Das kleine MongoDB Buch

von Karl Seguin

aktualisiert für 2.6

Über dieses Buch

Lizenz

Über den Autor

Karl Seguin ist ein Softwareentwickler mit Erfahrung in vielen Bereichen und Technologien. Er ist ein Experte in .Net und Ruby. Des weiteren trägt er semi-aktiv zu OSS Projekten bei, ist technischer Autor und hält gelegentlich Vorträge. Bezogen auf MongoDB war er Kern Entwickler der C# MongoDB Library NoRM, schrieb das interaktive tutorial mongly und entwickelte Mongo Web Admin.

Sein kostenloser service für Spiele Entwickler, mogade.com, läuft natürlich auf MongoDB.

Karl hat auch das The Little Redis Book geschrieben

Hier findet ihr seinen blog: http://openmymind.net, und seine tweets via [@karlseguin](http://twitter.com/karlseguin)

Mit besonderem Dank an / With Thanks To

Ein besonderes Dankeschön geht an Perry Neal Der mir seine Augen, seinen Verstand und seine Leidenschaft geliehen hat. Deine Hilfe ist unbezahlbar. Ich danke dir.

Aktuelle Version

Diese Version wurde von Asya Kamsky für MongoDB 2.6 aktualisiert.

The aktuellen Original-Quellen dieses Buches findet ihr unter:

http://github.com/karlseguin/the-little-mongodb-book.

Die deutsche Übersetzung

Die deutsche Übersetzung wurde erstellt von Klaas-Henning Zweck (twitter: [@pythononwheels](http://twitter.com/pythononwheels)) Die aktuellen Quellen der deutschen Übersetzung des Buches findet ihr hier:

https://github.com/pythononwheels/the-little-mongodb-book.

Und hier findet ihr die jeweils aktuellen PDF, ePub und mobi Versionen.

http://pythononwheels.org/post/the_little_mongodb_book

Anmerkungen zur deutschen Übersetzung (Rechtschreibung, Sprachstil etc.) könnt ihr am besten via github issues beisteuern.

Anerkung zur Version: Das Buch behandelt die MongoDB Version 2.6 aktuell ist 3.6. Die Basics sind aber gleich geblieben und da das Buch dem Einstieg in die Arbeit mit MongoDB dient, kann man es genauso inklusive der Beispiele in neueren Versionen weiter verwenden.

Slideshare mit den Changes der MongoDB Versionen findet ihr hier

Einleitung

> It's not my fault the chapters are short, MongoDB is just easy to learn.
>
> Es ist nicht meine Schuld das die Kapitel so kurz sind, MongoDB ist eben einfach leicht zu lernen.

Es wird oft geagt, das sich die Technologie rasant entwickelt. Es ist zwar richtig, das die Liste der Technologien und Techniken immer länger wird, aber ich war immer der Meinung das die fundamentalen Methoden für Softwareentwickler sich eher langsam entwickeln. Man kann Jahre damit verbringen eine kleine, aber wichtige Technik zu erlernen. Auffällig ist jedoch, mit welcher Geschwindigkeit (heute) Technologien ersetzt werden. Scheinbar über Nacht wird etabliertes von einer veränderten Ausrichtung des Entwickler-Fokus bedroht.

Nichts könnte repräsentativer für diesen plötzlichen Wandel sein als der Erfolg der NoSQL Technologien gegen die sehr gut etablierte Basis der relationalen Datenbanken (RDBMS). Vor nicht all zu langer Zeit basierte das Netz noch auf ein paar RDBMS und plötzlich hatten sich fünf oder sechs NoSQL Systeme als eine echte Alternative etabliert.

Auch wenn diese Veränderungen scheinbar über Nacht kamen, benötigen sie in Wahrheit manchmal Jahre um sich in der Breite zu etablieren. Der anfängliche Enthusiasmus bezieht sich auf eine relativ kleine Gruppe von Entwicklern und Firmen.

Aus Fehlern wird gelernt, Lösungen werden verfeinert (refined) und wenn andere sehen das sich eine neue Technologie durchsetzt, dann fangen sie langsam an diese Technologie auch einzusetzen. Noch einmal, das was ich eben gesagt habe trifft insbesondere für NoSQL DBs zu. Viele NoSQL Lösungsansätze sind eben kein neuer Ersatz für bestehende Probleme sondern adressieren spezifische Bedarfe die von bestehenden (SQL) Systemen nicht abgedeckt werden.

Nach dieser Einleitung sollte ich zunächst erläutern was NoSQL eigentlich bedeutet. Es ist ein weiter Begriff der von verschiedenen Menschen unterschiedlich interpretiert wird. Persönlich benutze ich den Begriff eher weit gefasst für Systeme im Bereich der Datenspeicherung. Anders gesagt bedeutet NoSQL (für mich) der Überzeugung zu sein, das Datenpersistenz nicht die Verantwortung eines einzelenen Systems ist.

Hersteller relationaler Datenbankmanagementsysteme versuchen seit langer Zeit ihre Systeme als eine "One size fits all" Lösung zu positionieren. NoSQL Systeme hingegen gehen eher in Richtung kleinere Einheiten der Verantwortlichkeit um das beste tool für den jeweiligen Zweck wirksam einsetzen zu können. Also kann es durchaus so sein, das ihr NoSQL Anwendungs-Stack auch relationale Datenbanken, wie zum Beispiel MySQL beeinhaltet, sie aber eben auch z.B. Redis zur schnellen Datensuche und Hadoop für intensive Datenverarbeitung nutzen. Kurz gesagt geht es beim Thema NoSQL um Offenheit und Bewusstsein für alternative, existierende und neue Muster und Tools um Daten zu verwalten.

Jetzt fragen sie sich sicherlich wie nun MongoDB in all dies hineinpasst. Als dokumentenorientierte Datenbank (DB) ist MongoDB ein sehr breit einsetzbares (generalized) NoSQL System. Es sollte als eine Alternative zu relationalen Datenbanksystemen (RDB) betrachtet werden. Wie auch bei RDB Systemen kann MongoDB davon profitieren, wenn man es mit anderen, NoSQL Systemen für spezielle Bereiche kombiniert (Anm d. Übers.: Beispielsweise Redis für schnelle Suchen der Top 1000 Artikel o.ä.). MongoDB selbst hat natürlich Vor- und Nachteile, auf die in späteren Kapiteln des Buches eingegangen wird.

Einstieg / Getting started

Der grösste Teil des Buches befasst sich mit den Kernfunktionen von MongoDB. Deshalb werden wir sehr viel mit der MongoDB shell arbeiten. Die shell ist sowohl ein gutes Tool zum lernen als auch zur Administration. Zum Ausführen ihrer Programme benötigen sie jedoch auch einen MongoDB Treiber.

Das bringt uns zur ersten Sache die sie über MongoDB lernen sollten: die MongoDB Treiber. MongoDB hat eine ganze Reihe von offiziellen Treibern für verschiedene Programmiersprachen. Diese Treiber sind vergleichbar mit den Datenbanktreibern die sie wahrscheinliche schon von anderen Datenbank Systemen kennen. Auf der Basis dieser Treiber hat die Entwicklergemeinde speziellere libraries und Frameworks für verschieden Sprachen implementiert. Beispiele sind NoRM, eine C# library die LINQ implementiert, sowie MongoMapper, eine Active-Record freundliche Ruby library. Ob sie ihre Programme mit den Basistreiber oder mit Hilfe einer höheren Library entwicklen liegt alleine bei ihnen. Ich möchte dies hier einmal klarstellen, da manche Leute die mit MongoDB anfangen, durch den Umstand verwirrt werden, das es offizielle Treiber und Community libraries gibt. Erstere konzentrieren sich mehr auf Konnektivität und Kommunikaiton mit MongoDB während die Community libraries sich mehr auf die Programmiersprachen und Framework spezifischen Aspekte beziehen.

Während sie dieses Buch lesen möchte ich sie ermutigen auch immer praktisch auszuprobieren was ich vorstelle und auch Fragen die sie haben selbst durch ausprobierne zu klären. Es ist sehr leicht mit MongoDb zu starten, also lassen sie uns ein paar Minuten nehmen um die Umgebung aufzusetzen.

1. Gehen sie auf die offizielle download Seite und laden sie das MongoDB release (Ich empfehle die "stable version") für ihr Betriebssystem. Für Entwicklungszwecke können sie sowohl die 32-Bit als auch die 64-Bit Version nehmen.

2. Packen sie die Archivdatei aus (in welchem Verzeichnis sie wollen) und wechseln sie in das Unterverzeichnis bin. Starten sie jetzt noch nichts, zur ihrer Information sage ich ihnen aber schonmal das es sich bei mongod`` um den Server Prozeß und beimongo`um die client shell handelt. Dies sind die beiden Programme mit denen wir die meiste Zeit arbeiten werden.

3. Erstellen sie eine Textdatei Namens mongodb.config im Unterverzeichnis bin.

4. Fügen sie die folgende Zeile zur Datei mongodb.config hinzu: dbpath=PATH_TO_WHERE_YOU_WANT_TO_STORE_YOUR_DATABASE . Achten sie darauf das die Pfadangabe für ihre Betriebssystem richtig formuliert ist. Für Windows könnte der Eintrag Beispielsweise so aussehen dbpath=c:\mongodb\data und für Linux z.B. so dbpath=/var/lib/mongodb/data.

5. Stellen sie sicher das der angegebene dbpathauch existiert. Oder erstellen sie ihn jetzt.

6. Starten sie jetzt mongod mit dem Parameter --config /path/to/your/mongodb.config.

Hier ein komplettes Beispiel für Windows Anwender. Wenn sie das Archiv in das Verzeichnis c:\mongodb\ entpackt haben und sie haben als Datenverzeichnis c:\mongodb\data\ angelegt, dann müssen sie das wie folgt in der Datei c:\mongodb\bin\mongodb.config spezifizieren: dbpath=c:\mongodb\data\. Sie starten mongod dann von der Kommandozeile mit: c:\mongodb\bin\mongod --config c:\mongodb\bin\mongodb.config.

Sie können das bin Verzeichnnis natürlich auch zu ihrer PATH Umgebungsvariable hinzufügen um die Aufrufe kürzer (less verbose) zu machen. Für MacOSX und Linux Anwender funktioniert das im wesentlichen genauso, sie müssen nur die Pfade entsprechend anpassen.

Ich hoffe sie haben MongoDB nun gestartet. Sollten sie eine Fehlermeldung sehen, lesen sie diese bitte genau durch. Der Server ist wirlklich gut darin zu melden was schief gelaufen ist.

Sie können jetzt mongo starten (ohne das *d*) um eine shell mit dem Server zu verbinden. Geben sie als Test db.version() um sicher zu stellen das alles funktioniert. Sie sollten als Ausgabe die Versionsnummer (von MongoDB) sehen die sie installiert haben.

Kapitel 1 - Die Grundlagen

Wir beginnen unsere Reise mit den Grundlegenden Mechanismen der Arbeit mit MongoDB. Natürlich sind dies die Kernthemen um MongoDB zu verstehen aber es sollte und auch einen guten Eindruck vermitteln wo MongoDB einsetzbar ist.

Es gibt sechs einfache Konzepte die wir verstehen müssen.

1. MongoDB hat das selbe Konzept einer Datenbank das sie vermutlich schon kennen (oder das eines Schemas für die Oracle Leute). In jeder MongoDB Instanz kann es Null oder mehr Datenbanken geben, von denen jede als übergeordnerter Container für weitere Inhalte fungiert.

2. Jede Datenbank hat Null oder mehr collections. Eine collection hat so viele gemeinsamkeiten mit einer klassischen Tabelle, das man sich die beiden als identisch vorstellen kann.

3. collections bestehen aus Null oder mehr Dokumenten. Sie können sich ein Dokument als eine Zeile einer Tabelle vorstellen.

4. Jedes Dokument besteht aus einem oder mehr Feldern, die sie sich wie Spalten einer Tabelle vorstellen können.

5. Indexe funktionieren in MongoDB grösstenteils so wie in den entsprechenden RDBMS.

6. Cursors sind anders als die vorherigen fünf Konzepte. Sie werden oftmals übersehen, sind aber meines Erachtens nach wichtig genug, sie gesondert zu besprechen. Es ist wichtig zu verstehen, das wann immer sie in MongoDB nach Daten suchen, sie einen Zeiger auf das Ergebnis als Antowrt erhalten. Dieser Zeiger wird cursor genannt. Mit diesem cursor können sie verschiedene Operationen, wie Beispielsweise zählen oder überspringen (skipping ahead) ausführen bevor sie die eigentlichen Daten übertragen.

Zur Wiederholung: MongoDB besteht aus Datenbanken die collections enthalten. Jede collection enthält Dokumente, die wiederum aus Feldern bestehen. Sie können collections indexieren um die Such- und Sortiergeschwindigkeit zu erhöhen. Und am Ende erhalten wir die Suchergebnisse in Form eines cursors dessen eigentliche Ausführung so lange verzögert wird bis sie wirklich benötigt wird.

Warum benutzen wir hier neue Begriffe (collection vs. Tabelle, Dokument vs. Zeile, Feld vs. Spalte)? Machen wir das nur um die Dinge zu komplizieren ? In Wahrheit sind die (MongoDB) Konzepte zwar ähnlich der Konzepte in der Welt der raltionalen Datenbanken aber sie sind eben doch nicht genau gleich. Der haupt Unterschied liegt darin, das relationale Datenbanken ihre Spalten auf Tabellen Ebene definieren wohingegen dokumenten-orientierte Datenbanken ihre Felder auf der Dokumenten Ebene definieren. Das bedeutet das jedes Dokument einer collection seine eigenen Felder haben kann. (Die sich von anderen Dokumenten in der selben collection unterscheiden können). Einen collection ist also ein eher *dummer* container im Vergleich zu einer Tabelle, während in einem Dokument deutlich mehr Informationen als in einer Zeile stecken.

Auch wenn es wichtig ist die Unterschiede zu kennen, müssen sie nicht verzweifeln wenn ihnen jetzt noch nicht alles völlig klar ist. Sie werden nicht mehr als eine Handvoll "inserts" brauchen um zu verstgehen was damit wirklich gemeint ist. Letztendlich ist der Hautpaspekt, das eine collection nicht streng vorgiebt, was in ihr gespeichert wird (Eine collection ist schema-los). #todo Felder werden für jedes Dokument individuell verfolgt (tracked). Die Vor- und Nachteile dieses Konzeptes werden wir in einem folgenden Kapitel erforschen.

Lassen sie uns loslegen. Falls sie es noch nicht getan haben, starten sie jetzt den mongod server und die mongo shell. Die shell führt Javascript aus. Zusäzlich gibt es auch einige globale Kommandos, wie help oder exit die sie ausführen können. Kommandos die sie auf die aktuelle Datenbank anwenden, wie db.help() oder db.stats() werden auf das db Objekt angewendet (executed against). Kommandos die sie auf eine Collection beziehen und diese werden wir häufig verwenden, werden auf das db.COLLECTION_NAME Object angewendet, so wie z.B.: db.unicorns.help() oder db.unicorns.count().

Geben sie nun db.help() ein. Als Ausgabe erhalten sie eine Liste aller Kommandos die sie auf das db Objekt anwenden können.

Kleine Notiz am Rande. Da es sich bei der mongo shell um eine Javascript shell handelt, sehen sie, wenn sie bei einem Kommando die Klammern () weglassen, den Quellcode der Methode anstatt diese auszuführen. Ich möchte das nur deshalb hier anmerken, damit sie nicht überrascht sind, wenn ihnen das das erste Mal passiert und sie eine Ausgabe sehen die mit **function** (…) { anfängt. Wenn sie jetzt Beispielsweise db.help, also ohne die Klammern, eingeben, sehen sie die interne Implementierung der help Mehtode.

Als erstes werden wir jetzt die globale Hilfsfunktion use benutzen um zwischen verschiedenen Datenbanken zu wechseln. Geben sie jetzt bitte use learn ein. Es spielt dabei keine Rolle, das die Datenbank learn noch nicht existiert. Wenn wir die erste Collection anlegen, wird automatisch auch die Datenbank learn erzeugt. Da sie sich jetzt im Scope einer Datenbank bewegen können sie in der shell Datenbank Kommandos, wie db.getCollectionNames() ausführen. Wenn sie dieses Kommando ausführen sollten sie als Ausgabe ein leeres Array sehen ([ ]). #todo Hier fehlt eigentlich noch ein Wort der Erklärung warum das so ist (Weil eben keine Collection existieren) Da Collections Schema-los sind müssen wir sie nicht explizit erzeugen. Es reicht einfach ein Dokument in einer Collection anzulegen. Um das zu tun benutzen sie das insert Kommando und übergeben als Parameter das anzulegende Dokument:

```
db.unicorns.insert({name: 'Aurora',
    gender: 'f', weight: 450})
```

Das oben aufgeführte Kommando führt ein insert auf der unicorns collection aus und übergibt ihm genau einen Parameter (nämlich das anzulegende Dokument als JSON). Intern verwendet MongoDB ein BSON genanntes, serialisiertes, binäres JSON Format.
Extern, also für uns, bedeutet das, das wir sehr häufig JSON verwenden werden, wie zum Beispiel bei den Methoden Parametern. Wenn sie jetzt db.getCollectionNames() noch einmal ausführen, sehen sie zwei collections, nämlich: unicorns und system.indexes. Die collection system.indexes wird einmal pro Datenbank angelegt und enthält Informationen über die Indexe dieser Datenbank.

Sie könnne nun das find Kommando auf die unicorns collection anwenden um eine Liste der Dokumente zu erhalten:

```
db.unicorns.find()
```

Es fällt ihnen vielleicht auf das zusätzlich zu den Attributen die sie definiert haben (name, gender, weight) noch ein _id Feld existiert. Jedes Dokument in MongoDB muss ein eindeutiges _id Feld besitzen. Sie können ein solches Feld selbst definieren, wenn nicht legt MongoDB automatisch ein solches
Feld für sie an. In diesem Fall hat das _id Feld dann den Datentyp ObjectId. Meistens werden sie MongoDB das Feld für sich anlegen lassen. Das _id Feld wird auch automatisch indexiert, was auch erklärt warum die collection system.indexes angelegt wurde. Sie können sich den Inhalt der collection nun einmal ansehen:

```
db.system.indexes.find()
```

Als Ausgabe sehen sie den Namen des Index sowie die Datenbank und Collection für die er angelegt wurde. Zusätzlich sehen sie auch noch die Felder die indexiert werden.

Nun aber zurück zu unserer Diskussion über schema-lose collections. Fügen sie nun ein völlig anderes Dokument in die unicorns collection ein. Zum Beispiel:

```
db.unicorns.insert({name: 'Leto',
    gender: 'm',
    home: 'Arrakeen',
    worm: false})
```

Benutzen sie nun noch einmal find um alle Dokumente aufzulisten. Wenn wir etwas mehr gelernt haben, werden wir dieses interessante Verhalten von MongoDB im Detail beleuchten aber ich hoffe das sie jetzt schon sehen warum die traditionelle Terminologie hier nicht so gut passt.

Mastering Selectors / Selektoren meistern

Zusätzlich zu den sechs Konzepten die wir untersucht haben (#todo kommt mir irgendwie weniger vor) gibt es einen Aspekt für den sie ein gutes Verständnis haben sollten, bevor wir uns weiterführenden Themen zuwenden: Abfrage Selektoren (query selectors). Ein MongoDB query selector ist mit dem where (clause) eines SQL Statements zu vergleichen. Sie benutzen ihn wenn sie Dokumente finden, zählen, updaten oder aus einer collection entfernen wollen. Ein selector ist ein JSON Objekt. Im einfachsten Fall entspricht er {} was auf alle Dokumente einer collection zutirfft. Wenn wir Beispielsweise alle weiblichen unicorns finden wollen, müssten wir {gender:'f'} angeben.

Bevor wir zu tief in Selektoren abtauchen lassen sie uns einige Daten anlegen mit denen wir spielen können. Als erstes löschen sie bitte alles was wir bisher in der unicorns collection angelegt haben mit:
db.unicorns.remove({}). Führen sie jetzt bitte die folgenden inserts aus um unsere Spieldaten anzulegen (Ich schlage vor das sie das mit Copy&Paste machen):

```
db.unicorns.insert({name: 'Horny',
    dob: new Date(1992,2,13,7,47),
    loves: ['carrot','papaya'],
    weight: 600,
    gender: 'm',
    vampires: 63});
db.unicorns.insert({name: 'Aurora',
    dob: new Date(1991, 0, 24, 13, 0),
    loves: ['carrot', 'grape'],
    weight: 450,
    gender: 'f',
    vampires: 43});
db.unicorns.insert({name: 'Unicrom',
    dob: new Date(1973, 1, 9, 22, 10),
    loves: ['energon', 'redbull'],
    weight: 984,
```

```
    gender: 'm',
    vampires: 182});
db.unicorns.insert({name: 'Roooooodles',
    dob: new Date(1979, 7, 18, 18, 44),
    loves: ['apple'],
    weight: 575,
    gender: 'm',
    vampires: 99});
db.unicorns.insert({name: 'Solnara',
    dob: new Date(1985, 6, 4, 2, 1),
    loves:['apple', 'carrot',
        'chocolate'],
    weight:550,
    gender:'f',
    vampires:80});
db.unicorns.insert({name:'Ayna',
    dob: new Date(1998, 2, 7, 8, 30),
    loves: ['strawberry', 'lemon'],
    weight: 733,
    gender: 'f',
    vampires: 40});
db.unicorns.insert({name:'Kenny',
    dob: new Date(1997, 6, 1, 10, 42),
    loves: ['grape', 'lemon'],
    weight: 690,
    gender: 'm',
    vampires: 39});
db.unicorns.insert({name: 'Raleigh',
    dob: new Date(2005, 4, 3, 0, 57),
    loves: ['apple', 'sugar'],
    weight: 421,
    gender: 'm',
    vampires: 2});
db.unicorns.insert({name: 'Leia',
    dob: new Date(2001, 9, 8, 14, 53),
    loves: ['apple', 'watermelon'],
    weight: 601,
    gender: 'f',
    vampires: 33});
db.unicorns.insert({name: 'Pilot',
    dob: new Date(1997, 2, 1, 5, 3),
    loves: ['apple', 'watermelon'],
    weight: 650,
```

```
    gender: 'm',
    vampires: 54});
db.unicorns.insert({name: 'Nimue',
    dob: new Date(1999, 11, 20, 16, 15),
    loves: ['grape', 'carrot'],
    weight: 540,
    gender: 'f'});
db.unicorns.insert({name: 'Dunx',
    dob: new Date(1976, 6, 18, 18, 18),
    loves: ['grape', 'watermelon'],
    weight: 704,
    gender: 'm',
    vampires: 165});
```

Jetzt wo wir endlich Daten haben, können wir auch die Selektoren meistern. Der Selektor {field : value} findet (matched) alle Dokumente bei denen field den Wert value hat. Mit {field1 : value1, field2 : value2} können sie eine UND Verknüpfung abbilden. Die speziellen Operartoren: $lt, $lte, $gt, $gte and $ne enstprechen less than (<), less than or equal (<=), greater than (>), greater than or equal (>=) and not equal (!=). Um Beispielweise alle männlchen unicorns zu finden die mehr als 700 Pfund wiegen müssten wir folgenden Abfrage ausführen:

```
db.unicorns.find({gender: 'm',
    weight: {$gt: 700}})
//or (not quite the same thing, but for
//demonstration purposes)
db.unicorns.find({gender: {$ne: 'f'},
    weight: {$gte: 701}})
```

Der $exists Operator wird benutzt um die Existenz eines Feldes zu prüfen. Die folgende Abfrage sollte genau ein Dokument zurückliefern:

```
db.unicorns.find({
    vampires: {$exists: false}})
```

Der $in Operator wird benutzt um eines von mehreren Elementen eines Arrays zu treffen (match). Die folgende Abfrage liefert alle unicorns bei denen loves entweder apple oder orange entspricht.

```
db.unicorns.find({
    loves: {$in:['apple','orange']}})
```

Wenn sie eine ODER Verknüpfung anstelle einer UND Verknüpfung auf verschiedene Felder anwenden wollen, so können sie den $or Operator auf ein Array von Selektoren anwenden. (Das ist einfacher als es klingt):

```
db.unicorns.find({gender: 'f',
    $or: [{loves: 'apple'},
          {weight: {$lt: 500}}]})
```

Die obrige Abfrage liefert alle weiblichen (gender: 'f') unicorns die entweder apple lieben oder weniger als 500 Pfund wiegen.

Es gibt da ein nettes kleines Detail in den letzten beiden Beispielen das ihnen vielleicht sogar schon aufgefallen ist. Das loves Feld ist ein Array und MongoDB unterstützt Arrays als Objekte erster Klasse (first class Objects). Das ist ein unglaublich nützliches feature. Wenn sie es erstmal vwerwenden werden sie sich schnell fragen wie sie jemals ohne ausgekommen sind. Was vielleicht noch interessanter ist, wie leicht sie eine Abfrage auf einem Array machen können: {loves: 'watermelon'} gitb alle Dokumente zurück bei denen der Wert watermelon im loves Array des Dokumentes vorkommt.

Es gibt in M ongoDB noch mehr Operatoren als wir bisher betrachtet haben. Diese werden ausführlich im Query Selectors](http://docs.mongodb.org/manual/reference/operator/query/#query-selectors) Kapitel des MongoDB Handbuches beschrieben. Was wir bisher behandelt haben sind die Grundlagen die sie für den Start benötigen. Im Grunde ist es aber auch das, was sie später meistens benötgen werden.

Wir haben gesehen wie sie Selektoren mit dem find Kommando verwenden können. Sie könnne sie auch auf das remove Kommando anwenden, was wir ein bisschen betrachtet haben. Des weiteren könnne sie sie auch auf das count Kommando anwenden, was wir bisher nicht angesehn haben sowie auf das update Kommando, mit dem wir später noch einige Zeit verbringen werden.

Die ObjectId die MongoDB für uns angelegt hat, kann wie folgt selektiert (selected) werden:

```
db.unicorns.find(
    {_id: ObjectId("TheObjectId")})
```

in diesem Kapitel / In This Chapter

Wir haben bisher weder auf update geschaut noch auf einige der ausgefalleneren (fancier) Sachen die man mit find machen kann. Aber wir haben MongoDB gestartet und haben uns kurz mit den insert und remove Kommandos beschäftigt (Bei denen es aber auch nicht viel mehr zu sehen gibt). Wir haben auch find kennen gelernt und haben gesehen was es mit MongoDB Selektoren auf sich hat. Wir hatten einen guten Start und haben eine gute Grundlage für das gelegt was noch kommt. Ob sie es glauben oder nicht, sie wissen tatsächlich beinahe schon alles was sie wissen müssen um mit MongoDB zu starten - Es ist wirklich schnell zu lernen und einfach zu benutzen. Ich möchte sie an dieser Stelle ermuntern noch ein wenig mit MongoDB zu spielen bevor sie weiterlesen. Fügen sie verschiedenen Dokumente, vielleicht in unterschiedliche collections, ein und werden sie ein bisschen vertrauter mit verschiedenen Selektoren. Benutzen sie einfach find, count und remove. Nach ein paar eigenen Versuchen wird sich viel unklares von selbst (er)klären.

Kapitel 2 - Updating / Chapter 2 - Updating

In Kapitel 1 haben wir drei der vier CRUD (create, read, update and delete) Operationen eingeführt. Dieses Kapitel ist der bisher ausgelassenen update Operation gewidmet. #todo Update hat einige überraschende Funktionen (behaviors) die es Wert machen der Operation ein eigenes Kapitel zu widmen.

Update: Replace Versus $set

In seiner einfachsten Form werden update zwei Parameter übergeben: der Selektor (where) worauf update anzuwenden ist und welche updates auf Felder erfolgen sollen. Wenn also Roooooodles ein bisschen zugenommen hat, sollten wir vielleicht folgendes update anwenden:

```
db.unicorns.update({name: 'Roooooodles'},
    {weight: 590})
```

(Wenn sie schon etwas mit ihrer unicorn collection herumgespielt haben und diese nicht mehr dem originale Datenbestand entspricht, dann können sie sie jetzt einfach löschen und mit den Daten aus Kapitel 1 per Cut&Paste wieder in den Originalzustand bringen)

Wenn wir uns jetzt den upgedateten Datensatz ansehen:

```
db.unicorns.find({name: 'Roooooodles'})
```

Dann sollten sie auch die erste Überraschung bei update erleben. #todo Die Abfrage liefert kein Dokument zurück, da der zweite Parameter den wir übergeben haben keinen Update Operator hat, weshalb das Dokument in diesem Fall **ersetzt** (**replaced**) wurde. Anders gesagt, das update hat ein Dolukent über das Feld name gefunden und dann das gesamte Dokument durch das neue Dokument (eben den zweiten Parameter) ersetzt. Es gibt in SQL keine vergleichbare Funktion des update Kommandos. In einigen Situationen ist dieses Verhalten ideal um einige seht dynamische updates zu ermöglichen. Wenn sie hingegen den Wert eines oder mehrere Felder ändern wollen, dann müssen sie MongoDBs $set` Operator verwenden. Führen sie nun das folgende Kommando aus um die verlorenen Felder wieder her zu stellen:

```
db.unicorns.update({weight: 590}, {$set: {
    name: 'Roooooodles',
    dob: new Date(1979, 7, 18, 18, 44),
    loves: ['apple'],
    gender: 'm',
    vampires: 99}})
```

Dieses Kommando überschreibt das Feld weight nicht, da wir es nicht expizit angegeben haben. Wenn sie jetzt folgendes ausführen:

```
db.unicorns.find({name: 'Roooooodles'})
```

Erhalten wir auch das erwartete Ergebnis. Daher ist der richtige Weg um ein update auf dem Feld weight durchzuführen:

```
db.unicorns.update({name: 'Roooooodles'},
    {$set: {weight: 590}})
```

Update Operatoren / Update Operators

Zusätzlich zu $set können wir auch noch mit anderen Kommandos einige nette Dinge machen. update Operatoen arbeiten auf Feldern - so das nicht ihr gesamtes Dokument gelöscht wird. (#todo hier muss man update eigentlich anders nennen). Den $inc Operator können sie Beispielsweise verwenden um ein Feld um einen positiven oder negativen zu erhöhen. Wenn wir Pilot zum Beispiel fälschlicherweise einige Vampirtötungen angerechnet haben, können wir das mit folgendem Kommando berichtigen:

```
db.unicorns.update({name: 'Pilot'},
    {$inc: {vampires: -2}})
```

Wenn Aurora plötzlich ein failble für Süsses entwickelt, könnte wir mittels des $push Operators einen neuen Wert zu ihre loves Feld hinzufügen:

```
db.unicorns.update({name: 'Aurora'},
    {$push: {loves: 'sugar'}})
```

Mehr Informationen über die verfügbaren Update Operatoren finden sie im [Update Operators] (http://docs.mongodb.org/manual/referenc
operators) Kapitel den MongoDB Handbuchs.

Upserts

Eine der angenehmsten Überraschungen bei der Verwendung vonupdate ist, das es upserts voll unterstützt. Ein upsert führt ein update auf einem Dokument durch, wenn es ein entsprechendes Dokument findet und ein insert wenn nicht. In bestimmten Situationen ist manchmal sehr nützlich upserts zu haben, sie werden es merken wenn solch ein Fall eintritt. Um upserts zu aktivieren übergeben sie update den dritten Paramter {upsert:**true**}.

Ein einfaches Beispiel ist ein Aufrufzähler (hitcounter) auf einer Website. Wenn wir einen aufsummierten Echtzeit hitcounter haben wollem, müssten wir prüfen ob es bereits einen solchen Datensatz gibt und dann Entscheiden ob wir einen neuen anlegen oder den bestehenden erhöhen. Ohne die upsert Option (oder wenn diese als false deklariert wird), würde das folgende Statement nichts verändern:

```
db.hits.update({page: 'unicorns'},
    {$inc: {hits: 1}});
db.hits.find();
```

Wenn wir allerdings die upsert Option aktivieren sehen die Ergebnisse ziemlich anders aus:

```
db.hits.update({page: 'unicorns'},
    {$inc: {hits: 1}}, {upsert:true});
db.hits.find();
```

Da kein Dokument existiert bei dem das Feld page den Wert unicorns hat, wird ein neues Dokument (in die Collection) eingefügt. Wenn wir das obige Kommando noch einmal audführen, wird das (jetzt) existierende Dokument upgedated und hits wird auf 2 erhöht.

```
db.hits.update({page: 'unicorns'},
    {$inc: {hits: 1}}, {upsert:true});
db.hits.find();
```

Mehrfach Updates / Multiple Updates

Die letzte Überraschung die update zu bieten hat ist, das immer nur ein Dokument updedated wird. Das erscheint ziemlich logisch für die Beispiele die wir bisher angesehen haben. Wenn sie aber Beispielsweise folgendes Statement ausführen:

```
db.unicorns.update({},
    {$set: {vaccinated: true }});
db.unicorns.find({vaccinated: true});
```

Würden sie wahrscheinlich erwarten das alle unicorns geimpft (vaccinated) sind. Um das (erwartete) Verhalten zu erreichen müssen sie allerdings die multi Option auf True setzen:

```
db.unicorns.update({},
    {$set: {vaccinated: true }},
    {multi:true});
db.unicorns.find({vaccinated: true});
```

In desem Kapitel / In This Chapter

Mit diesem Kapitel beenden wir die Einführung in die einfachen CRUD Operationen auf collections. Wir haben uns update im Detail angeschaut und drei interessante Verhaltensweisen festgestellt. Erstens: wenn sie ein Dokument ohne update Operatoren übergeben, wird das gesamte Dokument ersetzt. (MongoDB's update will replace the existing document). Deshalb werden sie normalerweise den $set Operator verwenden (oder einen der vielen anderen Operatoren die das Dokument verändern (und nicht ersetzen)). Zweitens bietet update eine intuitive upsert Option die besonders hilfreich ist, wenn sie nicht genau wissen ob das Dokument bereits exisitiert. Drittens und letztens wird durch update nur das erste gefundene Dokument geändert. Wenn sie alle Dokumente auf einmal ändern wollen müssen sie die multi Option verwenden.

Kapitel 3 - Mastering Find

In Kapitel eins haben wir einen Überblick über das find Kommando gegeben. Es gibt aber noch mehr im Bezug auf find als Selektoren. Wir haben bereits erwähnt, das das Ergebnis von find ein cursor ist. Wir werden jetzt einmal genauer betrachten was das im Detail bedeutet.

Feld Selektion / Field Selection

Bevor wir uns weiter mit cursorn beschäftigen sollten sie wissen das find einen optionalen zweiten Parameter Namens projection hat. Dieser Parameter enthält die Liste der Felder die wir ein- oder ausschliessen wollen. Wenn sie beispielsweise nur die Namen der unicorns, ohne die anderen Felder, erhalten wollen, führen sie folgendes Kommandfo aus:

```
db.unicorns.find({}, {name: 1});
```

Das _id Feld wird standardmässig immer elas Ergebnis zurückgeliefert. Sie können es explizit mit folgender Anweisung ausschliessen: {name:1, _id: 0}.

Sie können Inklusion und Exklusion in der selben Anweiseung nur beim _id Feld mischen. Bei anderen Feldern ist dies nicht möglich. Wenn man darüber einmal nachdenk ergibt dies auch Sinn. Man möchte Felder entweder explizit selektieren (inlcude) oder ausschliessen (exclude).

Ordnen / Ordering

Ich habe bereits ein paar Mal erwähnt das find einen cursor zurückliefert, dessen Ausführung so lange verzögert wird, bis sie wirklich benötigt wird. Während der Benutzung der Shell werden sie hingegen bemerkt haben, das find direkt (also nicht verzögert) ausgeführt wird. Dieses Verhalten ist allerdings spezifisch für die Shell.
Um das echte Verhalten der cursor zu untersuchen können wir uns einige der Methoden ansehen, die sich mit find verketten (chained to find) lassen. Die erste dieser Methoden die wir untersuchen wollen ist sort. Wir spezifizieren die Felder nach denen wir sortieren wollen als JSON Dokument, wobei wie 1 für aufsteigende- und -1 für absteigende Sortierung angeben. Zum Beispiel:

```
//heaviest unicorns first
db.unicorns.find().sort({weight: -1})

//by unicorn name then vampire kills:
db.unicorns.find().sort({name: 1,
    vampires: -1})
```

Wie bei relationalen Datenbanken kann MongoDB auch einen Index zur (A.d.Ü.: Optimierung der) Sortierung verwenden. Wir werden Indexe später noch genauer betrachten. Sie sollten allerdings wissen, das MongoDB die Grösse eines sort ohne Index begrenzt. Das bedeutet, das sie eine Fehlermeldung erhalten, wenn sie sehr grosse Ergebnismengen sortieren wollen, ohne einen Index zu verwenden. Manche sehen das als Limitierung an, ich würde es jedoch ehrlicherweise begrüssen, wenn noch mehr Datenbanken nicht optimierte queries unterbinden würden. (Ich werde jetzt nicht

jeden MongoDB Nachteil ins positive verkehren aber ich habe genug armselig optimierte Datenbanken gesehen bei denen ich mir gewünscht hätte das sie einen stricten Modus gehabt hätten.)

Blättern / Paging

Paging von Ergebnissen kann mit den limit und skip cursor Methoden erreicht werden. Um also das zweit- und drittschwerste unicorn zu finden könnten wir folgendes ausführen:

```
db.unicorns.find()
    .sort({weight: -1})
    .limit(2)
    .skip(1)
```

limit zusammen mit sort zu verwenden kann ein guter Ansatz sein um Probleme beim sortieren von Feldern ohne Index zu vermeiden.

Zählen / Count

In der shell ist es möglich count direkt auf einer collection auszuführen.

```
db.unicorns.count({vampires: {$gt: 50}})
```

Tatsächlich ist count eigentlich eine cursor Methode, für die die shell lediglich einen shortcut bietet. Bei Treibern die diesen shortcut nicht anbieten muss der Aufruf folgendermassen durchgeführt werden (das funktioniert natürlich aus genauso in der shell):

```
db.unicorns.find({vampires: {$gt: 50}})
    .count()
```

In diesem Kapitel/ In This Chapter

Die Anwendung von find und cursors ist ein ziemlich gradliniger Ansatz. Es gibt noch ein paar weitere Kommandos die wir entweder später weiter behandeln oder die nur eher seltene Fälle betreffen. Sie können sich jedoch bereits in der MongoDB shell zu Hause fühlen und haben ein gutes Verständnis der MongoDB Grundlagen.

Kapitel 4 Daten Modellierung / Chapter 4 - Data Modeling

Lassen sie uns an dieser Stelle mal die Gangart wechseln und etwas abstrakter über MongoDB sprechen. Ein paar neue Begriffe und syntaktische Strukturen einzuführen ist eher trivial. Die Wahrheit ist, das die meisten von uns am besten herausfinden was funktioniert und was nicht, wenn sie wirklich mit diesen Technologien arbeiten. Wir können hier zwar viel theoretisch darüber sprechen aber letztendlich lernt man am besten wenn man das ganze auch pratisch anwendet und wirklich programmiert.

Von allen NoSQL Datenbanken sind die Dokument orientierten ('document-oriented') wharscheinlich diejenigen, die den relationalen Datenbanken am ähnlichsten sind, zumindest in der Datenmodellierung. Die exisitierenden Unterschiede sind aber wichtig.

No Joins / Keine Joins

Der fundamentalste Unterschied an den sie sich bei MongoDB gewöhnen müssen ist wahrscheinlich das Fehlen von joins. Ich weiss nicht genau warum MongoDB keinerlei join-Syntax unterstützt aber ich weiss das joins generell als nicht skalierend betrachtewt werden. Das bedeutet, wenn sie ihre Daten horizontal splitten, werde sie ihre joins im Application-Server implementieren. Ohne die Gründe weiter zu hinterfragen, Daten sind relational (stehen im Bezug zueinander) und MongoDB unsterstützt keine joins.

Sie werden also joins in ihrem Appliction code bauen müssen. Im Wesentlichen müssen wir eine zweite Suche ausführen um die relevanten Daten in einer anderen Collention zu finden. Unsere Daten enstsprechend zu definiren unterscheidet sich nicht von der Definition eines foreign-key in einer relationalen Datenbank. Lassen sie uns zu diesem Zweck mal den Fokus weg von den unicorns und hin zu den employees lenken. Das Erste was wir machen müssen ist eine employee zu erzeugen (Ich gebe hier eine explizite _id an um kohärente Beispiele zu ermöglichen) #todo: einfacher formulieren.

```
db.employees.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d730"),
    name: 'Leto'})
```

Jetzt fügen wir einige weitere employees hinzu und setzen ihren Manager auf Leto:

```
db.employees.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d731"),
    name: 'Duncan',
    manager: ObjectId(
    "4d85c7039ab0fd70a117d730")});
db.employees.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d732"),
    name: 'Moneo',
    manager: ObjectId(
    "4d85c7039ab0fd70a117d730")});
```

Ich möchte hier nochmal wiederholen das _id jeden beliegen Wert annehmen kann. Da man aber in der Praxis eigentlich immer ObjectIds verwednert machen wir das hier auch so.

Um alle employees von Leto zu finden muss man folgende Suche ausführen:

```
db.employees.find({manager: ObjectId(
    "4d85c7039ab0fd70a117d730")})
```

Daran ist nicths magisches. Im schlimmsten Fall muss man eine extra query ausführen, die allerdings indexiert ist (sie sollten immer einen Index auf ihre Hauptsuchattribute legen)

Arrays und eingebettete Dokumente / Arrays and embedded documents

MongoDB hat zwar keine joins, hat dafür aber ein paar andere Asse im Ärmel. Erinnern sie sich noch das MongoDB Arrays als first -**class** Objekte in Dokumenten unterstützt? Das ist unglaublich praktisch wenn man mit many-to-one oder many-to-many relationships zu tun hat. (Anm: und das ist ja wirklich häfig der Fall). Ein einfaches Beispiel ist der Fall das ein employee zwei Manager hat. In diesem Fall können wir die Manager einfach in einem Array speichern.

```
db.employees.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d733"),
    name: 'Siona',
    manager: [ObjectId(
    "4d85c7039ab0fd70a117d730"),
    ObjectId(
    "4d85c7039ab0fd70a117d732")] })
```

Hier ist besonders interessant, das für einigen Dokumente das Attribut Manager ein Skalar sein kan (also nur genau eine Manager enthält) während es bei anderen employees als Array abgespeichert ist. Unsere ursprüngliche Suche funktioniert (einfach so) mit beiden:

```
db.employees.find({manager: ObjectId(
    "4d85c7039ab0fd70a117d730")})
```

Sie werden recht schnell merken das Arrays viel praktischer sind als sich mit many-to-many joins herum zu schlagen.

Neben Array unterstützt MongoDB auch eingebettet Dokumente. Fügen wir doch einfach mal ein eingebettetes Dokument ein:

```
db.employees.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d734"),
    name: 'Ghanima',
    family: {mother: 'Chani',
        father: 'Paul',
        brother: ObjectId(
    "4d85c7039ab0fd70a117d730")}})
```

Eingebettet Dokumente können mittels einer dot-notation gesucht werden:

```
db.employees.find({
    'family.mother': 'Chani'})
```

Wir werden die Anwendungsfälle von eingebetteten Dokumente und deren Benutzung noch detailliert besprechen.

Wenn wir die beidern Konzepte kombinieren können wir natürlich auch mehrere Dokumente in einemen Array einbetten:

```
db.employees.insert({_id: ObjectId(
    "4d85c7039ab0fd70a117d735"),
    name: 'Chani',
    family: [ {relation:'mother',name: 'Chani'},
        {relation:'father',name: 'Paul'},
        {relation:'brother', name: 'Duncan'}]})
```

Denormalisierung / denormalisation

Eine andere Alternative zu joins ist die Daten zu denormalisieren. Früher war dieser Ansatz nur für performance-sensitiven code oder snapshots (wie zum Beispiel Audit logs) vorbehalten. Mit der wachsenden Popularität von NoSQL (DBs), von denen viele keine joins haben, wird Denormalisierung immer verbreiteter. Das bedeutet nicht das sie jede Information in jedem Dokument aktiv duplizieren müssen. Haben sie aber keine Angst davor, Daten so zu ordnen, wie sie auch zu den Dokumenten gehören.

Nehmen wir mal an sie schreiben eine Forum Applikation. Dann wäre der klassiche Weg einen speziellen user mit einem post zu verbinden eine userid in dem post zu verlinken. Mit einem solchen Ansatz kann man erst user suchen (joinnen) um zugehörige posts darzustellen. Eine Alternative wäre die userid und den name direkt im post zu speichern. Man kann sogar eingebettete Dokumente, wie user: {id: ObjectId('Something'), name: 'Leto'}, speichern. Und ja, richtig, wenn ein User seinen Namen ändert muss man alle zugehörigen Dokumente ändern (das wäre dann ein multi-update). Anmerkung des Übersetzers: Das kommt aber im Gegensatz zu Suche sehr selten vor. Es finden über 90% Suchen und nur 10% updates statt. Der Performance Gewinn ist also sehr hoch

Sich mit diesem Ansatz anzufreunden wird einigen sicher nicht leicht fallen. Und in einigen Fällen macht das auch keinen Sinn. Aber haben sie keine Angst mit diesem Ansatz zu experimentieren.

Welchen Ansatz sollten sie wählen?

Arrays von Ids können eine sinnvolle Strategie sein um mit one-to-many oder many-to-many Szenarien umzugehen. Aber meistens müssen Entwickler sich zwischen 'manuellen' Referenzen oder eingebtteten Dokumenten entscheinden.

Zunächstmal muss man wissen das einzelne Dokumente aktuell auf 16 MB Grösse beschränkt sind. Zu wissen das Dokumente überhaupt ein Grösselimit haben, auch wenn das erstmal recht grosszügig erscheint, gibt einem auch gleich eine Indikation wie Dokumente zu gebrauchen sind. Zur Zeitr scheinen die meisten Entwickler dahin zu tendieren, manuelle Referenzen für Relationen (relationships) zu verwenden. Eingebettete Dokumenten (embedded documents) werden ebenfalls häufig verwendet aber meistens für kleinere Datenmengen die man immer mit dem "Elterndokument" (parent document) erhalten möchte. Ein Besipiel wäre das speichern eines eingebetteten addresses documents direkt in jedem user document:

```
db.users.insert({name: 'leto',
    email: 'leto@dune.gov',
```

```
    addresses: [{street: "229 W. 43rd St",
                 city: "New York", state:"NY",zip:"10036"},
                {street: "555 University",
                 city: "Palo Alto", state:"CA",zip:"94107"}]})
```

Das soll nicht heissen, das sie eingebettet Dokumente unterschätzen oder nur als kleines utility betrachten sollten. Denn wenn man die Programm Objecte direkt auf die Daten abbilden kann macht es einem das leben oftmals erheblich leichter. Das muss man ansonsten oft mit joins künstlich nachbilden. Dies wird umso besser und richtiger wenn man zusätzlich bendent das man in MongoDB auch eingebettete Dokumente indexieren und direkt abfragen kann. (Anm.d.Ü.: First-class-citizens)

Wenig oder viele Collections ?

Da Collections keinem Schema entsprechen müssen wäre es absolut möglich ein System mit nur einer Colletion, mit einem Mischmasch von Dokumenten zu bauen aber das wäre eine sehr schlechte Idee. Die meisten MongoDB Systems sind ähnlich aufgebaut wie man es auch bei relationalen Datenbanken finden würde, wenn auch i.d.R. mit weniger Collentions (Anm: als Tabellen in SQLDBs). Anders gesagt: Wenn es in einer relationaled DB eine Tabelle wäre, dann ist die Chance recht hoch, das es auch eine Collention in MongoDB wäre (Ausnahmen bilde many-to-manyjoins, sowie Tabellen die nur dazu dienen one-to-manyBeziehungen mit einfachen Entities abzubilden.)

Die Betrachtung wird sogar noch interessanter, wenn man eingebettete Dokumente mit in Betracht zieht. Ein häufiges Beispiel dafür ist ein Blog. Soll man eine postsund eine comments Collection erstellen oder ist es besser, das jeder postein Array von eingebetteten commentshat? Wenn wir in diesem Fall mal das 16MB Limit ausser acht lassen (Das ganze Buch Hamlet umfasst als (ASCII) Text nur 200KB - also wie Populär muss ihr Blog erst werden um das zu sprengen?), sollten man in der Regel die beiden Collection spearat erstellen. Das ist einfach *sauberer*, expliziter und ist am Ende auch performanter. MongoDB's flexible Schemas erlauben es sogar einen gemischten Ansatz zu fahren und die ersten paar Kommentare einzubetten und die folgenden in ihrer eigenen Collection abzuspeichern. Das würde auch dem Prinzip entsprechen Daten zusammen zu halten die man in einer Abfrage direkt zurück erhalten möchte (Anm: Post + relevanteste Comments. Der Rest würde erst bei einem klick auf *…more* geladen)

Es gibt aber keine feste Regel (ausser dem 16MB limit). Spielen sie also ruhig mit verschiedenen Ansätzen und bekommen sie ein Gefühl dafür was sicht *richtig* anfühlt und was nicht.

In diesem Kapitel

Ziel dieses Kapitels war es, einige hilfreiche Richtlinien als Startpunkt für die Datenmodellierung in MongoDB zu geben. Wie sie sehen konnten unterscheidet sich die Datenmodellierung in dokumentenorientierten leicht zu der in relationalen Welt. MAn hat mehr Freiheiten aber auch eine Einschränkung (Anm: das 16MB limit) aber für neue Systeme passt das eigentlich immer sehr gut. Im Grunde ist der einzige Fehler den man machen kann der, es nicht (ausgiebig) zu probieren.

Wann sollten sie MongoDB benutzen

Eigentlich sollten sie jetzt bereits selbst ein gutes Gefühl dafür haben wo MongoDB gut in ihre Systeme passen könnte. Es gibt so viele neue und konkurrierende Storage Technologien das man leicht von der schieren Antzahl der Auswahlmöglichkeiten überwältigt werden kann.

Die wichtigste Erkenntnis für mich ist es, das man heute nicht mehr auf eine Lösung vertrauen muss um seine Daten zu verwalten (das hat im Grunde nichts mit MongoDB zu tun). Zweifellos hat eine einzige Lösung für Datenverwaltung auch ihre Vorteile und für einige Projekte (vielleicht sogar die meisten) ist die "eine" Lösung auch der sinnvollste Ansatz. Die Idee ist nicht das sie immer mehrere/verschiedene Lösungen in jedem Projekt verwenden *müssen* aber das sie es *können*. Sie alleine wissen ob der Nutzen einer neuen Lösung den Aufwand übersteigt.

Jetzt wo das klar ist, hoffe ich das sie MongoDB als generellen Lösungsansatz zur Datenverwaltung sehen. Es wurde ja bereits mehrfach erwähnt das dokumenten orientierte Datenbanken eine ganze Menge mit relationalen Ansätzen gemeinsam haben. Darum kann man ziemlich einfach sagen, statt immer darum herum zu reden, das MongoDB als direkte Alternative zu relationalen Datenbanken gesehen werden kann (Anm.d.Ü.: zumindest mit Ausnahme von Transaktionen auf DB Ebene (ACID vs eventual consistency))

Bitte beachten sie das ich MongoDB nicht als *Ersatz* für relationale Datenbanken bezeichne sondern als *Alternative*. MongoDB ist ein Tool das Dinge macht, die andere Tools auch können. Manche eben deutlich besser, manche nicht so gut. Lassen sie uns die Dinge mal etwas näher beleuchten.

Flexible Schemas

Ein oft zitierter Vorteil von dokumenten orientierten Datenbanken ist, das sie nicht zwingend ein fixes Schema voraussetzen. Das macht sie viel flexibler als traditionelle Datenbanktabellen. Ich stimme zu das flexible Schemas ein tolles Feature sind allerdings nicht aus dem Grund, den die meisten vermuten.

Viele Leute reden über schemalose Dbs als würde man plötzlich beginnen einen verrückten Mischmasch von Daten zu speichern. Es gibt zwar Domänen und Datensätze die wirklich extrem schwer in relationalen DBs abzubilden sind aber ich denke das das eher Ausnahmen sind. Schemalos ist zwar cool aber die meisten ihrer Daten sind eben trotzdem sehr strukturiert. Es ist zwar manchmal sehr nützlich (in derselben collection) unterschiedliche Strukturen zu haben, speziell wenn man neue Features einführen möchte aber auch das sind Dinge die sich mit nullable columns lösen lassen.

Für mich besteht daher der echte Vorteil von dynamischen (oder felxiblen) Schemas darin, das man sich den gesamten EInrichtungsaufwand spart und nahezu eine direkte Abbildung auf OOP hat. Das wird umso vorteilhafter, wenn sie mit einer statischen Programmiersprache arbeiten. Ich habe mit MongoDB sowohl mit C# als auch mit Ruby gearbeitet und die Unterschiede sind beachtlich. Ruby's dynamische Natur und die populäre ActiveRecord Implementierung elimieren die Objekt-relationalen Abbuildungsprobleme schon zu weiten Teilen von sich aus. Das soll jetzt nicht heissen das MongoDB und Ruby nicht zusammen passen, eher das Gegenteil ist der Fall. Aber ich denke die meisten Ruby developer werden MongoDB eher als inkrementelle Verbesserung sehen, wohingegen C# oder Java Developer eine fundamentale (positive) Veränderung erleben werden wie sie mit ihren Daten umgehen.

Versetzen sie sich doch mal in die Perspektivbe eines Treiber Entwicklers. Sie wollen ein Objekt abspeichern? Serialisieren sie es in JSON (technisch korrekter in BSON aber das ist vergleichbar) und senden sie es an MongoDB. Es bedarf keines Attribute- oder Typ-Mappings. Von diesem absolut direkte arbeiten profitieren sofort sie, der Entwickler.

Schreiben / Writes

Ein Gebiet in dem MongoDB eine spezielle Rolle spielen kann ist Logging. Es gibt zwei Aspekte in MongoDB die Schreiboperationen seht schnell machen. Erstens haben sie Option ein Write command zu senden das sofort endet ohne auf eine Bestätigung (des backends) zu warten. Zweitens können sie das Schreiverhalten je nach Daten Lebensdauer (durability) einstellen. Diese Einstellungen zusammen mit der Möglichkeit auch noch einzustellen wie viele Server ihren Schreibvorgang bestätigen müssen bevor er als erfolgreich angesehen wird, geben ihnen eine grossartige Kontrolle über Schreibperformance und Daten Lebensdauer.

Zusätzlich zu diesen Performance Faktoren können Logdaten auch sehr vom Schemalosen Ansatz profitieren und MongoDB hat sogar noch sogenannte capped collections. Bius Jetzt ware alle Collections die wir angelegt haben oder mit denen wir gearbeitet haben, ganz *normale* collentions. Eine *capped* collection legt man an, indem man im db.createCollection Kommando das flag capped setzt:

```
//limit our capped collection to 1 megabyte
db.createCollection('logs', {capped: true,
    size: 1048576})
```

Wenn unsere capped collection das 1 MB limit erreicht (size: 1048576) werden die ältesten Dokumente automatisch überschrieben (purghed).Wenn man das limit statt als Grösse auf die Anzahl der Dokumente einschränken möchten, kann man das mit dem Parameter max machen. Capped Collections haben einige interessante Eigenschaften. Beispielsweise kann man Dolkumente verändern (updaten) aber sie behalten dabei ihre ursprüngliche Grösse bei. Die Reihenfolge für Einfügeoperationen (insert) wird immer beibehalten. Man muss also nicht extra einen Zeit basierten Index aufbauen um nach Zeit zu ordnen. Man kann auf capped collections auch einfach ein tail Kommando ausführen genauso wie man es auch von Dateien unter Unix gewohnt ist. Ein `tail -f <filename>` erlaubt es also immer die neusten Einträge zu erhalten ohne eine neue Suche auszuführen.

Wenn man seine Einträge nicht auf Grösse oder Anzahl, sonden Zeitbasiert verfallen lassen möchte, kann man einen sogenannten TTL Index verwende. TTL Steht dabei für "time-to-live".

(Daten)Haltbarkeit / Durability

Vor Version 1.8 hatte MongoDB keine single-server durability. Das bedeutet, das ein Ausfall eines Servers leicht zu Datenverlust oder korrupten Daten führen konnte Siehe Auch. Die Lösung bestand bis zu dieser Verison darin, MongoDB immer in einer Multi-Server Umgebung zu betreiben (MongoDB unterstützt von Haus aus Replikation). Journaling war dann eines der Hauotfeatures, die in Version 1.8 eingeführt wurden. Seit Version 2.0 ist Journaling standardmässig eingeschaltet so das man im Falle eines Stromausfalles oder Servercrashes leicht die Daten wiederherstellen kann.

Ich erwähne durability hier nur, da in der Vergangenheit eine Menge Wirbel um MongoDBs nicht vorhandene single-server durability gemacht wurde. Wenn sie also heute noch Informationen über fehlendes Journaling finden sollten, so sind diese schlicht nicht mehr up to date.

Volltextsuche / Full Text Search

In einer der letzten Ergänzugen wurde echte Volltextsuche zu den MopngoDB features hinzugefügt. Es werden aktuell 15 Sprachen direkt unterstüztz inklusive stemming und Stopwortlisten. Mit der UNterstützung von Volltextsuche und Arrays wird man nur in seltenen Fällen nach anderen Lösungen suchen müssen.

Transactions

MongoDB selbst unterstützt keine direkten Transaktionen. Aber es gibt zwei Alternativen. Die eine ist super aber eingeschränkt in der Nutzung, die andere ist sehr flexibel aber etwas schwerfällig in der Benutzung.

Die erste (leichte, schlanke) ist die sogenannte atomare Update Operation. Die sind super so lange sich damit das Problem auch adressiernew und lösen lässt. Einige der einfacheren, wie $inc und $set, haben wir bereits kennengelernt. Es gibt auch etwas komplexere wie findAndModify mit denen man ein Dokument updaten (ode auch löschen) kann und direkt (atomar) das Ergebnis erhält.

Die zweite Möglichkeit, wenn atomare Operationen nicht ausreichend sind, ist ein klassische two-phase Commit. Ein two-phase commit ist für Transkationen, was für joins manuelles derefenzieren wäre. Es ist eine Lösung die unabhängig von der enginer ist, da sie sie im Code umsetzen. Two-phase commits sind tatsächlich auch in der relationalen Welt gebräuchlich um eine Transaktion über zwei Datenbanken zu ermöglichen. Die MongoDB Website illustriert das am typischen Beispiel eines Geldtransfers. Die Grundidee dabei ist die, das man den Transaktionsstatus mit einer atomaren Operation im zu ändernden Dokument speichert und die init-pending-commit/rollback Schritte dann manuell durchführt.

Dabei machen MongoDB's eingebettete Dokumente und das schemalose Design die Arbeit mir two-phase commits zwar leichter aber es ist immer noch, gerade am Anfang, nicht wirklich perfekt.

Datenverarbeitung / Data Processing

Vor Version 2.2 basierte die Datenverarbeitung mit MongoDB vornehmlich auf MapReduce. Mit Versiion 2.2 wurde dann ein Leistungsfähiges Feature namens: aggregation framework or pipeline eingeführt. Man benötigt MapReduce jetzt nur noch in den Fällen in denen komplexe Funktionen für Aggregationen nötig sind, die in den pipelines noch nicht unterstützt werden. Im nächsten Kapitel werden wir uns die Aggregation pipeline und MapReduce im Detail ansehen. Für jetzt reicht es aus sich diese Technologien als eine Art sehr vielfältige group-by vorzustellen (was aber eine himmelweite Untertreibung ist.). Für die Parallelverarbeitung von sehr sehr grossen Datenmengen werden sie wahrscheinlich trotzdem eher auf andere Systeme wie etwa Hadoop setzten. Glücklierweise ergänzen sich MongoDB und Hadoop sehr gut. Es gibt auch einen nativen MongoDB connector for Hadoop.

Natürlich ist gerade die Parallele Datenverarbeiutung (sehr grosser Datenmengen) etwas bei dem relationale Datenbanke auch nicht gerade glänzen. Für MongoDB gibt es Pläne die Verarbeitung sehr grosser Datenmengen in Zukunft besser zu unterstützen. (Anm.d.Ü.: Es wäre hier sicher sinnvoll eine Indikation zu geben was *sehr grosse Datenmengen* sind - ich erstelle einen github issue dazu).

Räumliche Daten / Geospatial

Eine besondere Stärke hat MongoDb im Bereich von räumlichen/Geodaten durch die direkte Unterstützung von geospatial indexes. Das erlaubt es ihnen direkt geoJSON oder x,y Koordination in Dokumenten zu speichern und dann Dokumente zu finden die nahe an einer Koordinatenmenge liege ($near) oder die sich in einer definierten Box oder in einem Kreis befinden ($within). Da das Features sind die man am besten visuell erklären kann möchte ich sie hier ermuntern sich einmal das 5 minute geospatial interactive tutorial anzusehen.

Tools und Reifge (maturity)

Sie kennne die Antwort darauf wahrscheinlich bereits, denn MongoDB ist nunmal offensichtlich jünger als die meinsten relationalen Datenbanksysteme. Das ist auch etwas was man immer berücksichtigen sollte, wobei die relevanz dieses Aspektes eben auch stark von dem abhängt *was* sie machen wollen und *wie* sie es machen wollen. Dennoch kann man bei einer ehrlichen Betrachtung nicht verleugnen, das MongoDB eben jünger ist und die Verfügbarkeit von Tools deshalb (noch) nicht grossartig ist (Allerdings sind die Tools die z.T rund um sehr lang existierende, relationale Datenbanken existieren zum Teil auch eher schrecklich). (Anm.d.Ü.: Wobei sich hier Aufgrund der grossen Popularität von MongoDB seit dem Erscheinen der Version 2.6 in 2014 eine Menge zum positiven getan hat) Beispielsweise könnte das fehlen von base-10 floating point numbers eine echte Einschränkung für Systeme sein, die mit Gelddaten arbeiten (auch wenn das kein echter Show-stopper sein muss.)

Auf der anderen Seite ist das Protokoll modern und einfachen, es existieren native Treiber für eine grosse Anzahl an Programmiersprachen und schreitet die Entwicklung mit unglaublicher Geschwindigkeit voran. MongoDB ist heute in so vielen Unternehmen im produktiven Einsatz, das Bedenken bezüglich der Reife, wenn auch berechtigt, schnell der Vergangenheit angehören werden.

In diesem Kapitel / In This Chapter

Die Botschaft dieses Kapitels ist es, das MongoDB in den meisten Fällen eine relationale Datenbank ersetzen kann. MongoDB ist leichter, direkter, schneller und erzeugt in der Regel weniger Restriktionen für (Applikations) Entwickler. Fehlende Transaktionen sind hingegen ein bereiten dem einen oder anderen Sorge. Wenn Leute mich fragen: *Und wo ordnet sich MongoDB schlussendlich in der Datenbank Landschaft ein?* ist die Antwort für mich leicht: **genau in der Mitte**

Kapitel 6 - Daten aggregieren / Aggregating Data

Aggregation Pipeline

Die aggregation pipeline ist ein Mittel um Dokumente in einer Collection zu transformieren und zu kombinieren. Dazu werden die Dokumente entlang einer Pipeline verarbeitetm, die sich so wie das klassische Unix "pipe" Kommando verhält. Der Output des ersten Kommandos ist der Input des Zweiten, dessen Output ist der Input des nächsten, usw.

Die einfachste Aggregation, die sie wahrscheinlich auch schon kennen, ist die SQL group-by exprression. Wir haben auch schon die einfache count()Methode kennengelernt. Wie können wir also sehen wie viele Einhörner (unicorns) männlich und wie viele weiblich sind ?

```
db.unicorns.aggregate([{$group:{_id:'$gender',
    total: {$sum:1}}}])
```

In der shell haben wir zu diesem Zweck die aggregate Hilfsfunktion, die ein Array von Pipeline Operatoren erhält. Für ein einfaches zählen, das irgendwie gruppiert ist, reicht ein einziger Operator, nämlich $group. Dieser entspricht tatsächlich genau dem GROUP BY aus SQL, bei dem wir ein neues Dokument mit der _id anlegen, nach der wir gruppieren wollen (hier: gender). Weiteren Feldern werden üblicherweise Ergebnisse von Aggregationen zugewiesen. In diesem Fall $sum 1 für jedes Dokument das einem spezifischen Geschlecht enstspricht. Ihnen ist vielleicht aufgefallen das wir dem _id Feld nicht gender sondern $gender zugewiesen haben. Das '$' vor einem Feldnamen bedeutet das der Wert des eingehenden Dokumentes ersetzt wird.

Welche anderen Pipeline Operatoren können sie noch benutzen ? Der am häufigsten in Verbindung mit $group verwendete Operator dürfte $match sein. Dieser entspricht genau der find Methode und erlaubt es nur die Dokumente zu aggregieren, die dem Filter entsprechen.

```
db.unicorns.aggregate([{$match: {weight:{$lt:600}}},
    {$group: {_id:'$gender',   total:{$sum:1},
      avgVamp:{$avg:'$vampires'}}},
    {$sort:{avgVamp:-1}} ])
```

An dieser Stelle haben wir gleich den nächsten Operator, nämlich $sort eingeführt. Der macht genau das was sie denken und lässt sich mit den Operatoren $skip und $limit erweitern. Des weiteren haben wir in der o.a. Aggregation noch den $groupOperator $avg benutzt.

MongoDB Arrays sind sehr mächtig und können deshalb ebenso in Aggregationen einbezogen werden. Wir müssen sie allerdings erstmal *glätten* um alle Attribute richtig zählen zu können:

```
db.unicorns.aggregate([{$unwind:'$loves'},
    {$group: {_id:'$loves',   total:{$sum:1},
    unicorns:{$addToSet:'$name'}}},
    {$sort:{total:-1}},
    {$limit:1} ])
```

Mit dieser Aggregation finden wir heraus, welches Essen bei Einhörnern besonders beliebt ist und wir erhalten zusätzlich zu jedem Nahrungs Item eine Liste der Einhörner die es mögen. Durch die Verwendung von $sort und $limit kann man sehr leicht "Top n" Fragen beantworten.

Ein weiterer sehr mächtiger Pipeline Operator ist $project (Mit ihm kann man projektionen wie bei find durchführen) mitr dem man nicht nur vorhandene Felder einbeziehen, sondern auch neue Felder erzeugen kann die auf vorhandenen Werten basieren. Man könnte Beispielsweise mathematische Operatoren verwenden um zunächst mehrere Felder zu addieren und anschliessend den Durchschnitt über alles zu ermitteln. Man könnte auch String Operatoren verwenden um ein neues Feld mit konkatenierten Werten vorhandener Strings zu erzeugen.

Das alles berührt aber nur die Oberfläche von dem was sie mit Aggregationen wirklich machen können-In Version 2.6 sind Aggregationen noch einmal deutlich mächtiger geworden da sie nun entweder einen Cursor zurückgeben (Den sie bereits aus Kapitel 1 kennen) oder, mit dem $out Pipeline Operator, die Ergebnisse auch in eine neue Collection schreiben können. Sie finden eine Menge mehr Beispiele und alles Unterstützten Pipeline Expressions und Operatoren hier im MongoDB manual.

MapReduce

MapReduce ist ein wzeischrittiger Ansatz der Datenverarbeitung. Als erstes erfolgt ein map, dann das reduce (Anm.d.Ü: Nomen est Omen ;). Der mapping Schritt transformiert die Eingabedokumente und erzeugt ein key=>value Paar (Wobei der key und die values ruhig komplex sein können). Dann werden die key/value Paare nach keys gruppiert, so das alle Werte für denselbe key in einem Array landen. Der reduce Schritt erhält jeweils die keys und die dazugehörigen Werte-Arrays und generiert daraus das endgültige Ergebnis. Die map und reduce Funktionen werden in Javascript geschrieben.

Bei MongoDB benutzen wir das mapReduce Kommando auf Collection Ebene. mapReduce erhält eine map Funktion, eine reduce Funktion und eine output Direktive. In der shell kann man dazu direkt JavaScript Funktionen anlegen und an mapReduce übergeben. Aus den meisten anderen Libraries übergibt man die Funktionen als String (was sich ein bisschen unschön anfühlt). Der Dritte Parameter (output) setzt zusätztliche Optionen. Besipielsweise kann man hier die Dokumente filtern, sortieren oder limitieren bevor sie Analysiert werden. Man kann auch noch eine finalize Methode angeben, die nach dem reduce Schritt ausgeführt wird.

Wahrscheinlich werden sie MapReduce für die allermeisten ihrer Aggregationen nicht benötigen aber wenn es doch der Fall sein sollte, dann können sie in meinem Blog und im MongoDB manual mehr darüber erfahren.

In diesem Kapitel

In diesem Kapitel haben wir MongoDB's Aggregations Möglichkeiten behandelt. Die Aggregation Pipeline ist recht einfach zu benutzen, wenn man einmal die Struktur verstanden hat. Und sie ist ein mächtiges Werkzeug um Daten zu gruppieren (aggregieren). MapReduce ist hingegen deutlich komplexer, dafür sind die Möglichkeiten aber im Grunde nur durch die Limits von JavaScript begrenzt.

Chapter 7

In diesem letzten Kapitel werden wir uns mit Performance Aspekten und Tools für MongoDB Entwickler beschäftigen. Wir werden jeweils nicht sehr tief in die Themen eintauchen aber die jeweils wichtigsten Aspekte herausarbeiten.

Indizes / Indexes

Ganz am Anfang des Buches haben wir die spezielle system.indexes collection kennengelernt, in der man Informationen über alle Indizes einer Datenbank finden kann. Indizes in MongoDb funktionieren im Grunde genauso wie man sie bei relationalen Datenbanken kennt: Sie helfen dabei die Abfrage- und Sortierperformance zu erhöhen. Inizes werden mit ensureIndex erzeugt:

```
// where "name" is the field name
db.unicorns.ensureIndex({name: 1});
```

und mit dropIndex wieder gelöscht:

```
db.unicorns.dropIndex({name: 1});
```

Um einen unique Index zu erstellen kann man den Parameter unique auf **true** setzen:

```
db.unicorns.ensureIndex({name: 1},
    {unique: true});
```

Indizes können auch auf eingebetteten Felder (wieder mit der dot Notation) und Arrays erzeugt werden. Man auch sogenannte compound (Verbundene) Indizes erzeugen:

```
db.unicorns.ensureIndex({name: 1,
    vampires: -1});
```

Die Richtung des Index (1 für aufsteigend und -1 für absteigend) macht für einen einzelnen Index keinen Unterschied, kann aber einen Unterschied für verbundene Indizes machen, wenn man z.B eine Sortierung auf mehr als einem Indexfeld durchführt.

Weiterführende Informationen finden sie auf der MongoDB Index Seite

Explain

Um zu sehen ob ihre Abfragen einen Index verwenden können sie die explain Methode auf einen Cursor anwenden:

```
db.unicorns.find().explain()
```

Die Ausgabe zeigt uns, das ein BasicCursor verwendet wird (also ohne Index), das 12 Objecte gescanned wurden, wie lange es gedauert hat, welcher Index verwendet wurde, wenn denn einer verwendet wurde und noch ein paar weitere nützliche Informationen.

Wenn wir unsere Abfrage so ändern das ein Index verwendet wird, werden wir sehen das ein BtreeCursor verwendet wurde und auch welche Index konkret benutzt wurde um die Abfrage durchzuführen:

```
db.unicorns.find({name: 'Pilot'}).explain()
```

Replikation / Replication

Die MongoDB Replikation funktioniert im Grunde genommen vergleichbar zur Repliaktion bei relationalen Datenbanken. Alle Produktivumgebungen sollten aus Replika Sets bestehen die mindestens 3 Instanzen umfassen auf denen die Daten repliziert werden. Schreiboperationen werden an einen primären Server gesendet, von dem aus sie asynchron auf weitere (secondary) Server repliziert werden. Über die Konfiguration können sie festlegen ob Abfragen auf secondary Servern zugelassen sind oder nicht. Damit können sie Last vom primary Server verteilen, nehmen aber in Kauf leicht inaktuelle Daten zu erhalten (stale data). Sollte der primary Server ausfallen wird automatisch einer der secondary Server zu neuen primary. Die genauen Details der MongoDB Replikation sind aber nicht Teil dieses Buches.

Sharding

MongoDB unterstützt auto-sharding. Unter sharding versteht man einen technischen Ansatz zur Skalierung der Daten automatisch auf verschiedene Server partitioniert. Ein einfaches Beispiel wäre alle Userdaten deren Namen mit A-M anfangen auf Server 1 zu speichern und alle anderen auf Server 2. Glücklicherweise gehen MongoDB's sharding Funktionalitäten und Möglichkeiten weit über einen solch simplen Ansatz hinaus. Sharding würde den Scope des Buches deutlich sprengen aber sie sollten einfach wissen das so etwas in MongoDB exisitert und sie es in Betrachtung ziehen können wenn sie mehr als ein Replika set benötigen.

Während Replikation machmal auch dazu dienen kann die Performance zu erhöhen (indem man z.B. lange Abfragen an secondaries sendet) dient sie im allgemeinen aber zur Erhöhung der Verfügbarkeit (high availability). Sharding is demgegenüber die primäre Methode zur Skalierung von MongoDB Clustern (Also der Performance Erhöhung). Die Kombination aus Relikation und Sharding ist allerings *kein* Ansatz um Skalierung und Verfügbarkeit zu erreichen. (Original: Combining replication with sharding is the proscribed approach to achieve scaling and high availability.)

Statistiken / Stats

Sie können mit dem Kommando db.stats() statistische Informationen über ihre Datenbank abrufen. Bei denen meisten dieser Informationen geht es um die Grösse der Datenbank. Sie können aber auch Statistiken über Collections abrufen. Wenn sie z.B. Informationen über die unicorns Collection abrufen möchten können sie das mit dem Kommando db.unicorns.stats() machen. Auch hier beziehen sich die meisten Informationen auf die Grösse der Collection und ihrer Indizes.

Profiler

Mit folgendem Kommando können sie den MongoDB profiler aktivieren:

```
db.setProfilingLevel(2);
```

Führen sie jetzt ein Kommando aus, wie z.B.:

```
db.unicorns.find({weight: {$gt: 600}});
```

und untersuchen sie dann den Profiler:

```
db.system.profile.find()
```

Die Ausgabe sagt ihnen was ausgeführt wurde, wann es ausgeführt wurde, wie viele Dokumente untesucht wurden und wie viele Daten zurückgegeben wurden.

Um den Profiler zu deaktivieren führen sie wieder das Kommando setProfilingLevel aus, diesmal mit dem Paramter 0. Wenn sie 1 als Parameter setzen werden alle Abfragen (queries) untersucht, die mehr als 100 Millisekunden benötigen. 100 Millisekunden sind der Standard Schwellwert. Sie können andere Schwellwerte setzen indem sie diese über eine zweiten Parameter definieren:

```
//profile anything that takes
//more than 1 second
db.setProfilingLevel(1, 1000);
```

Backups und Restore / Backups and Restore

Im MongoDb bin Verzeichnis finden sie eine asführbare Datei Namens mongodump. Wenn sie mongodump einfach ausführen wird eine Verbindung zu localhost aufgebaut und alle Datenbanken werden im dump Unterverzeichnis gesichert (backup). Um weiter Optionen zu sehen können sie mongodump --help eingeben. Oft genutzte Optionen sind --db DBNAME um eine spezielle Datenbank zu sichern oder -- collection COLLECTIONNAME um eine spezielle Collection zu sichern. Um eine gesicherte Datenbank wieder herzustellen können sie das Programm mongorestore verwenden, das sich ebenfalls im bin Verzeichnis befindet. Auch hier können sie die Optionen --db und -- collection verwenden um spezielle Datenbanken oder Collections wieder herzustellen. Sowohl mongodump als auch mongorestore arbeiten auf BSON, dem nativen MongoDB Datenformat.

Wenn wir Beispielsweise unsere learn Datenbank im Verzeichnis backup sichern wollen, dann können wir folgendes Kommando ausführen (Achtung: dieses Kommando wird direkt und nicht in der MongoDB shell ausgeführt.)

```
mongodump --db learn --out backup
```

Wenn wir dann nur die unicorns collection wieder herstellen möchten können wir das wie folgt tun:

```
mongorestore --db learn --collection unicorns \
    backup/learn/unicorns.bson
```

An diesere Stelle möchte ich darauf hinweisen, das mit mongoexport und mongoimport zwei weitere Programme existieren um Daten in JSON oder CSV zu im- oder exportieren. Einen JSON output können wir z.B. mit folgendem Kommando erzeugen:

```
mongoexport --db learn --collection unicorns
```

Und einen CSV Export mit:

```
mongoexport --db learn \
    --collection unicorns \
    --csv --fields name,weight,vampires
```

Beachten sie aber bitte das mongoexport und mongoimport nicht alle BSON Datentypen vollständig repräsentieren können. Zur produktiven Datensicherung (backup) sollten sie also immer mongodump und mongorestore verwenden. Wenn sie mehr über Datensicherung erfahren wollen lesen sie bitte das Kapitel your backup options im MongoDB Handbuch.

In diesem Kapitel

Haben wir uns mit diversen Kommandos, Tools und Performance Details rund um MongoDB beschäftigt. Es gibt noch viel mehr rund um MongoDB als wir in diesem Buch behandeln konnten. Aber ihre nächsten Schritte sollten sich darum drehen das gelernte anzuwenden und sich näher mit dem konkreten Treiber ihrer Programmiersprache zu beschäftigen. Die MongoDB website](http://www.mongodb.org/) hat eine Menge nützlicher Informationen un die offizielle MongoDB user group](http://groups.google.com/group/mongodb-user) ist ein grossartiger Platz um Fragen zu stellen.

NoSQL wurde nicht nur aus der Notwendigkeit heraus gebohren sondern auch aus dem Interesse neue Ansätze zu erproben. Es ist auch ein Tribut an das Wissen das sich unser Arbeitsfeld immer weiterentwickelt und das wir niemals erfolgreich sein werden, wenn wir nicht neue Ansätze erproben. Ich persönlich denke das das ein guter Weg ist unser berufliches Leben gut zu gestalten.