timestamp DESC LIMIT 1;
DELETE FROM comments ORDER BY entry_timestamp DESC LIMIT 1;
COMMIT;
```

Die Tabelle ist während eines solchen Lesens für Schreibzugriffe gesperrt. Die Performance leidet darunter.

Weiterhin müssen die Kommentar-Prozessoren bei der MySQL Lösung periodisch die Tabelle auslesen um neue Kommentare in der Queue überhaupt zu erkennen. Auch dabei wird Rechnerleistung verschwendet.

Redis implementiert nicht nur Nachrichtenkanäle, sondern auch blockierende zerstörende Lesezugriffe auf Listen. In unserem Beispiel veröffentlichen die Kommentar-Handler neue Kommentare per LPUSH auf einen Schlüssel, die Prozessoren rufen in einer Schleife BRPOP auf jenen Schlüssel auf[8]:

Ist die Liste als Queue leer, blockiert der Aufruf. Die Anwendung tut nichts und kann etwa vom Betriebssystem-Scheduler schlafen gelegt werden (oder ist gleich eine ereignisbasiert Anwendung, die erst gar nicht aufwacht um geblockt zu werden). Sobald in der Queue ein Eintrag ist, wird er exakt von einem Kommentar-Prozessor empfangen.

4.3 Datenbank für Rohdaten

Ungeachtet dessen, was sich auf einer Wiki-Seite ändert, muss um eine Versionshistorie beizubehalten für jeden Versionsstand eine neue Zeile im RDBMS geschrieben werden. Alternativ kann die Tabelle soweit normalisiert werden, dass über mehrere neue Relationen mit JOIN gearbeitet werden muss.

[8] Dies sogar atomar und in O(1).

Bei einem ‚Data Structures Server' wie Redis drängen sich hierfür geradezu wieder Listen auf: Das Titelfeld wird genauso zu einer wie das Feld für den Inhalt oder die Rohdaten. Bei jeder Veränderung muss nur noch die neue Version vor alte gefügt werden. Die jeweils aktuellsten Einträge werden dann nicht per GET sondern LINDEX gelesen, wobei als Index 0 für den ersten Eintrag gesetzt wird.

Der Einzeiler in der NoSQL Datenbank ist wesentlich übersichtlicher als ein JOIN über mindestens drei Tabellen (Inhalt, Titel, Metadaten) oder die Datenstruktur ist wesentlich speichereffizienter als eine komplett denormalisierte Relation mit vielen Redundanzen. Beim RDBMS brauchen die Operationen in beiden Fällen O(log n), bei Redis wird in O(1) gelesen und geschrieben.

5 Ingenieurspraktiken

Bereits im Entwurf des Frontends der zukünftigen Webanwendung muss der Ingenieur klar die Bestandteile einer HTML-Seite von jenen trennen, die asynchron nachgeladen werden können. Je größer die unveränderlichen Blöcke, desto vorteilhafter wird ihre Zwischenspeicherung.

Bei der Entwicklung von Webanwendungen ist es von entscheidender Bedeutung sich zuerst klarzumachen, mit welchen Datenstrukturen überhaupt umgegangen wird. Erst danach sollte ein geeigneter Datenspeicher ausgewählt werden. Zwar wird nicht jede Anwendung zum neuen Facebook oder Google, aber hier führt eine falsche Wahl unweigerlich in der Zukunft entweder zu Performance-Problemen oder zeitaufwendiger Neuprogrammierung. Nicht für alles muss Redis genommen werden, aber auch nur MySQL einzusetzen kann der falsche Weg sein.

Wenn man zu viele verschiedene Datenbanken einsetzt, dann kann das dazu führen, dass der Wartungsaufwand explodiert: Auch die Zahl der Abhängigkeiten der Anwendung werden steigen sowie die verschiedenen Zugriffsmethoden.

Funktionseinheiten sollten weitestgehend entkoppelt werden, etwa durch Queues, um flexibel auf Lastspitzen in einem Bereich reagieren zu können. Dem ist ein Monitoring der entsprechende Systeme unterstellt. Im Beispiel aus 4.2 kann das so aussehen, dass von einer Leitstation aus periodisch die Länge der Warteschlange gemessen wird und bei einem Missverhältnis aus Einfalls- und Verarbeitungsrate ein weiterer Kommentar-Handler oder -Prozessor zugeschaltet wird. Bestenfalls sind die entsprechenden Systeme autonom und homogen und lassen sich aus Abbildern klonen: Dann lässt sich das volle Potential von Cloud Computing nutzen.

Die Wahl einer geeigneten Datenbank wird also auch davon beeinflusst, wie skaliert werden soll. Für eine vertikale Skalierung spricht, dass RDBMS eingesetzt werden können und lediglich die Mittel für leistungsfähigere Server (und Datenbanklizenzen für mehr Prozessoren) aufgebracht werden müssen. Bei einer horizontalen Skalierung kommen verteilte „NoSQL" Lösungen in Betracht. Allerdings zu dem Preis dass ein anderer Programmieransatz verfolgt werden muss. Insofern sollte auch hier die Entscheidung früh fallen.

In seinem Buch „Refactoring" [Fowler99], im Kapitel „Bad Smells In Code" nannte Martin Fowler es „Shotgun Surgery" – das Verändern an vielen verschiedenen Stellen im Quellcode, nur weil eine einzige Stelle ursprünglich geändert wurde. Genau dies sollte durch die strikte Trennung von Controllern und Modellen an dieser Stelle vermieden werden. Alles was mit der Datenbank direkt interagiert, gehört zu den Modellen. Einschließlich Datenbankobjekten von Object/Relational-Mappern (ORM, siehe auch 3.1) die zur Konstruktion von Abfragen verwendet werden könnten.

Meiner Erfahrung nach sollte auf ORM dann besser zu Gunsten von SQL Abfragen verzichtet werden, wenn ein Wechsel auf eine NoSQL Datenbank absehbar ist. Hierdurch entfällt zum einen die Einarbeitung in den ORM, zum anderen kann mit etwas Übung SQL leichter übersetzt werden weil eine doppelte Interpretation (SQL und ORM Unterstützung) entfällt.

Selbstverständlich sollte weniger den Performance-Angaben der Hersteller vertraut werden, sondern mögliche DBMS Kandidaten durch eigene repräsentative Daten verglichen werden.

6 Zusammenfassung

Skalierbarkeit, Performance und Einbindungs- bzw. Programmieraufwand sind die primären Dimensionen, in denen die Auseinandersetzung ‚SQL vs. „NoSQL"' geführt wird. Ein Produkt, das universell gut bei allem ist, gibt es nicht. Jedoch hat die Diskussion das Augenmerk vieler Programmierer auf alte Lösungen in neuen Implementierungen gerichtet und eine Vielzahl von Systemarchitekten umdenken lassen (Beispiele siehe [Ellis10]). Auch sind seit Beginn der Diskussion etwa mit Tokyo Cabiner, Cassandra und Redis viele neue Datenbanken entstanden, die in speziellen und nicht wenigen Einsatzbereichen RDBMS überlegen sind.

Mit der Zahl der Werkzeuge, die Programmierern, Anwendungsentwicklern und Systemarchitekten zur Verfügung stehen, steigt auch die Entscheidungskomplexität. Umso wichtiger wird nachvollziehbares, wissenschaftliches Arbeiten und besonders die eigene Fortbildung. Nicht nur in Universitäten, sondern auch in Betrieben.

Nach Evaluation vieler DBMS Vertreter muss ich feststellen, dass ein Schritt noch nicht gemacht worden ist: Nämlich der zu einer verteilten Datenbank im Sinne von „NoSQL" mit einer eigenen, standardisierten Abfragesprache. Wünschenswert wäre, dass sie mindestens ebenso viele Datenstrukturen wie Redis unterstützt, gerne auch mit starkem Einschlag von ‚Mutli-Dimensional' und ‚Wide Columnar Stores'. Wenn sie auf ein solides Gruppenkommunikationsframework aufsetzt, dann wäre das in der Tat der nächste Entwicklungssprung.

Man mag von der ‚SQL vs. „NoSQL"' Bewegung denken, sie sei bereits dagewesen oder nötig, sei falsch geführt oder provoziere zu Recht, sei destruktiv oder konstruktiv. Aber keiner stellt in Abrede, sie sei irreführend bezeichnet und mit ihren Lösungen: Bereichernd.

Der Autor entwickelt seit 2009 selbst eine verteilte Datenbank.

7 Literaturverzeichnis

[ATT79] Thompson, K.: DBM(3X), Unix Programmer's Manual, Seventh Edition, Volume 1. AT&T, 1979

[Codd70] Codd, E.: A Relational Model of Data for Large Shared Data Banks". Communications of the ACM, San Jose, CA, USA, 1970. ACM, New York, USA, 1970; S. 377-387

[RS09] Evans, E.: NoSQL 2009, http://blog.sym-link.com/2009/05/12/nosql_2009.html, Stand 12.05.2009, Zugriff 11.05.2010

[YCSB] Cooper, B. und Silverstein, A.; Tam, E. et.al: Benchmarking Cloud Serving Systems with YCSB. Santa Clara, CA, USA, 2010. http://research.yahoo.com/files/ycsb.pdf

[Forbes10] Forbes, D.: Getting Real about NoSQL and the SQL-Isn't-Scalable Lie. http://www.yafla.com/dforbes/Getting_Real_about_NoSQL_and_the_SQL_Isnt_Scalable_Lie/, Stand 02.03.2010, Zugriff 13.05.2010

[Danga03] Memcached, http://memcached.org/. Stand 14.04.2010, Zugriff 13.05.2010

[AMZN-SQS] Amazon Simple Queue Service, http://aws.amazon.com/sqs/. Zugriff 13.05.2010

[AMQP] Vonoski, S.: Advanced Message Queueing Protocol. IEEE Internet Computing 10, 2006; S. 87-89

[Redis] Redis, http://redis.io/. Stand 30.05.2010, Zugriff 09.06.2010

[Spread] Spread Toolkit, http://www.spread.org/, Zugriff 09.06.2010

[Redis-UDP] Antirez, S.: [Redis] UDP Protocol, http://code.google.com/p/redis/wiki/UDP. Stand 08.06.2010, Zugriff 09.06.2010

[Vertica] Vertica, http://www.vertica.com/. Zugriff 13.05.2010

[MonetDB] MonetDB, http://monetdb.cwi.nl/. Zugriff 13.05.2010

[Hibernate] Hibernate, Relational Persistence for Java & .NET, http://www.hibernate.org/. Zugriff 13.05.2010

[ELUG] Introduction to EclipseLink Expressions, http://wiki.eclipse.org/Introduction_to_EclipseLink_Expressions_%28ELUG%29. Stand 20.08.2008, Zugriff 13.05.2010

[BlitzDB] Maesaka, T.: BlitzDB – General Purpose Storage Engine for Drizzle. https://launchpad.net/blitzdb, Zugriff 09.06.2010

[SO] SQLObject, http://www.sqlobject.org/. Zugriff 13.05.2010

[Yen09] Yen, S.: NoSQL is a horseless carriage, http://dl.getdropbox.com/u/2075876/nosql-steve-yen.pdf. Stand 03.11.2009, Zugriff 13.05.2010

[HiScal09] Hoff, T.: A Yes for A NoSQL Taxonomy, http://highscalability.com/blog/2009/11/5/a-yes-for-a-nosql-taxonomy.html. Stand 05.11.2009, Zugriff 13.05.2010

[Obasanjo01] Obasanjo, D.: An Exploration of Object Oriented Database Management Systems. http://www.25hoursaday.com/WhyArentYouUsingAnOODBMS.html, Stand 10.01.2010, Zugriff 13.05.2010

[Edlich09] Edlich, S.: 4th Generation Object Databases. Präsentation auf der NoSQL Berlin 2009, Berlin, 2009. http://nosqlberlin.de/slides/NoSQLBerlin-DB4O.pdf

[Riak] Basho Riak, http://basho.com/Riak.html. Zugriff 13.05.2010

[Google04] Dean, J. und Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, USA, 2004. http://labs.google.com/papers/mapreduce.html

[Riak-MR] Cribbs, S.: MapReduce [in Basho Riak; Erg. des Autors]. http://wiki.basho.com/display/RIAK/MapReduce. Stand 04.06.2010, Zugriff 09.06.2010

[Redis10] Antirez, S.: Redis Command Reference. http://code.google.com/p/redis/wiki/CommandReference, Stand 25.05.2010, Zugriff 13.05.2010

[Willison10] Willison, S.: Redis tutorial. Stand 27.04.2010, Zugriff 13.05.2010

[Browne09] Browne, J.: Brewer's CAP Theorem. http://www.julianbrowne.com/article/viewer/brewers-cap-theorem, Stand 11.01.2009, Zugriff 13.05.2010

[Brewer00] Brewer, E.: Towards Robust Distributed Systems. In: ACM Symposium on the Principles of Distributed Computing, Portland, OR, USA, 2000; Folie 14 auf Seite 4 ff. http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[Cassandra] Apache Cassandra Project, http://cassandra.apache.org/. Stand 28.05.2010, Zugriff 09.06.2010

[CassArch] Cassandra Wiki: Architecture Overview. http://wiki.apache.org/cassandra/ArchitectureOverview, Stand 09.06.2010, Zugriff 09.06.2010

[Abadi10] Abadi, D.: Problems with CAP, and Yahoo's little known NoSQL system. http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html, Stand 09.06.2010, Zugriff 09.06.2010

[Nginx] Sysoev, I.: Nginx. http://nginx.org/, Zugriff 09.06.2010

[Nginx-M] [Nginx; Erg. des Autors] Third Party Modules, http://wiki.nginx.org/Nginx3rdPartyModules, Zugriff 09.06.2010

[Anzu] Kubacki, W.: Anzu. http://github.com/wmark/anzu, Zugriff 09.06.2010

[Sheeva] Marvell SheevaPlug, version 1.3 (1.2 GHz ARM5tel Prozessor, 512 MB RAM). http://www.marvell.com/platforms/plug_computer/plug_dev_kit.html

[Meyer10] Meyer, M.: A Collection Of Redis Use Cases. http://www.paperplanes.de/2010/2/16/a_collection_of_redis_use_cases.html, Stand 01.06.2010, Zugriff 09.06.2010

[SQLInjection] SQL Injection. http://en.wikipedia.org/wiki/SQL_injection, Stand 09.06.2010, Zugriff 09.06.2010

[Fowler99] Fowler, M. et.al: Refactoring: Improving the Design of Existing Code. Addison Wesley, Boston, USA, 1999.

[Ellis10] Ellis, A.: Cassandra in Action. http://spyced.blogspot.com/2010/03/cassandra-in-action.html, Stand 04.06.2010, Zugriff 09.06.2010