Tobias Trelle

MongoDB

Der praktische Einstieg

dpunkt.verlag

Dipl.-Math. Tobias Trelle ist Senior IT Consultant bei der codecentric AG, Solingen. Er ist seit knapp 20 Jahren im IT-Business unterwegs und interessiert sich für Softwarearchitekturen und skalierbare Lösungen. Zu diesen Themen veröffentlicht Tobias Artikel in Fachzeitschriften und hält Vorträge auf Konferenzen und Usergruppen. Außerdem organisiert er die Düsseldorfer MongoDB-Usergruppe.

Tobias Trelle

MongoDB

Der praktische Einstieg

dpunkt.verlag

Tobias Trelle
tobias.trelle@codecentric.de

Lektorat: René Schönfeldt
Copy-Editing: Sandra Gottmann, Münster
Herstellung: Birgit Bäuerlein
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN
Buch 978-3-86490-153-9
PDF 978-3-86491-533-8
ePub 978-3-86491-534-5

1. Auflage 2014

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1 0

❖❖❖ Für Vera und Kelian ❖❖❖

Inhaltsverzeichnis

Anhang

1 Einleitung

1.1 Big Data

Sie wollen ein Buch über MongoDB lesen und werden als Erstes mit dem Begriff *Big Data* konfrontiert. Was hat MongoDB mit Big Data zu tun, werden Sie sich fragen. Was ist Big Data überhaupt genau? Um die Motivation zu verstehen, die zur Entstehung von MongoDB (und anderen nichtrelationalen Datenbanken) führte, will ich Ihnen zunächst eine Begriffsdefinition von Big Data geben.

Die grundlegenden technischen Herausforderungen im Big-Data-Umfeld wurden bereits 2001 von Doug Laney (heute Analyst bei Gartner) formuliert[1] und sind heute unter der Abkürzung *3V* bekannt. Dabei handelt es sich um die Aspekte:

- Volume
- Velocity
- Variety

Bei *Volume* geht es um die schiere Menge an Daten, die pro Zeiteinheit gespeichert werden muss.

Velocity adressiert die Geschwindigkeit, mit der die Daten persistiert, aber auch verarbeitet werden. Die Verarbeitung kann dabei im Batch oder aber auch in (Fast-)Echtzeit gewünscht sein.

Der Aspekt *Variety* bezieht sich auf die unterschiedlichen Grade von Strukturiertheit der Daten, das reicht von völlig unstrukturiert über semistrukturiert bis hin zu stark strukturiert.

Daneben machen zwei weitere Dinge den Begriff Big Data aus: Kostenersparnis durch elastische Skalierung und die Gewinnung neuer Informationen aus den Daten.

Ich möchte Ihnen folgende Fragen in Bezug auf Ihr relationales Datenbanksystem stellen. Haben Sie Probleme mit der Anzahl an schreibenden Operationen pro Sekunde? Wie schnell bzw. wie oft können und wollen Sie Ihre

1. *http://blogs.gartner.com/doug-laney/deja-vvvue-others-claiming-gartners-volume-velocity-variety-construct-for-big-data/*

Reports erstellen? Wie gut passt Ihr logisches Datenmodell wirklich zum relationalen Modell?

Wenn Sie mit keinem dieser Aspekte oder ähnlich gelagerten Fragestellungen Probleme haben, dann haben Sie wahrscheinlich auch kein Big-Data-Problem. Stehen Sie aber vor einer oder allen diesen Herausforderungen, dann Sie sind sicherlich schon über den Begriff *NoSQL* gestolpert.

1.2 NoSQL

Der Begriff NoSQL[2] ist keineswegs ein Imperativ, der zum vollständigen Boykott relationaler Datenbanksysteme aufruft. Er leitet sich ab von *not only SQL*, wobei SQL als Synonym für relationale Datenbankensysteme verwendet wird.

Die Grundidee ist die, dass man sich beim Umsetzen seiner konkreten Anforderungen (ob nun Big Data oder nicht) nicht im Vorhinein auf relationale Datenbanksysteme (RDBMS) beschränkt, nur weil man dies bei der Datenhaltung in den letzten Dekaden immer so getan hat. Riskieren Sie ruhig mal den Blick über den Tellerrand und setzen Sie das Datenbanksystem ein, das Ihr Problem optimal löst. Können Sie Ihr Problem am besten mit einer relationalen Datenbank lösen, dann tun Sie das auch und laufen nicht dem Hype hinterher.

Denn zum Einsatz von NoSQL-Technologien gehört Mut: Datenbankadministratoren haben wenig bis keine Erfahrung mit diesen neuartigen Technologien, Entwickler kennen die meist nichtstandardisierten APIs (noch) nicht. Das möchte ich mit diesem Buch in Bezug auf MongoDB ändern.

Entstanden sind die NoSQL-Datenbanken in der Web-2.0-Welt. Soziale Netzwerke wie Facebook, Twitter und Co. haben mit vielen Millionen, über den ganzen Globus verteilten Benutzern ganz andere Anforderungen an die Datenhaltung als das Bestandsführungssystem eines Versicherungsunternehmens. Es müssen sehr viele Daten sehr schnell gespeichert und abgerufen können. Dabei stoßen relationale Datenbanksysteme durchaus an ihre Grenzen, sowohl technologisch

2. *http://de.wikipedia.org/wiki/NoSQL*

als auch vom Lizenzmodell her. Um eine fundierte Entscheidung bei der Auswahl der passenden NoSQL-Datenbank treffen zu können, ist natürlich – wie immer – Know-how gefragt, zumal sich in diesem Umfeld sehr viele Anbieter und auch Konzepte tummeln. Die wichtigsten Kategorien von NoSQL-Datenbanken[3] sind:

- *Key-Value*-Datenbanken verwalten Tupel bestehend aus einem Schlüssel und einem Wert. Abfragen sind nur über den einen Schlüssel möglich. Der Wert des Datensatzes ist in der Regel ein Byte-Array, ist also im Sinne des Variety-Aspekts völlig unstrukturiert.
- *Spaltenorientierte* Datenbanken, im Englischen *Wide Column Stores*, verwenden Tabellen, bei denen ein Datensatz allerdings eine dynamische Anzahl an Spalten haben kann. Sie können damit als eine Verallgemeinerung von Key-Value-Datenbanken angesehen werden. Indizes sind frei definierbar und ermöglichen die Abfrage über beliebige Spalten.
- *Graphen*-Datenbanken spezialisieren sich auf die Verwaltung von Knoten und Kanten zwischen diesen Knoten. Abfragen ermöglichen u.a. das Traversieren des Graphen.
- *Dokumentenorientierte* Datenbanken werde ich im Folgenden noch näher erklären.

Abbildung 1–1 zeigt einige der populärsten Vertreter:

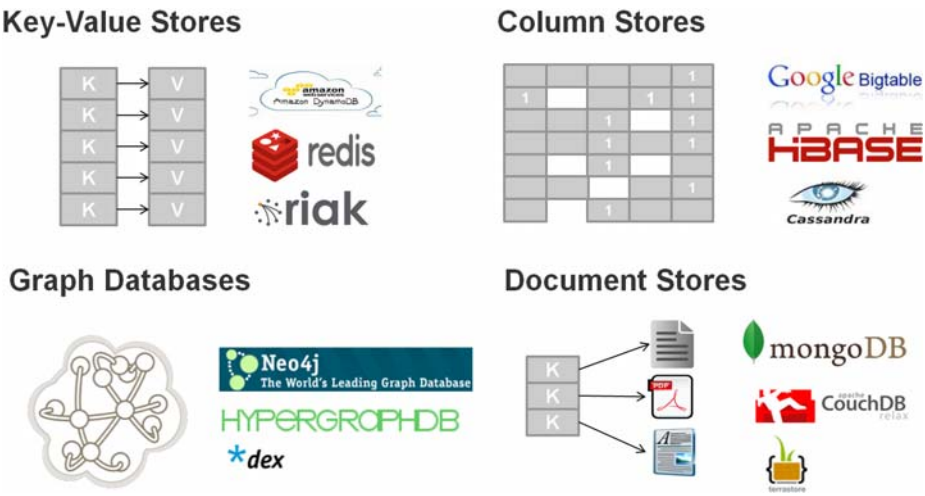


Abb. 1–1 *NoSQL-Datenbanken im Überblick*

Relationale Datenbanksysteme sind Allzweckwaffen, die sich in der Regel auf eine große Menge von Problemen anwenden lassen. Im Gegensatz dazu stellen

3. Stefan Edlich pflegt eine sehr umfangreiche Liste mit NoSQL-Datenbanken: *http://www.nosql-database.org/*

NoSQL-Datenbanken tendenziell eher Nischenlösungen dar, die bestimmte Probleme allerdings sehr viel besser lösen können.

1.3 Dokumentenorientierte Datenbanken

MongoDB gehört zur Kategorie der sogenannten dokumentenorientierten Datenbanken. Damit werden wir uns im weiteren Verlauf dieses Buches sehr detailliert auseinandersetzen. Ein *Dokument* ist ein einzelner Datensatz, der im Prinzip aus einer geordneten Liste von Key-Value-Paaren besteht und als Werte auch Arrays und eingebettete Dokumente zulässt. Ein Dokument kann man gut im JSON-Format darstellen, z.B. so:

```
{
    "name": "MongoDB",
    versionen: [
        { "major": 2, "minor": 6 },
        { "major": 2, "minor": 4 },
        { "major": 2, "minor": 2 }
    ]
}
```

Zur Speicherung verwendet MongoDB intern allerdings nicht das JSON-Format, sondern eine Abwandlung davon. Dazu aber später mehr. Mit diesem Datenformat adressiert MongoDB den Big-Data-Aspekt *Variety*: Ein Dokument eignet sich hervorragend zum Umgang mit semistrukturierten Daten, z.B. komplexen Vererbungshierarchien in OO-Sprachen. Aber auch für *Volume* und *Velocity* bietet MongoDB Lösungen an, auf die ich in den folgenden Kapiteln noch eingehen werde.

Zum Ursprung von MongoDB

Als ich zum ersten Mal von MongoDB gehört habe, war ich angesichts des Namens etwas verwundert. Denn in vielen Sprachräumen, so auch im Deutschen, hat der Begriff *Mongo* eine eher negative Konnotation. Tatsächlich leitet sich der Begriff offiziell jedoch vom Englischen *hu**mongo**us*[a] ab, was so viel wie *gigantisch* oder *wahnsinnig groß* bedeutet, was dann zu einem versteckten Hinweis auf den *Volume*-Aspekt von Big Data interpretiert werden kann.

MongoDB ist Open Source (Gnu AGPL), es gibt allerdings auch kommerziellen Support von der Firma *MongoDB Inc.* Diese wurde 2007 von den ehemaligen Doubleclick-Gründern Dwight Merriman und Eliot Horowitz unter dem Namen *10gen* gegründet. In 2013 erfolgte die Umbenennung in MongoDB Inc., vermutlich um die immer weiter wachsende Popularität der Datenbank im Firmennamen widerzuspiegeln.

a. *http://www.kchodorow.com/blog/2010/08/23/history-of-mongodb/*

1.4 Verteilte Systeme und das CAP-Theorem

Den Big-Data-Aspekten *Volume* und *Velocity* ist in der Regel nur mit dem Einsatz von verteilten Systemen[4] zu begegnen, in denen die Last und die Daten auf viele einzelne Rechnerknoten verteilt werden.

Aus dem CAP-Theorem (s. Kasten) folgt allerdings, dass man bei der Implementierung eines verteilten (Datenbank-)Systems nicht alle der Anforderungen an Konsistenz, Verfügbarkeit und Toleranz gegenüber Ausfällen gleichzeitig in vollem Maße erreichen kann.

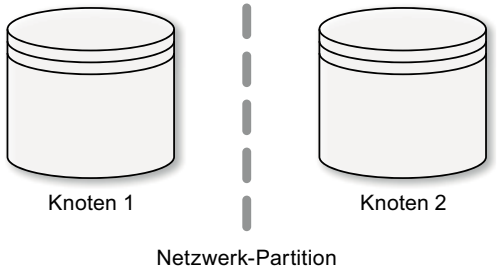


Abb. 1–2 *Veranschaulichung des CAP-Theorems*

CAP-Theorem

In seiner ursprünglichen Fassung aus dem Jahre 2000 besagt das sogenannte *CAP-Theorem*[a] (oder auch Brewers Theorem), dass in verteilten Systemen maximal zwei der drei folgenden Eigenschaften gleichzeitig gelten können:

- Consistency (C):
  Alle Knoten haben jederzeit den gleichen Datenbestand.
- Availability (A):
  Das System steht für Lese- und Schreibzugriffe zur Verfügung.
- Partition Tolerance (P):
  Toleranz gegenüber dem Ausfall einzelner Knoten und/oder Netzwerkstrecken.

Die Eigenschaften sind dabei als graduelle Größen zu verstehen, d.h., sie können irgendwo zwischen gar nicht und voll erfüllt liegen.

a. *http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf*

Die Auswirkungen des CAP-Theorems lassen sich gut an einem System veranschaulichen, das aus zwei Knoten besteht (s. Abb. 1–2). Das Netzwerk zwischen diesen Knoten soll gestört sein, sodass eine Partition des Systems in zwei Teile vorliegt, die sich gegenseitig nicht mehr erreichen können.

Erlaubt man nun einem der Knoten, seinen Zustand zu ändern (also im Falle eines Datenbanksystems schreibende Operationen auszuführen), wird das Gesamt-

4. *http://de.wikipedia.org/wiki/Verteilte_Systeme*

system inkonsistent, man gibt also *C* auf. Will man die Konsistenz erhalten, muss der Knoten, dessen Zustand sich nicht ändert, sich als nicht verfügbar »abmelden« (da er sonst gegenüber den Clients einen nicht aktuellen Datenbestand ausliefern würde), womit man die Verfügbarkeit (*A*) des Systems auf nur noch einen Knoten reduziert. Nur wenn alle Knoten stets miteinander kommunizieren können, kann man dauerhaft *C* und *A* aufrecht erhalten, was dann offenbar im Widerspruch zu *P* steht.

Relationale Datenbanksysteme fallen in der Regel in die CA-Kategorie, da sie großen Wert auf die Konsistenz legen und auch hohe Verfügbarkeiten garantieren. Bei Systemen, die nur auf einem Knoten laufen, kann die Ausfallsicherheit durch den Kauf sehr teurer Hardware ebenfalls nah an die 100% gebracht werden. Fällt der eine Knoten aus, ist das System aber nicht mehr verfügbar. Bei Datenbank-Clustern verringert sich die Verfügbarkeit, da die Transaktionen zur Sicherstellung der Konsistenz auf mehreren Knoten eine längere Laufzeit haben.

Bei vielen NoSQL-Datenbanken kann man aufgrund der hohen Verteilung nicht auf die Partition Tolerance (*P*) verzichten. Da die Verfügbarkeit (*A*) ebenfalls einen sehr hohen Stellenwert einnimmt (da man grundsätzlich alle Anfragen an das System beantworten können möchte), bleibt nur der Ausweg, an der Konsistenz (*C*) zu sparen. Dies ist dann aber eine bewusste Design-Entscheidung beim Entwurf dieser Systeme und nicht dem Unwillen oder der Vergesslichkeit der Entwickler geschuldet.

Sie als Anwender solcher Technologien sollten sich aber stets bewusst sein, dass Sie sich nicht mehr auf Ihre bisherigen Konsistenzzusagen verlassen können!

1.5 ACID vs. BASE

Das ACID[5]-Prinzip, das Sie von relationalen Datenbanksystemen gewohnt sind, stellt in Zusammenhang mit Transaktionen[6] Folgendes sicher:

- Atomicity (Atomarität):
  Eine Transaktion wird nach dem Alles-oder-nichts-Prinzip entweder vollständig oder gar nicht ausgeführt. Wird eine atomare Transaktion abgebrochen, ist das System in einem unveränderten Zustand.
- Consistency (Konsistenz):
  Eine Folge von Datenbankoperationen führt wieder zu einem konsistenten Zustand, sofern das System zuvor schon in einem konsistenten Zustand war. Hier bezieht sich die Konsistenz vorwiegend auf die inhaltliche und referenzielle Integrität[7] des Datenbestandes.

5. *http://de.wikipedia.org/wiki/ACID*
6. *http://de.wikipedia.org/wiki/Transaktion_(Informatik)*
7. *http://de.wikipedia.org/wiki/Referentielle_Integrit%C3%A4t*

- Isolation:
  Nebenläufig ausgeführte Transaktionen beeinflussen sich nicht gegenseitig.
- Durability (Dauerhaftigkeit):
  Die Auswirkungen von Transaktionen müssen dauerhaft im System gespeichert werden, insb. auch bei Systemabstürzen.

Bitte beachten Sie, dass der Konsistenzbegriff aus dem CAP-Theorem und dem ACID-Prinzip nicht identisch sind, sondern sich auf verschiedenen Granularitätsebenen bewegen.

NoSQL-Datenbanken können (oder wollen) eine Konsistenz im Sinne des ACID-Prinzips oft nicht garantieren, gewährleisten dann aber einen anderen Grad an Konsistenz. Hier kommt der Begriff BASE[8] ins Spiel:

- *B*asically *A*vailable
- *S*oft State
- *E*ventual Consistency

Dieses zugegebenermaßen sehr konstruiert wirkende Akronym will den Gegensatz zu ACID (im Englischen Säure) und BASE (Lauge) aus der Chemie aufgreifen. Die beiden Begriffe *Basically Available* und *Soft State* sind nicht präzise definiert, greifen aber nach meiner Interpretation die Verfügbarkeit (A) des CAP-Theorems auf.

Unter *Eventual Consistency* wird relativ klar das C des CAP-Theorems adressiert, und zwar in dem Sinne, dass das verteilte System (nach einer möglichst kurzen Zeitspanne der Inkonsistenz) *letztendlich* wieder in einem konsistenten Zustand ist.

Die letztendliche Konsistenz möchte ich an einem konkreten Beispiel mit zwei Knoten erläutern. Schreibzugriffe sind nur an einem der Knoten, den wir Master nennen, erlaubt. Der Master repliziert den Datenbestand in regelmäßigen Abständen an den anderen Knoten, den wir Slave nennen. Lesezugriffe sind an beiden Knoten erlaubt. Nach einem Schreibvorgang auf dem Master sehen Clients, die vom Slave lesen, die neuen oder geänderten Daten nicht sofort. Erst nach Abschluss der Replikation, die ggf. asynchron arbeitet, ist der Slave wieder auf dem aktuellen Stand.

Mit MongoDB können Sie keine Transaktionen fahren, die mehr als ein Dokument umfassen. Selbst bei Operationen auf nur einem Dokument ist kein explizites Rollback möglich. Immerhin sind Änderungen an einem Dokument aber atomar im Sinne des ACID-Prinzips. Da Sie ganze Objektnetze in einem Dokument unterbringen können, sind dokumentenübergreifende Transaktionen oft auch gar nicht notwendig.

---

8. *http://en.wikipedia.org/wiki/Eventual_consistency*

1.6 Zusammenfassung

In dieser Einleitung haben Sie die Herausforderungen kennengelernt, die es im Big-Data-Umfeld zu meistern gilt. Sie haben ferner die Grundidee hinter der NoSQL-Bewegung verstanden und wissen, dass Sie insbesondere bei der Konsistenz umdenken müssen, wenn Sie MongoDB einsetzen.

Was Sie dabei beachten müssen, damit Sie dennoch keine Daten verlieren, werde ich Ihnen im Rest des Buches zeigen. Lassen Sie uns also jetzt direkt durchstarten in die Welt abseits des relationalen Pfades, in eine Welt ohne ACID, in die Welt von MongoDB ...

2 MongoDB in 21 Minuten

In diesem Kapitel zeige ich Ihnen, wie schnell Sie einen MongoDB-Server installieren, starten und benutzen können. Sie werden Testdaten importieren und Ihre ersten Abfragen absetzen.

2.1 Installation

Die jeweils aktuelle Version von MongoDB kann auf der Homepage heruntergeladen werden: *http://www.mongodb.org/downloads*. Sie sollten die stabile Produktionsversion verwenden. Ich werde die Installation exemplarisch für Windows durchführen, Anleitungen für andere Betriebssysteme können Sie der Online-Dokumentation[1] entnehmen.

Nachdem die passende Archivdatei heruntergeladen ist, entpacke ich den Inhalt in ein Verzeichnis meiner Wahl, z.B. `C:\nosql`. Dort befindet sich nun ein neues Verzeichnis, dessen Name mit `mongodb` gefolgt von Betriebssystem- und Versionsinfos beginnt, z.B.:

```
c:\nosql\mongodb-win32-x86_64-2008plus-2.x
```

Auf dieses Verzeichnis werde ich mich im Folgenden als `MONGO_HOME` beziehen.

2.2 Server starten

Um den Server-Prozess zu starten, öffnen Sie eine Kommandozeilen-Shell und wechseln in das Verzeichnis `MONGO_HOME\bin`. Dort befinden sich alle ausführbaren Programme, die wir zum Betrieb und zur Administration eines MongoDB-Systems benötigen. Zunächst müssen wir noch ein neues Verzeichnis anlegen, in dem MongoDB seine Daten speichert:

1. *http://www.mongodb.org/display/DOCS/Quickstart*

```
rem Windows
mkdir \data

# Unix
mkdir /data
```

Anschließend starten wir den Server mit

```
mongod --dbpath /data
```

Auf der Konsole sollten nun in etwa folgende Log-Ausgaben zu sehen sein:

```
Sat Dec 01 14:15:40 [initandlisten] MongoDB starting : pid=5616 port=27017
dbpath=/data 64-bit host=localhost
Sat Dec 01 14:15:40 [initandlisten] db version v2.x, pdfile version 4.5
Sat Dec 01 14:15:40 [initandlisten] git version:
f5e83eae9cfbec7fb7a071321928f00d1b0c5207
Sat Dec 01 14:15:40 [initandlisten] build info: windows
sys.getwindowsversion(major=6, minor=1, build=7601, platform=2,
service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
Sat Dec 01 14:15:40 [initandlisten] options: { dbpath: "/data", rest: true }
Sat Dec 01 14:15:40 [initandlisten] journal dir=/data/journal
Sat Dec 01 14:15:40 [initandlisten] recover : no journal files present, no
recovery needed
Sat Dec 01 14:15:40 [initandlisten] waiting for connections on port 27017
```

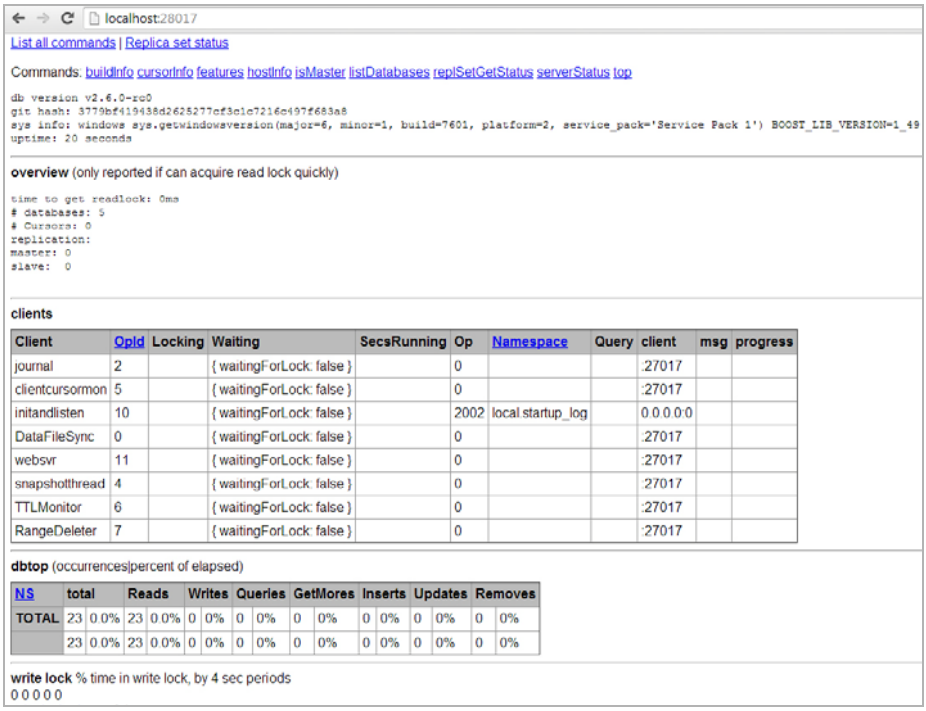
Ohne weitere Konfiguration verwendet ein MongoDB-Server als Default den TCP/IP-Port 27017 für eingehende Verbindungen. Wenn Sie den Server mit der Kommandozeilenoption

```
$ mongod --httpinterface --rest
```

starten, können Sie 1000 Ports höher, also unter folgender URL:

```
http://localhost:28017
```

eine einfache, HTML-basierte Administrationsoberfläche erreichen (s. Abb. 2–1)



localhost:28017

List all commands | Replica set status

Commands: buildInfo cursorInfo features hostInfo isMaster listDatabases replSetGetStatus serverStatus top

```
db version v2.6.0-rc0
git hash: 3770bf419438d2625277cf3c1c7216c497f683a8
sys info: windows sys.getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
uptime: 20 seconds
```

overview (only reported if can acquire read lock quickly)

```
time to get readlock: 0ms
# databases: 5
# Cursors: 0
replication:
master: 0
slave:  0
```

clients

Client	Opid	Locking	Waiting	SecsRunning	Op	Namespace	Query	client	msg	progress
journal	2		{ waitingForLock: false }		0			:27017		
clientcursormon	5		{ waitingForLock: false }		0			:27017		
initandlisten	10		{ waitingForLock: false }		2002	local.startup_log		0.0.0.0:0		
DataFileSync	0		{ waitingForLock: false }		0			:27017		
websvr	11		{ waitingForLock: false }		0			:27017		
snapshotthread	4		{ waitingForLock: false }		0			:27017		
TTLMonitor	6		{ waitingForLock: false }		0			:27017		
RangeDeleter	7		{ waitingForLock: false }		0			:27017		

dbtop (occurrences|percent of elapsed)

NS	total		Reads		Writes		Queries		GetMores		Inserts		Updates		Removes	
TOTAL	23	0.0%	23	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
	23	0.0%	23	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%

write lock % time in write lock, by 4 sec periods
0 0 0 0 0

Abb. 2–1 *Interface zur Administration*

Dort finden sich Informationen über verbundene Clients, Datenbankoperationen und aufgetretene Locks, auf die wir im weiteren Verlauf des Buchs näher eingehen werden.

2.3 Mongo Shell

Im weiteren Verlauf werden wir die sogenannte *Mongo Shell* verwenden. Dabei handelt es sich um einen Kommandozeilen-Client, mit dem wir den Server administrieren und auch Daten manipulieren können.

Als Benutzer erhalten Sie einen Prompt, auf dem Sie JavaScript[2]-Befehle und -Skripte ausführen können. Dabei können Sie auf eine Reihe bestehender JavaScript-Objekte zugreifen, die jeweils bestimmte Datenbankkonstrukte repräsentieren. Aus Sicht des Servers verhält sich die in C++ implementierte Mongo Shell wie jeder andere Datenbank-Client.

Gestartet wird das Ganze im Verzeichnis `MONGO_HOME/bin`:

```
$ mongo
```
[3]

2. Intern verwendet die Mongo Shell Googles V8 JavaScript-Engine
3. Eine Auflistung aller Kommandozeilenparameter finden Sie im Anhang.

Da der zuvor gestartete Server auf dem Standardport 27017 horcht, müssen wir keine weiteren Verbindungsparameter angeben. Danach werden wir von der Mongo Shell wie folgt begrüßt:

```
MongoDB shell version: 2.x
connecting to: test
>
```

Wenn wir beim Start der Mongo Shell keinen Namen einer Datenbank angeben, verwenden wir standardmäßig eine zunächst leere Datenbank namens `test`. Setzen Sie zum Einstieg ein paar einfache JavaScript-Befehle ab:

```
> var a = 2
> b = a + 2
4
> for (i=0; i<5; i++) { print(i) }
0
1
2
3
4
```

2.4 Erste Schritte

Ein MongoDB-Server verwaltet mehrere logische Datenbanken. Einen Überblick über alle Datenbanken verschaffen wir uns mit dem Befehl

```
> show dbs
local   0.078125GB
```

In einem frisch aufgesetzten System sehen wir nur die stets existierende Datenbank `local`.

Eine Übersicht über weitere Befehle erhalten Sie mit dem Kommando `help()`.

2.4.1 Collections

Einzelne Datensätze, die in MongoDB Dokumente heißen, werden innerhalb logischer Namensräume organisiert, den sogenannten *Collections*. Eine Collection entspricht in etwa einer Tabelle aus einem relationalen Datenbanksystem, ist aber nicht identisch damit. Collections müssen nicht explizit angelegt werden, sie werden bei der ersten Verwendung automatisch erzeugt. In unserer Datenbank `test` gibt es zunächst noch keine Collections:

```
> show collections
>
```

2.4.2 Dokumente

Dokumente werden innerhalb der Mongo Shell im JSON[4]-Format repräsentiert. Das JSON-Format ist ein leichtgewichtiges, textbasiertes Format, in dem man ein Objekt gut lesbar darstellen kann, z.B.:

```
> doc = { hello: "MongoDB" }
```

Im obigen Beispiel ist `doc` ein Dokument mit einem Feld namens `hello`, das den String-Wert `"MongoDB"` hat.

Wir werden in Kapitel 3 noch genauer auf Collections und Dokumente eingehen.

2.4.3 CRUD-Operationen

Ich werde nun in der Collection `dokumente` einige Dokumente speichern, manipulieren und suchen. Mit

```
> db.dokumente.insert( doc )
> db.dokumente.insert( {a: 1, b: "zwei" } )
> db.dokumente.insert( {a: 2, b: "drei" } )
```

ist das Speichern bereits erledigt. Das JavaScript-Objekt `db` repräsentiert in der Mongo Shell die aktuell selektierte Datenbank. Collections werden automatisch zu Attributen dieses Datenbankobjekts und sind daher über den Ausdruck `db.name` oder `db["name"]` zugreifbar. Mit dem ersten schreibenden Zugriff auf die Collection `dokumente` wurde diese angelegt, wie die Auflistung aller Collections nun zeigt:

```
> show collections
dokumente
system.indexes
```

Mit db.dokumente.help() erhalten Sie Hilfe zu den Befehlen der Collection.

Neben unserer Collection `dokumente` ist auch eine weitere Collection `system.indexes` entstanden, die Informationen über Indizes enthält (mehr dazu in Kap. 6). Wir können nun die zuvor eingefügten Dokumente z.B. zählen oder auch suchen:

4. *http://www.json.org/*

```
> db.dokumente.count()
3
> db.dokumente.find()
{ "_id" : ObjectId("50ba1112b9b2400412cbe6bd"),
    "hello" : "MongoDB" }
{ "_id" : ObjectId("50ba1112b9b2400412cbe6be"),
    "a" : 1, "b" : "zwei" }
{ "_id" : ObjectId("50ba112eb9b2400412cbe6bf"),
    "a" : 2, "b" : "drei" }
```

Dabei fällt auf, dass beim Schreiben ein Feld `_id` erzeugt wurde, mit dem das Dokument eindeutig identifiziert werden kann. Das Feld `_id` wird automatisch ergänzt, wenn Sie es nicht selbst angeben.

Da man oft nicht an allen Dokumenten innerhalb einer Collection interessiert ist, kann man der Methode `find(…)` Suchkriterien und Feldgruppen in Form von Dokumenten mitgeben, um die Treffermenge einzuschränken, z. B.:

```
> db.dokumente.find( {a: 2}, {_id: 0} )
{ "a" : 2, "b" : "drei" }
```

In diesem Beispiel schränken wir mit dem ersten Parameter `{a: 2}` die Menge der gefundenen Dokumente auf solche ein, bei denen das Feld `a` den Wert `2` besitzt. Mit dem optionalen zweiten Parameter schränken wir die Menge der zurückgelieferten Felder innerhalb eines Dokuments ein, indem wir für Felder, die wir ignorieren möchten, eine `0` angeben. (In SQL würden Sie an dieser Stelle eine konkrete Spaltenliste anstatt von * beim SELECT angeben.)

Neben dem einfachen Feldvergleich auf Identität gibt es noch viele weitere sog. Query-Operatoren, die ich ausführlich in Kapitel 6 erklären werde.

Im Gegensatz zu relationalen Datenbanken kann ein einzelnes Feld eines Dokuments aber auch Arrays oder eingebettete Unterobjekte enthalten:

```
> db.autoren.insert( {
    titel: "MongoDB: Ein praktischer Einstieg",
    autoren:[
        {name: "N.N."},
        {name: "Tobias Trelle"}
    ]
} )
```

Eine anschließende Suche mit Formatierung macht die Struktur deutlicher:

```
> db.autoren.find().pretty()
{
"_id" : ObjectId("50ba1bafb9b2400412cbe6c0"),
"titel" : "Praxisbuch MongoDB",
"autoren" : [
    {
    "name" : "N.N"
    },
```

```
        {
        "name" : "Tobias Trelle"
        }
    ]
    }
```

Wenn wir nun ein bestimmtes Objekt verändern wollen, können wir die Methode `update(…)` verwenden:

```
> db.dokumente.update( {a: 2}, {b: "vier"} )
```

Intuitiv würde man nun vielleicht vermuten, dass im Dokument mit `a = 2` der Wert für `b` auf `"vier"` gesetzt wurde. Dies ist aber nicht der Fall, wie eine anschließende Suche zeigt (mit leerem Suchkriterien-Dokument `{}` werden alle Dokumente gefunden):

```
> db.dokumente.find({}, {_id: 0})
{ "a" : 1, "b" : "zwei" }
{ "b" : "vier" }
```

Die Semantik der `update(…)`-Methode ist so, dass der zweite Parameter (hier `{b: "vier"}`) das bestehende Dokument *vollständig ersetzt* und nicht nur die darin vorkommenden Fehler manipuliert!

Partielle Updates und noch einiges mehr sind aber dennoch möglich und werden in Kapitel 7 ausführlich behandelt.

Einzelne Dokumente werden mit `remove(…)` gelöscht:

```
> db.dokumente.remove( {a: 1} )
> db.dokumente.find({}, {_id: 0})
{ "b" : "vier" }
```

Um alle Dokumente einer Collection zu löschen, ruft man `remove()` ohne Parameter auf. Dies kann bei großen Datenmengen sehr lange dauern, da jedes Dokument einzeln gelöscht wird, was z. B. auch eine Aktualisierung von Indizes auslösen kann. Schneller ist man hier mit:

```
> db.dokumente.drop()
true
> show collections
system.indexes
```

Mit `drop()` wird die Collection vollständig gelöscht, inklusive aller bestehenden Indizes.

2.5 Daten importieren

Die bisherigen Beispiele mögen Ihnen zu Recht trivial vorkommen. Im weiteren Verlauf des Buchs werden wir auf eine größere und komplexere Datenmenge zurückgreifen. Es handelt sich um einen Auszug aus öffentlich zugänglichen Tweets der Twitter-Plattform. Diese Daten können Sie von

```
https://github.com/ttrelle/mongodb-buch/blob/master/data/tweets.zip?raw=true
```

herunterladen und ins Verzeichnis `MONGO_HOME` entpacken. Der Inhalt des Archivs ist eine Datei im BSON-Format, die gut 50.000 Dokumente beinhaltet, die wir gleich in eine Collection einspielen werden.

Beenden Sie dazu nun die Mongo Shell mit dem Befehl `exit`. Anschließend rufen Sie im Verzeichnis `MONGO_HOME\bin` den folgenden Befehl auf, um die Daten in eine Datenbank namens `twitter` in eine Collection namens `tweets` zu importieren:

```
$ mongorestore -d twitter../tweets.bson
connected to: 127.0.0.1
2014-02-27T08:30:07.181+0100 tweets.bson
2014-02-27T08:30:07.185+0100    going into namespace [twitter.tweets]
2014-02-27T08:30:07.187+0100 tweets.metadata.json not found. Skipping.
2014-02-27T08:30:10.002+0100           Progress: 76560817/85711256    89%
(bytes)
53641 objects found
```

Die Ausführungszeit des Importvorgangs kann in Abhängigkeit von Ihrem System variieren. Sobald die Daten importiert sind, rufen wir wieder die Mongo Shell auf und wechseln in die Twitter-Datenbank, um uns die Anzahl der Tweets anzuzeigen:

```
$ mongo
MongoDB shell version: 2.x
connecting to: test
> use twitter
switched to db twitter
> show collections
system.indexes
tweets
> db.tweets.count()
53641
```

Wenn wir nun mittels

```
> db.tweets.find()
```

nach allen importierten Tweets suchen, gibt die Mongo Shell einen Wust von Dokumenten aus. Die Anzahl der angezeigten Dokumente wird per Default auf 20 beschränkt. Diese Anzahl wird bestimmt durch:

```
> DBQuery.shellBatchSize
20
```

Um die jeweils nächste Tranche der Ergebnismenge anzuzeigen, geben Sie bitte den Befehl `it` ein. Um Dokumente lesbarer anzuzeigen, verwenden Sie diesen Befehl:

```
> db.tweets.find().pretty()
```

Probieren Sie einfach mal ein paar Suchen nach verschiedenen Kriterien aus. Die Hilfe zur `find()`-Methode unterstützt Sie dabei.

```
> db.tweets.find().help()
```

Alle weiteren Details zu Queries und auch zur Indexierung finden Sie in Kapitel 6.

2.6 Schemafreiheit

Eines der Alleinstellungsmerkmale von MongoDB ist die sogenannte *Schemafreiheit*, auf die ich in Kapitel 8 noch genauer eingehen werde. Zum Einstieg will ich Ihnen anhand eines einfachen Beispiels die Möglichkeiten aufzeigen, die Sie ohne ein starres Schema haben.

Sie können z. B. einen Produktkatalog mit den unterschiedlichsten Produkttypen innerhalb *einer* Collection realisieren:

```
> use shop
switched to db shop
> db.produkte.insert({
  typ: "dvd",
  titel: "Lost Highway",
  regie: "David Lynch"
})
> db.produkte.insert({
  typ: "cd",
  titel: "Highway to Hell",
  band: "AC/DC"
})
```

Die Suche nach gemeinsamen Feldern der unterschiedlichen Produkte ist ganz einfach:

```
> db.produkte.find({titel: /Highway/})
{     "_id" : ObjectId("530eefa913109e3002c125de"),
      "typ" : "dvd",
      "titel" : "Lost Highway",
      "regie" : "David Lynch" }
{     "_id" : ObjectId("530eefe313109e3002c125df"),
      "typ" : "cd",
      "titel" : "Highway to Hell",
      "band" : "AC/DC" }
```

Eine Einschränkung auf bestimmte Produktarten ist dabei möglich:

```
> db.produkte.find({titel: /Highway/, typ: "cd"})
...
```

Das gilt ebenso für die Suche nach Feldern, die es nur für bestimmte Produkte gibt:

```
> db.produkte.find({regie: "David Lynch"})
...
```

Das Beispiel ist zugegebenermaßen trivial, deutet aber die Mächtigkeit an, die die Schemafreiheit beim Umgang mit semistrukturierten Daten bietet. Hier kristallisiert sich einer der Hauptvorteile von MongoDB heraus, der insbesondere auch den Big-Data-Aspekt *Variety* adressiert.

2.7 Zusammenfasssung

In diesem Kapitel haben Sie gesehen, wie einfach der Einstieg in MongoDB ist. Nach den ersten Schritten haben Sie bereits Daten importiert und die Vorteile der Schemafreiheit kennengelernt. Im nächsten Kapitel werde ich Ihnen die technologischen Grundlagen für all das näher erklären.

Selber machen!

Ergänzen Sie den Produktkatalog um weitere Produkte und verwenden Sie dabei noch weitere Produkttypen, die sich strukturell deutlich von den bisherigen unterscheiden.