5 NoSQL-Datenbanken am Beispiel von MongoDB

Neben den in der Praxis seit Jahrzehnten bewährten und bereits besprochenen relationalen Datenbankensystemen (RDBMS) gewinnt der Markt für NoSQL-Datenbanken immer mehr an Bedeutung. Daher möchte ich Ihnen in diesem Kapitel einen einführenden Überblick über dieses aktuelle Thema geben.

5.1 Einführung und Überblick

NoSQL-Datenbanken zeichnen sich dadurch aus, dass sie eine alternative Form der Datenspeicherung vornehmen und dazu keine Tabellen wie traditionelle RDBMS nutzen. NoSQL ist eine Antwort auf den zunehmenden Bedarf

- nach flexibler Datenspeicherung (kein fixes Modell),
- an extrem umfangreichen zu verarbeitenden Datenmengen und
- nach guter, insbesondere horizontaler Skalierbarkeit (verteilter Datenspeicherung).

Wieso sind diese Eigenschaften so erstrebenswert? Versuchen wir, in diesem Kapitel eine Antwort darauf zu finden.

Hinweis: Namensgebung »NoSQL«

NoSQL ist momentan ein viel beachtetes Thema, wobei der Name etwas unglücklich gewählt ist, weil man vermuten könnte, dass er für »KEIN SQL« steht, was aber nicht stimmt. Schaut man nur auf den Namen, so könnte dieser zudem ein wenig so klingen, als ob NoSQL eine Kampfansage an RDBMS oder als Ersatz dafür gedacht wäre. Doch auch das stimmt nicht, sondern NoSQL stellt vielmehr eine sinnvolle Ergänzung dar, und zwar dort, wo traditionelle RDBMS Schwächen zeigen.

Demzufolge meint die Abkürzung »Not Only SQL«, die meiner Ansicht nach auch unglücklich ist und nicht den Kern trifft, denn der Fokus liegt nicht auf SQL, sondern darauf, dass alle NoSQL-Datenbanken KEIN relationales Datenmodell besitzen. Stattdessen findet man verschiedene andere Formen der Datenspeicherung, etwa die Speicherung von Schlüssel-Wert-Paaren, Dokumenten, Graphen usw. Bei allen spricht man von NoSQL-Datenbanken.

Aktuelle Entwicklungen

Die Menge der von Applikationen zu verarbeitenden Daten hat in den letzten Jahren extrem zugenommen. In einigen Bereichen ist sie förmlich explodiert. Das lässt sich an einem Beispiel verdeutlichen. Den Trend stetig wachsender Datenmengen und steigender Anzahl an Benutzern kann man anhand von Twitter gut erkennen: Waren es 2010 noch etwa 65 Millionen Tweets pro Tag, so hat sich die Anzahl innerhalb eines Jahres mehr als verdreifacht. Im Jahr 2011 wurden etwa 200 Millionen Tweets pro Tag ermittelt. Aber auch in vielen anderen Bereichen, besonders in sozialen Netzwerken, sind solche Trends beobachtbar. Insgesamt werden für die Zukunft die Themen ***Big Data*** und ***Data Analytics***[1] immer wichtiger. Dabei sind nahezu unüberschaubare Datenmengen in Tera- und Peta-Bytes oder gar mehr zu verarbeiten. Das stellt eine große Herausforderung dar und gilt insbesondere für die herkömmliche Speicherung in RDBMS, die aus mehreren Gründen zunehmend an Grenzen stößt: Zwar läuft ein RDBMS meistens auf einem dedizierten, sehr leistungsfähigen Rechner, aber auch dieser besitzt gewisse Grenzen in seinem Ausbau. Erschwerend kommt hinzu, dass von RDBMS verschiedenste Konsistenzbedingungen sichergestellt werden müssen und dazu aufwendige Aktionen wie das Sperren von Datensätzen erforderlich sind. Dadurch wird die Möglichkeit zur Parallelverarbeitung eingeschränkt und zudem die Abarbeitungsgeschwindigkeit verringert.

Neben dem Anstieg der reinen Datenmenge beobachtet man Folgendes:

- Die Daten sind häufig immer stärker miteinander vernetzt (Facebook, XING,...). Ein Datensatz allein bietet kaum einen Mehrwert, sondern erst deren Kombination.
- Die Daten sind weniger strukturiert und besitzen eine größere Varianz.
- Die transaktionalen Eigenschaften, vor allem die Isolation, sind weniger wichtig.
- Die Erwartungshaltung an Geschwindigkeit und Durchsatz nimmt zu.

Varianten von NoSQL-Datenbanken

Im Bereich der NoSQL-Datenbanken gleicht keine der anderen. Trotz der Unterschiede lassen sich aber vier Hauptströmungen (mit Beispiel in Klammern) ausmachen:

- **Key Value Stores** (Analogie: `Map<K,V>`) – Speichern zu einem Schlüssel einen Wert, wobei ein Wert durchaus komplexe Daten enthalten kann. (REDIS)
- **Document Stores** (Analogie: `List<Object>`[2]) – Speichern Einträge als Dokumente, die Objekten ähneln; Dokumente müssen keiner festen Struktur folgen. (MONGODB, COUCHDB)

[1] Bei dieser Form der Datenanalyse hat man nicht nur große Datenmengen zu bewältigen, sondern versucht auch noch Korrelationen und Muster zu erkennen.

[2] Die Liste entspricht einem Dokument – weil Document Stores mehrere sogenannte Collections verwalten, wäre `Map<String, List<Object>>` für den Document Store als Ganzes die korrektere Analogie.

- **Column Stores** (Analogie: `Map<RowKey,Map<ColumnKey,ColumnValue>>` (geschachtelte Maps)) – Einer Zeile können beliebig viele Spalten zugeordnet werden; Datensätze mit potenziell sehr vielen dynamischen Spalten. (CASSANDRA)
- **Graph DBs** (Analogie: Referenzierungen zwischen Java-Objekten) – Speichern Graphen, also Knoten und Verbindungen dazwischen; ideal um Netzwerke und Beziehungen zu modellieren; in RDBMS sind diese Verbindungen nur durch aufwendige und viele Verknüpfungen (sogenannte Joins) ermittelbar. (NEO4J)

Diese Typen von NoSQL-Datenbanken besitzen mitunter proprietäre Abfragesprachen und APIs, womit wir einen entscheidenden Nachteil gegenüber RDBMS und SQL als deren Standard für Zugriffe aufgedeckt haben. Zwar gibt es für SQL auch verschiedene Dialekte, allerdings sind die Unterschiede so gering, dass Frameworks wie Hibernate oder JPA sogar nahezu datenbankunabhängige Abfragen ermöglichen.

Ein Blick zurück auf RDBMS

Bekanntermaßen werden Daten in RDBMS in Tabellen gespeichert. Eine Zeile entspricht einem Datensatz und besitzt die durch die Spalten festgelegten Eigenschaften. Dabei gibt ein Schema den möglichen Inhalt vor und erlaubt es dadurch, Integritäts- und Konsistenzprüfungen durch die Datenbank automatisch auszuführen.

Datensätze werden zeilenweise eingefügt und können über Schlüssel gefunden werden. Vielfach werden Daten zur Vermeidung von Redundanz und zum Sparen von Speicherplatz, der in den frühen Jahren der Datenbanken ein heiliges Gut war, auf diverse Tabellen verteilt. Die auf verschiedene Tabellen verstreuten Daten können später über Joins wieder passend aggregiert werden. Die dazu notwendigen Verknüpfungen zu anderen Tabellen erfolgen über Fremdschlüsselbeziehungen.[3] Insgesamt ist das ein gut verstandenes Modell, das mit SQL eine mächtige, aber trotzdem einfach nutzbare, deklarative Abfragesprache besitzt. Sie erlaubt oftmals eine fast umgangssprachliche Formulierung von Abfragen, sofern deren Komplexität nicht allzu hoch ist. Zudem lässt sich die darunterliegende Komplexität der Zugriffe auf die Datenbank vor Benutzern weitestgehend verbergen.

Die Organisation der Daten in Tabellen bereitet aber auch Probleme und wirft unter anderem folgende Fragen auf, für die es mit RDBMS keine optimalen Lösungen gibt:

- Wie speichert man Listen in Attributen, etwa die Hobbys einer Person?
- Wie geht man mit Varianzen oder optionalen Werten in den Daten um?[4]
- Wie repräsentiert man Verbindungen zwischen Objekten, also die durch die Referenzierungen entstehenden Objektnetze?
- Wie modelliert man Vererbungshierarchien?

[3] Daher kommt auch die namensgebende Eigenschaft »relational«: Entitäten werden in Tabellen gespeichert und in Beziehung (Relation) zueinander gesetzt.

[4] Einzelne `NULL`-Werte sind nicht problematisch. Mühsam wird es aber, wenn ganze Teile des Objektgraphen optional sind. Das erfordert recht komplexe Abfragen.

Darüber hinaus kann man weitere Schwachstellen bei RDBMS ausmachen. RDBMS liefern bei großen Systemen und hoher Anfragelast – insbesondere in Kombination mit Schreibzugriffen und vor allem im Transaktionskontext – mitunter keine allzu gute Performance mehr. Das resultiert aus der Forderung nach Konsistenz und Isolation bei ACID-Transaktionen (Atomicity, Consistency, Isolation und Durability), was wiederum das Sperren von Zeilen oder Bereichen in Tabellen erfordert. Dadurch werden die Möglichkeiten zur Parallelverarbeitung und damit indirekt auch die Verarbeitungsgeschwindigkeit vermindert.

Tipp: Auswirkungen der Isolationslevel bei Transaktionen

In den vorangegangenen Diskussionen zu Transaktionen habe ich mögliche negative Auswirkungen auf die Performance erwähnt. Über verschiedene Isolationslevel lässt sich einstellen, wie sich verschiedene Aktionen in Transaktionen gegenseitig beeinflussen. Nur für das strengste Level `SERIALIZABLE` sind Transaktionen vollständig gegeneinander abgeschottet. Bei schwächeren Isolationsleveln muss die Datenbank für Read-Only-Transaktionen dagegen keine Locks halten. Damit ist die Performance beim Lesen meist weniger das Problem. Schwieriger ist jedoch die Wahrung von Konsistenz zum Zeitpunkt des Schreibens.

Grundlagen Document Stores – Analogien und Unterschiede zu RDBMS

Da wir uns später MongoDB als Vertreter der Document Stores ein wenig genauer anschauen wollen, gebe ich hier eine kurze Einführung in Document Stores und deren Analogie zu RDBMS. Die meisten Ähnlichkeiten bestehen in folgenden Punkten:

- Die Datenspeicherung in Document Stores erfolgt in Collections, ähnlich zu Tabellen in RDBMS.
- Die Daten sind als Dokument bzw. Objekt repräsentiert, was in etwa mit einer Zeile einer Tabelle vergleichbar ist.
- Die Dokumente besitzen Attribute (etwa analog zu den Spalten im RDBMS).

Im Unterschied zu RDBMS sind Document Stores schemafrei. Das bedeutet:

- Dokumente müssen keine einheitliche Struktur besitzen und deren Aufbau muss beim Einrichten der Datenbank nicht zwingend bekannt sein.
- Objekte können eine (nahezu) beliebige Struktur aufweisen und aus verschiedenen Attributen bestehen.
- Typen und Attribute können sich von Dokument zu Dokument unterscheiden.
- Attribute können auch mehrere Werte enthalten, z. B. eine Aufzählung von Hobbys oder eine Menge von Adressen.
- Attribute können insbesondere andere Objekte einbetten, d. h., Daten können hierarchisch strukturiert sein, etwa Adressinformationen innerhalb einer Person.

Beispiel zur Schemafreiheit

Um das Ganze etwas besser verstehen zu können, betrachten wir die Repräsentation zweier Personen. Nachfolgend wollen wir Max Mustermann und Michael Inden in Form von JSON (JavaScript Object Notation) modellieren – ähnlich dazu, wie Daten in MongoDB gespeichert werden. An diesem Beispiel lassen sich einige der genannten Eigenschaften von schemafreien Daten darstellen. Wir erkennen die offensichtliche Gleichartigkeit, aber auch die Unterschiede in der Granularität der Adressinformationen. Zudem differieren die Angaben von Hobbys und Programmiersprachen:

```
[
    {
        "name" : "Mustermann",
        "vorname" : "Max",
        "adresse" : {  "ort" : "Musterstadt",
                       "plz" : "52070",
                       "strasse" : "Musterstrasse"
                       "hausnr" : "17a"
                            },
        "hobbys": [ "Lesen", "Kino" ]
    },
    {
        "name" : "Inden",
        "vorname" : "Michael",
        "adresse" : { "ort" : "Zürich" },
        "programmiersprachen" : [ "Java", "Groovy", "C++" ]
    }
]
```

Diskussion zur Schemafreiheit

Man erkennt, dass durch die JSON-Notation eine verständliche und menschenlesbare Darstellung erfolgen kann. Auch das ORM oder genauer ODM,[5] also das Mapping von Objektdaten in und aus der Datenbank, wird oftmals einfacher als bei RDBMS.[6] Der bei RDBMS auftretende Impedance Mismatch – also die Frage: Wie bilde ich Objekte auf Tabellen ab? – das wird hier einfach durch die direkte Speicherung gelöst: Ein Objekt kann als JSON an die Datenbank übergeben werden. Dort kann es dann in eine interne Darstellung überführt werden. Im Falle von MongoDB ist dies eine BSON (für Binary JSON) genannte Repräsentation, die etwas Speicherplatz schonender als JSON ist. Optionale Attribute oder die Speicherung von Java-Collections (Arrays, Listen usw.) sowie Vererbung stellen gewisse Herausforderungen beim Einsatz von RDBMS dar. Auch hier macht man sich die Schemafreiheit von NoSQL-Datenbanken zunutze. Das vermeidet eine Aufteilung der Daten auf mehrere Tabellen, wie man es für RDBMS kennt.

[5] Bei Document Stores spricht man von O/D-Mapping (Object/Document Mapping, kurz ODM).

[6] Allerdings nur dann, wenn die Varianz in der Struktur der Daten überschaubar bleibt. Ansonsten erweist sich die Rücktransformation von Daten aus der Datenbank in Objekte einer Klasse oftmals als schwierig.

Mit der Einbettung von Daten und durch die Schemafreiheit kann man davon ausgehen, dass dadurch etliche Probleme gelöst würden. Das stimmt häufig. Allerdings gibt es auch negative Eigenschaften. Durch die Schemafreiheit müssen in jedem Datensatz wiederum die Attributnamen gespeichert werden, wodurch der Speicherbedarf pro Datensatz höher als bei einem RDBMS ausfällt: Dort müssen die durch das Schema vorgegebenen Attributnamen (Spalten) nicht erneut abgelegt werden, da diese fix vorgegeben sind. Speicherplatz ist aber heutzutage oftmals nicht mehr der kritische Faktor.

Viel entscheidender ist jedoch ein anderes Kriterium: die Konsistenz der Daten. Wir haben im Beispiel die Angabe von Personen und die Einbettung von deren Adressen gesehen. Wenn wir jetzt annehmen, dass sich für die Modellierung die Postleitzahlen oder andere Bestandteile ändern, so führt dies dazu, dass zur Gewährleistung von Konsistenz alle Datensätze überprüft und gegebenenfalls aktualisiert werden müssten. Dies wurde bereits bei der Vorstellung der Normalformen diskutiert (vgl. Abschnitt 2.1.3). Zur Vermeidung von Redundanz nutzt man in RDBMS die Normalisierung. Im Beispiel werden Personen und Adressen getrennt gespeichert und verweisen aufeinander. Eine Änderung einer Adresse führt dann zu einer Aktualisierung lediglich eines Datensatzes und ist danach konsistent für alle Nutzer.

Wenn man die Normalisierung einmal außer Acht lässt, könnte man auf die Idee kommen, JSON auch in RDBMS zu speichern. Das wäre möglich, aber man würde einen großen Vorteil eines Document Store verlieren: die komfortable Suchfunktionalität. In einem RDBMS würde JSON wohl in der Regel in einer Spalte gespeichert. Dadurch müssten bei vielen Datensätzen Volltextsuchen innerhalb von JSON erfolgen, die die Performance negativ beeinflussen können. An Dinge wie Escaping, Zeilenumbrüche usw. wollen wir erst gar nicht denken. Verwerfen wir also die Idee ganz schnell.[7]

Hinweis: Gedanken zur Schemafreiheit

Bei Document Stores hört man häufiger den Begriff ***Schemafreiheit***. Damit ist gemeint, dass einem Document Store im Gegensatz zu einem RDBMS kein vorgegebener Aufbau (das Schema) bekannt gemacht werden muss. Jedoch wird durch die Verarbeitung der Daten in der Applikation selbst implizit ein Schema angenommen. Dieses ist allerdings nicht fix, und es ist jederzeit möglich, dass Einträge nach altem und neuem Schema in einer Collection koexistieren können. Dieser Umstand kann für Wartungsarbeiten und Datenmigrationen praktisch sein. Betrachten wir dies etwas genauer.

Möchte man z. B. das Domänenmodell einer Applikation mit RDBMS aktualisieren, ist das mit einer Anpassung des Schemas und einer Migration aller bestehenden Daten mit einer entsprechenden Ausfallzeit verbunden. Bei NoSQL hingegen muss kein dediziertes Wartungsintervall mit einer gewissen Downtime vorgesehen werden, da altes und neues Schema parallel zueinander existieren können. Gerade für (globale) rund um die Uhr genutzte Systeme kann dies ein großer Vorteil sein.

[7] Aktuelle RDBMS, wie MySQL 5.7.8, haben in diesem Bereich aufgeholt und bieten eine direkte Unterstützung zur Verarbeitung von JSON.

5.2 Einführung MongoDB

Nachfolgend möchte ich Ihnen MongoDB[8] als einen populären Vertreter der NoSQL-Datenbanken vorstellen. MongoDB ist für alle gängigen Betriebssysteme frei verfügbar und zeichnet sich dadurch aus, dass die Installation kinderleicht ist und für eine Vielzahl gebräuchlicher Programmiersprachen Treiber mitgeliefert werden.

MongoDB lässt sich grob durch folgende Eigenschaften charakterisieren:

- Dokumentorientiert und schemafrei
- Für große Datenmengen ausgelegt
- Ausfallsicherheit und Skalierbarkeit
- Abfragen gemäß dem Muster Query-by-Example in Form von Dokumenten

Für die in MongoDB gespeicherten Daten gelten folgende Charakteristika:

- Jeder Datensatz bzw. jedes Objekt in MongoDB besitzt eine eindeutige ID.
- Diese ID ist ein künstlicher Primärschlüssel.
- MongoDB baut automatisch einen Index über die ID auf.
- Es können weitere Indizes basierend auf beliebigen Attributen definiert werden, wodurch Abfragen beschleunigt werden können.

Installation von MongoDB

Die Installation von MongoDB gestaltet sich ziemlich einfach. Laden Sie MongoDB von der Seite `http://www.mongodb.org/` herunter und entpacken Sie es in einen beliebigen Ordner[9]. Im Unterverzeichnis `bin` findet man sowohl den MongoDB-Server als Applikation (`mongod`) als auch den Mongo-Client als Kommandozeilentool (`mongo`). Nun müssen Sie nur noch den MongoDB-Server starten und schon können erste Experimente beginnen, z. B. Daten eingefügt und Abfragen ausgeführt werden. Beim Start ist allerdings noch eine Kleinigkeit zu beachten: Man sollte ein Verzeichnis angeben, in dem MongoDB seine Informationen zu Datenbanken und deren Inhalt speichert. Der Aufruf geschieht wie folgt:

```
mongod --dbpath <path_to_db>
```

Auf diese Weise wird der MongoDB-Server mit den Datenbanken aus dem angegebenen Pfad[10] gestartet und kann mit dem im Anschluss kurz beschriebenen Mongo-Client kontaktiert werden. Weiterführende Details sowohl zu MongoDB-Server als auch Mongo-Client entnehmen Sie bitte der recht ausführlichen und guten Dokumentation auf der schon genannten MongoDB-Webseite.

[8] Der Name ist angelehnt an hu***mongo***us (gigantisch).

[9] Bei Windows etwa im Verzeichnis `C:\Tools\MongoDB` – für Mac OS beispielsweise `/usr/local/var/mongodb`.

[10] Somit kann man einfach mehrere Datenbanken in verschiedenen Verzeichnissen verwalten.

Mongo-Client

Die Distribution von MongoDB liefert den Mongo-Client als Kommandozeilentool mit. Damit lassen sich Kommandos ausführen – standardmäßig verbindet sich der Mongo-Client dazu mit einer lokalen Datenbank auf Port 27017.[11]

Für die ersten Gehversuche sind folgende Kommandos nützlich:

Tabelle 5-1 *Nützliche erste Kommandos des Mongo-Clients*

Kommando	Bedeutung
`show dbs`	Liste der verfügbaren Datenbanken
`use <dbname>`	Aktivieren der Datenbank mit dem Namen `dbname`
`show collections`	Zeigt die Collections der aktuellen Datenbank
`help`	Anzeige einer Hilfeseite
`db.help()`	Hilfe zu Datenbankzugriffen
`db.<mycol>.help()`	Hilfe zu Methoden auf der Collection `mycol`

Beispieldatenbank mit dem Mongo-Client vorbereiten

Zunächst schauen wir uns an, welche Datenbanken innerhalb von MongoDB existieren. Also tippen wir folgenden Befehl ein:

```
show dbs
```

Wir erhalten in etwa Folgendes als Antwort:

```
local         0.078GB
```

Bei `local` handelt es sich um die in MongoDB vordefinierte Datenbank. Für unsere Entdeckungsreise zu MongoDB wollen wir eine eigene Datenbank einsetzen. Starten wir deshalb mit dem Kommando `use mongoexample`. Damit aktivieren und nutzen wir die als Parameter übergebene Datenbank. Sofern diese noch nicht existiert, wird sie angelegt. Nun geben wir das Kommando `db` ein, um zu sehen, welche Datenbank gerade verwendet wird. Als Konsolenausgabe erhalten wir `mongoexample`.

Überraschenderweise würde eine Wiederholung des Kommandos `show dbs` die Datenbank jedoch noch nicht auflisten. Das liegt daran, dass von MongoDB noch keine Collections angelegt wurden. Das prüfen wir! Dazu tippen wir folgenden Befehl ein:

```
show collections
```

[11] Optional kann bei Bedarf eine abweichende Portnummer beim Start durch den Kommandozeilenparameter `--port <Port-Nr>` festgelegt werden.

Wir erhalten eine leere Ausgabe, da es (wie erwartet) in der Datenbank `mongoexample` noch keine Collections gibt. Wir wollen nun eine eigene Collection erstellen. Weil wir im Umgang mit MongoDB noch nicht so firm sind, nutzen wir die Hilfefunktion: Wir tippen `db.help()` und erhalten eine Liste von Kommandos. Anhand der Auflistung erkennen wir, dass alle Kommandos, die sich auf eine konkrete Datenbank beziehen, auf einem impliziten Datenbankobjekt namens `db` arbeiten. Dabei steht `db` für die gerade gewählte Datenbank, hier also `mongoexample`.

In der Liste der Kommandos finden wir auch `createCollection()`. Diesen Befehl führen wir wie folgt aus, um eine Collection `persons` zu erzeugen:

```
db.createCollection("persons")
```

Wiederholen wir nun das Kommando `show collections`, so erhalten wir folgende Ausgaben:

```
persons
system.indexes
```

Es wurde also die Collection `persons` angelegt ebenso wie eine interne Collection `system.indexes` zur Verwaltung von Indexinformationen. Werfen wir im Folgenden einen Blick auf weitere Befehle zur Verarbeitung von Collections.

5.2.1 Analogie von CRUD (RDBMS) zu IFUR (MongoDB)

Mittlerweile sind wir schon ein wenig warm mit dem Mongo-Client geworden. Daher wollen wir uns nun damit befassen, Daten einzufügen, zu lesen, zu verändern und zu löschen, also die bei RDBMS durch das Kürzel CRUD beschriebenen Aktionen auszuführen. CRUD steht bekanntermaßen für die vier sehr gebräuchlichen Operationen im Zusammenhang mit Datenbanken: Create, Read, Update und Delete.[12] Bei MongoDB benutzt man etwas abweichende Begriffe dafür. Hier kann man sich das Kürzel IFUR merken, das sich, wie in Tabelle 5-2 gezeigt, auf CRUD abbilden lässt.

Tabelle 5-2 *Abbildung von CRUD auf IFUR*

RDBMS	MongoDB
CREATE	INSERT
READ	FIND
UPDATE	UPDATE
DELETE	REMOVE

[12] Ein wenig im Spaß wies mich Tobias Trelle darauf hin, dass es das Kommando Read in SQL ja gar nicht gibt ;-) und dieses vielmehr einem `SELECT` entspricht.

INSERT

Wir wollen unsere ersten Objekte in der MongoDB speichern. Das soll unter anderem die Person Hugo Test mit dem Alter 33 sein, die wir als JSON wie folgt darstellen:

```
{ "name" : "Test", "firstname" : "Hugo", "age" : 33 }
```

Wir wissen bereits, dass in einem Document Store die Dokumente bzw. Objekte in Collections gespeichert werden. Zum Einfügen nutzen wir das implizite `db`-Objekt und geben dahinter den Namen der Collection sowie das auszuführende Kommando an:

```
db.persons.insert({ "name" : "Test", "firstname" : "Hugo", "age" : 33 })
db.persons.insert({ "name" : "Meyer", "firstname" : "Tim", "age" : 7 })
```

FIND

Prüfen wir nun durch einen Aufruf von `find()`, welche Daten in der Collection `persons` tatsächlich gespeichert sind. Wir führen folgendes Kommando aus:

```
db.persons.find()
```

Die Ausführung dieses Kommando liefert die zwei zuvor eingefügten Personenobjekte als Treffermenge – man erkennt, dass automatisch das Attribut `_id` von MongoDB vergeben wird:

```
{ "_id" : ObjectId("561e9b62f558714ee1029d3b"),
  "name" : "Test", "firstname" : "Hugo", "age" : 33 }
{ "_id" : ObjectId("561e9bbbf558714ee1029d3c"),
  "name" : "Meyer", "firstname" : "Tim", "age" : 7 }
```

Hinweis: Ansprechende Aufbereitung von Ergebnissen

Die gerade gezeigte Abfrage – sowie Abfragen im Allgemeinen – produzieren eine recht kompakte, manchmal jedoch nicht optimal lesbare Ausgabe. Das `pretty()`-Kommando bereitet das JSON ansprechend auf:

```
db.persons.find().pretty()
{
    "_id" : ObjectId("561e9b62f558714ee1029d3b"),
    "name" : "Test",
    "firstname" : "Hugo",
    "age" : 33
}
{
    "_id" : ObjectId("561e9bbbf558714ee1029d3c"),
    "name" : "Meyer",
    "firstname" : "Tim",
    "age" : 7
}
```

Spezifikation von Suchbedingungen Eher selten sind alle Datensätze einer Collection zu ermitteln. Oftmals soll die Treffermenge wie bei einem SQL-`SELECT` mithilfe einer `WHERE`-Angabe eingeschränkt werden. Dazu kann man dem Kommando `find()` ein JSON-Objekt mitgeben, das die Suchbedingung in Form eines Beispieldokuments spezifiziert. Diese Art der Suche wird ***Query By Example*** genannt. Folgendermaßen sucht man nach Datensätzen mit dem Attribut `firstname` und dem Wert `Tim`:

```
db.persons.find({"firstname": "Tim"})
```

Diese Suchabfrage findet die Person Tim Meyer:

```
{ "_id" : ObjectId("561e9bbbf558714ee1029d3c"),
  "name" : "Meyer", "firstname" : "Tim", "age" : 7 }
```

Mitunter benötigt man die Angabe von Bedingungen. Auch dafür kommt JSON mit der Syntax `Attributname : Bedingung` zum Einsatz. Die Bedingung wiederum hat das Format `Vergleichsoperator : Wert`. Nachfolgend ist dies für das Attribut `age` und die Bedingung größer als 25 gezeigt:

```
db.persons.find({"age": {$gt : 25}}) // $gt = greater than
```

Beim Formulieren der Bedingung erkennen wir Folgendes: Für Abfragen dürfen in JSON einige Zeichen nicht genutzt werden, sondern müssen ersetzt werden. Deshalb wird die Altersabfrage statt mit > durch `$gt` ausgedrückt. Diese Suchabfrage schließt die Person Tim Meyer aufgrund des zu geringen Alters aus und liefert als Ergebnis – wie erwartet – lediglich den Datensatz von Hugo Test:

```
{ "_id" : ObjectId("561e9b62f558714ee1029d3b"),
  "name" : "Test", "firstname" : "Hugo", "age" : 33 }
```

Eine Auswahl gebräuchlicher Vergleichsoperatoren ist in Tabelle 5-3 aufgeführt.

Tabelle 5-3 *Wichtige Vergleichsoperatoren in MongoDB*

JSON	Bedeutung	Operation
`$lt: <value>`	less than	<
`$lte: <value>`	less than or equal	<=
`$gte: <value>`	greater than or equal	>=
`$eq: <value>`	equal	=
`$in: [<value1>, ..., <valueN>]`	Enthalten in Wertemenge	

Der `$in`-Operator erlaubt die Angabe eines regulären Ausdrucks, der innerhalb von Slashes folgendermaßen notiert wird: `/<Reg-Ex>/`. Eine Abfrage nach Datensätzen,

deren Attribut `name` mit `Mey` beginnt oder den Wert `TesT` enthält, geschieht wie folgt – hierbei muss man noch wissen, dass mit `/i` ein case-insensitives Matching erfolgt:

```
db.persons.find( { name: { $in: [ /^Mey/, /TesT/i ] } } )
```

Somit finden wir dann:

```
{ "_id" : ObjectId("561e9b62f558714ee1029d3b"),
  "name" : "Test", "firstname" : "Hugo", "age" : 33 }
{ "_id" : ObjectId("561e9bbbf558714ee1029d3c"),
  "name" : "Meyer", "firstname" : "Tim", "age" : 7 }
```

UPDATE

Um einige Details beim Update zeigen zu können, fügen wir der Collection `persons` noch weitere Objekte hinzu – insbesondere auch zweimal den Eintrag `Test2 UserX`:

```
db.persons.insert({"name" : "User1", "firstname" : "Test1"})
db.persons.insert({"name" : "UserX", "firstname" : "Test2"})
db.persons.insert({"name" : "UserX", "firstname" : "Test2"})
db.persons.insert({"name" : "User3", "firstname" : "Test3", "age" : 42})
db.persons.insert({"name" : "Blue", "firstname" : "Mike", "eyecolor" : "blue"})
```

Beim Einfügen ist uns (zur Demonstration der Update-Operation absichtlich) ein Tippfehler unterlaufen. Statt `UserX` sollte es `User2` heißen. Mit folgendem Kommando korrigieren wir den Tippfehler im Nachnamen:

```
db.persons.update({"firstname": "Test2"},          // Bedingung
                  {$set: {"name" : "User2"}})  // $set setzt Werte
```

Wie man an dem Kommando sieht, werden für eine Änderung zwei JSON-Dokumente benötigt: Das erste ist ein Prototyp zur Spezifikation der Suche. Dieser bestimmt die durch die Änderung betroffenen Dokumente. Der zweite Parameter legt mithilfe eines Änderungsdokuments die gewünschten Änderungen fest. Man könnte annehmen, dass im obigen Beispiel in allen Dokumenten, die im Attribut `firstname` den Wert `Test2` enthalten, das Attribut `name` auf den Wert `User2` gesetzt würde. Dem ist nicht so! Tatsächlich wird nur das erste Vorkommen modifiziert, wie es die folgende Konsolenausgabe andeutet:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Wenn alle die Suchbedingung erfüllenden Datensätze geändert werden sollen (nachfolgend: `name` auf den Wert `UPDATE_ALL`), ist die Option `{multi : true}` erforderlich:

```
db.persons.update({"firstname": "Test2"}, {$set: {"name" : "UPDATE_ALL"}},
                  { multi: true })
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })
```

Neben dem `$set`-Kommando kann man mit `$rename` bzw. `$unset` Attribute aus einem Dokument umbenennen bzw. entfernen oder mit `$inc` erhöhen.

REMOVE

Als Letztes verbleibt das Löschen von Dokumenten. Dazu gibt es das Kommando `remove()`. Diesem übergibt man eine Menge von Attributen und Wertebelegungen, die als Löschprototyp dienen. Alle Dokumente, deren Attribut-Wert-Kombination mit der übergebenen übereinstimmen, werden aus der aktuellen Collection gelöscht.

Nachfolgend wird gezeigt, wie man aus der Collection `persons` all diejenigen Dokumente löscht, die ein Attribut `name` mit dem Wert `UPDATE_ALL` besitzen:

```
db.persons.remove({"name" : "UPDATE_ALL"})
```

Es verbleiben dann noch folgende fünf Datensätze:

```
db.persons.find()
{ "_id" : ObjectId("561e9b62f558714ee1029d3b"),
  "name" : "Test", "firstname" : "Hugo", "age" : 33 }
{ "_id" : ObjectId("561e9bbbf558714ee1029d3c"),
  "name" : "Meyer", "firstname" : "Tim", "age" : 7 }
{ "_id" : ObjectId("561e9f8bf558714ee1029d3d"),
  "name" : "User1", "firstname" : "Test1" }
{ "_id" : ObjectId("561e9f8bf558714ee1029d40"),
  "name" : "User3", "firstname" : "Test3", "age" : 42 }
{ "_id" : ObjectId("561e9f8bf558714ee1029d41"),
  "name" : "Blue", "firstname" : "Mike", "eyecolor" : "blue" }
```

Nun wollen wir noch die Datensätze entfernen, die den im per `$in` angegebenen Array von Namen entsprechen. Dazu schreiben wir Folgendes:

```
db.persons.remove({"name" : {$in: ["User1", "User3"]}})
```

Hinweis: Fehlertoleranz und Merkwürdigkeiten

Beachten Sie bei der Arbeit mit MongoDB, dass diese mitunter fehlertolerant ist und statt eine Warn- oder Fehlermeldung anzuzeigen unerwartete Aktionen ausführt. Nehmen wir dazu an, wir hätten einen weiteren Datensatz wie folgt hinzugefügt:

```
db.person.insert({"name": "Mysterious", "firstname" : "Mysterious"})
```

Ermitteln wir erneut, welche Daten gespeichert sind und nutzen das Kommando:

```
db.persons.find()
```

In den aufgelisteten Daten finden wir das neu erzeugte Objekt nicht. Wo ist es? Wenn Sie sehr genau hingeschaut haben, dann sehen Sie, dass beim Einfügen als Name der Collection `person` statt `persons` angegeben wurde. Weil diese noch nicht existiert, legt MongoDB automatisch eine gleichnamige Collection an. Ein solcher kleiner Tippfehler fällt kaum auf und löst manchmal Irritationen aus. In diesen Fällen sollten Sie Ihre Kommandos genau prüfen. Ich habe so schon den einen oder anderen Tippfehler fabriziert – oft sitzt der Fehler eben vor der Tastatur ;-).