

Spätestens, wenn Sie bei der Erweiterung einer Tabelle um eine Spalte mit *ALTER TABLE* zu bangen beginnen, wie lange Ihre Applikation dieses Mal eingeschränkt oder gar nicht verfügbar ist, werden Sie sich eine Alternative zu dem starren Datenschema relationaler Datenbanken wünschen. MongoDB, eine NoSQL-Dokumentdatenbank, bietet hier eine interessante und ausgereifte Alternative.

Dokumentdatenbanken gruppieren die Daten in einem strukturierten Dokument, typischerweise in einer JSON-Datenstruktur. Auch MongoDB verfolgt diesen Ansatz und bietet darauf aufbauend eine reichhaltige Abfragesprache und Indexe auf einzelne Datenfelder. Die Möglichkeiten der Replikation und des Shardings zur stufenlosen und unkomplizierten Skalierung der Daten und Zugriffe macht MongoDB auch für stark frequentierte Websites äußerst interessant.

MongoDB bietet zudem auch **Map/Reduce** zur Datenverarbeitung an. Map/Reduce ist ein Kernbaustein zur Verarbeitung großer Datenmengen (auch in anderen NoSQL-Datenbanken), da mit diesem Ansatz die Berechnungen auf einfachem Wege auf viele Server zur parallelen Berechnung verteilt werden können. Mehr dazu erfahren Sie in Abschnitt 14.6.

■ 14.1 Installation

Für MongoDB gibt es auf <http://www.mongodb.org/downloads> fertige Binärpakete für Windows, Linux, OS X und Solaris. Die Pakete müssen nur entpackt werden und sind erfreulicherweise ohne großen Konfigurationsaufwand sofort lauffähig. Um MongoDB zu starten, müssen Sie nur ein geeignetes Verzeichnis für die Datenbankdateien auswählen und *mongod* mit folgenden Optionen starten.

```
./mongod --dbpath /path/to/my/db/data
```

Mit *mongod --help* können Sie weitere Kommandozeilenoptionen in Erfahrung bringen. Unter Linux können Sie MongoDB auch über maßgeschneiderte Pakete der jeweiligen Distribution installieren. Diese enthalten meist auch schon vorgefertigte Konfigurationsdateien und Skripts, um MongoDB als Service automatisch zu starten.

14.1.1 Mongo-Shell

Die MongoDB-Installation enthält die interaktive Shell *mongo* (siehe Bild 14.1), mit der Sie direkt in MongoDB Daten suchen, anlegen, ändern und löschen können und auch Administrationsaufgaben durchführen können. Die Shell ist eine vollständige JavaScript-Umgebung, Sie können also auf alle Standardfunktionen von JavaScript zurückgreifen.

Die Shell ermöglicht mit den Methoden *help* (allgemeine Hilfe), *db.help()* (Hilfe zu Datenbanken) und *db.kollektion.help()* (Hilfe zu Kollektionen) einen raschen Einstieg in die Shell-Kommandos.



Bild 14.1 Mongo-Shell

14.1.2 Installation der PHP-Erweiterung

Die PHP-Erweiterung zu MongoDB ist über PECL verfügbar. Die Installation erfolgt mit:

```
pecl install mongo
```

Alternativ können Sie auch von <http://github.com/mongodb/mongo-php-driver> das aktuelle Quellpaket der Erweiterung herunterladen und selber kompilieren. Wenn Sie das Paket entpackt und in das neue Verzeichnis gewechselt haben, können Sie den Treiber mit folgenden Befehlen installieren.

```
phpize
./configure
make
make install
```

Anschließend müssen Sie noch die Erweiterung in die PHP-Konfiguration einbinden, indem Sie der Konfiguration die Zeile `extension=mongo.so` hinzufügen. Nach einem Neustart des Webservers sehen Sie beim Aufruf von *phpinfo()* die Daten der Erweiterung wie in Bild 14.2 und können mit der Programmierung beginnen.

MongoDB Support		enabled
Version		1.2.10

Directive	Local Value	Master Value
mongo.allow_empty_keys	0	0
mongo.allow_persistent	1	1
mongo.auto_reconnect	1	1
mongo.chunk_size	262144	262144
mongo.cmd	\$	\$
mongo.default_host	localhost	localhost
mongo.default_port	27017	27017
mongo.is_master_interval	60	60
mongo.long_as_object	0	0
mongo.native_long	0	0
mongo.no_id	0	0
mongo.ping_interval	5	5
mongo.utf8	1	1

Bild 14.2 Ausgabe von phpinfo() nach der Installation der MongoDB-Erweiterung

14.2 Datenbanken und Datenschema

MongoDB ist eine dokumentorientierte, schemalose Datenbank, in der Datenobjekte (Dokumente) in Kollektionen zusammengefasst werden, die ihrerseits in einer Datenbank liegen. Kollektionen sind analog zu Tabellen in relationalen Datenbanken zu sehen, nur dass sie ohne Schema sind und beliebige Daten enthalten können. Es ist also nicht notwendig – wie bei relationalen Datenbanken – ein Schema mit *CREATE TABLE* anzulegen, bevor Daten eingefügt werden können. MongoDB geht sogar so weit, dass nicht einmal die Datenbank angelegt werden muss. Eine Datenbank wird bei der ersten Verwendung automatisch angelegt, Kollektionen werden mit dem ersten Objekt, das eingefügt wird, erzeugt.

14.2.1 Verbinden und Auswahl der Datenbank

Nachdem Sie MongoDB installiert und den MongoDB-PHP-Treiber eingebunden haben, können Sie sich von PHP aus zur Datenbank verbinden, indem Sie ein neues Objekt der *Mongo*-Klasse instanziiieren.

```
$mng = new Mongo('mongodb://127.0.0.1:27017');
```

MongoDB verwendet standardmäßig die Portnummer 27017. Wenn Sie sich zu MongoDB am lokalen Rechner auf dem Standardport verbinden wollen, können Sie die Verbindungsangabe weglassen.

```
$mng = new Mongo(); // Verbindung zu localhost:27017
```

Auswählen und Erzeugen einer Datenbank

Zum Auswählen einer Datenbank benutzen Sie die Methode *Mongo::selectDB()*, alternativ können Sie den Namen direkt wie ein Attribut des *Mongo*-Objektes ansprechen:

```
Methode: $db = $mng->selectDB('styleShop');
Attribut: $db = $mng->styleShop;
```

Achtung: Wenn die Datenbank noch nicht existiert, wird sie automatisch angelegt! Überprüfen Sie daher den Namen der Datenbank beim Verbinden genau, um nicht durch einen Tippfehler eine neue Datenbank anzulegen.

In der Mongo-Shell wählen Sie eine Datenbank mit `use <dbname>` aus, zum Beispiel `use styleShop`. Auch beim Start der Shell können Sie einen Datenbanknamen als Parameter übergeben.

14.2.2 Auswahl und Erzeugen von Kollektionen

Gleich wie bei der Datenbank, wählen Sie eine Kollektion aus, indem Sie ein neues *MongoCollection*-Objekt instanziiieren bzw. über *MongoDB::selectCollection()* auswählen. Sie können auch hier eine vereinfachte Attributschreibweise verwenden.

```
Methode: $coll = $db->selectCollection('produkte');
Instanzieren: $coll = new MongoClient($db, 'produkte');
Attribut: $coll = $mng->styleShop;
```

Im Gegensatz zur Datenbank wird die Kollektion aber nicht durch das Auswählen automatisch angelegt, sondern erst, wenn Sie das erste Dokument in die Kollektion einfügen.

In der Mongo-Shell wählen Sie die Kollektion nicht direkt aus, sondern alle Befehle für Kollektionen haben das Format `db.<kollektion>.<befehl>`, zum Beispiel `db.produkte.storageSize()`, um den Speicherbedarf einer Kollektion zu ermitteln.

Begrenzte Kollektionen (Capped Collections)

MongoDB hat als Besonderheit begrenzte Kollektionen (*Capped Collections*) anzubieten, die nur eine fest definierte Anzahl an Dokumenten enthalten können. Werden mehr Dokumente eingefügt, dann werden die ältesten Einträge durch die neuen Dokumente ersetzt. Begrenzte Kollektionen müssen explizit mit *MongoDB::createCollection()* angelegt werden.

```
$ccoll = $db->createCollection('rotatingLog', true, 1024*1024, 1000)
```

Der zweite Parameter (*true*) gibt an, dass es sich um eine Kollektion mit fixer Größe handeln soll, der dritte Parameter gibt die Maximalgröße der Kollektion in Bytes an, der vierte Parameter die maximale Anzahl der Dokumente in der Kollektion.



PRAXISTIPP: Begrenzte Kollektionen eignen sich optimal zum Protokollieren von Ereignissen, Status- und Debug-Meldungen sowie für Listen der Art „die 10 neuesten Einträge“.

Aufgrund der internen Struktur sind begrenzte Kollektionen äußerst schnell beim Lesen und Schreiben, unterliegen aber der Einschränkung, dass eingefügte Dokumente nicht gelöscht werden dürfen und Aktualisierungen, die Dokumente vergrößern, nicht durchgeführt werden können.

14.2.3 Dokumente und Datentypen

Daten werden in MongoDB in BSON-Dokumenten gespeichert. BSON (<http://bsonspec.org>) ist ein an JSON (siehe Abschnitt 5.1) angelehntes binäres Datenformat. Wie bei JSON sind Dokumente verschachtelte Arrays und Objekte (= assoziative Arrays), BSON kennt aber mehr Datentypen (siehe Tabelle 14.1), und die Objektwurzel ist immer ein Dokument, nie ein einfacher Datentyp.



HINWEIS: MongoDB beschränkt die Größe der BSON-Dokumente aus Effizienzgründen auf 16 MB. Größere Dokumente lassen sich nur über Umwege des GridFS (siehe <http://www.mongodb.org/display/DOCS/GridFS>) speichern. Mit GridFS werden Metadaten und Inhaltsdaten in zwei Kollektionen aufgeteilt, zudem werden die Inhaltsdaten in einzelne Chunks aufgeteilt, damit beispielsweise eine große Binärdatei stückweise ausgelesen werden kann.

Tabelle 14.1 BSON-Datentypen

Datentyp	Beschreibung
<i>Integer</i>	BSON kennt 32-Bit- und 64-Bit-Integerzahlen.
<i>Array</i>	In BSON sind Arrays als Dokumente (assoziative Arrays) umgesetzt, deren Schlüssel die Array-Indexpositionen (als Integerzahlen) sind.
<i>Binary</i>	Binärdaten können in BSON ohne spezielle Kodierung gespeichert werden.
<i>Boolean</i>	Werte: <i>true</i> / <i>false</i>
<i>DateTime</i>	Speichert das Datum und die Uhrzeit in Millisekunden-Auflösung, intern als 64-Bit-Zahl seit der Unix-Epoche (01.01.1970) umgesetzt. Gilt als UTC-Zeitpunkt, obwohl man es auch für lokale Zeitzonen verwenden kann.
<i>Document</i>	Das Dokument ist der Wurzeldatentyp jedes BSON-Objektes. Es entspricht einem JavaScript/JSON-Objekt bzw. assoziativem Array. Dokumente können auch verschachtelt und ineinander eingebettet werden.
<i>Double</i>	Gleitkommazahlen mit doppelter Genauigkeit

Tabelle 14.1 BSON-Datentypen (Fortsetzung)

Datentyp	Beschreibung
<i>JavaScript</i>	Für JavaScript-Programme gibt es einen eigenen Datentyp, um sie von einfachen Strings zu unterscheiden.
<i>Null</i>	<i>Null</i> stellt wie bei Datenbanken oder in PHP das Fehlen jedweden Wertes dar.
<i>ObjectId</i>	Dokumente können in MongoDB eine speziell strukturierte ID haben, die auch einen Zeitstempel enthält (siehe Abschnitt 14.3.3).
<i>Regulärer Ausdruck</i>	Für reguläre Ausdrücke gibt es ebenfalls einen eigenen Datentyp, um sie von einfachen Strings zu unterscheiden.
<i>string</i>	Zeichenfolgen werden in BSON immer als UTF-8 kodiert, andere Kodierungen sind nicht vorgesehen.
<i>Symbol</i>	Symbole haben einen eigenen Datentyp, da manche Programmiersprachen zwischen Symbolen und Strings unterscheiden.

JavaScript-Syntax

In der Mongo-Shell werden BSON-Dokumente wie in JavaScript geschrieben. Dokumente mit geschwungenen Klammern und Arrays mit eckigen Klammern sowie Attributnamen können ohne Anführungszeichen sein. Werte können in doppelten oder einfachen Anführungszeichen stehen, wie in Listing 14.1 zu sehen. Spezielle Datentypen wie die *ObjectId* oder *DateTime* werden mit einer Funktion initialisiert, JavaScript-Funktionen (im Beispiel: *rabatt*) können direkt als Wert eingefügt werden.

Listing 14.1 BSON-Dokument in der JavaScript-Syntax der Mongo-Shell

```
{ "_id" : ObjectId("4fd375c90c41ebac1967de97"),
  produktnr : 820108,
  'beschreibung' : "Krawatte mit Muster, rot, handbemalt",
  kategorie : [ 'elegant', 'sommer', 'hochzeit' ],
  herkunft: { ort : "Veitsch",
              land: 'Österreich' },
  preis:39.9,
  bestell_datum: ISODate("2012-06-09T17:00:46Z"),
  rabatt: function (p) { return p*0.8 }
}
```

PHP-Syntax

In PHP wird das BSON-Objekt aus Listing 14.1 wie in Listing 14.2 geschrieben (unter Verwendung der neuen PHP 5.4-Array-Syntax). Für Dokumente werden in PHP assoziative Arrays verwendet, spezielle Datentypen werden über eine eigene Klasse abgebildet (zum Beispiel *MongoId*, *MongoDate*, *MongoCode*, *MongoRegex*).

Listing 14.2 BSON-Dokument aus PHP-Sicht

```
$doc = [ '_id' => new MongoId("4fd375c90c41ebac1967de97"),
        'produktnr' => 820108,
        'beschreibung' => "Krawatte mit Muster, rot, handbemalt",
```

```

'kategorie' => [ 'elegant', 'sommer', 'hochzeit' ],
'herkunft' => [ 'ort' => "Veitsch",
               'land' => 'Österreich' ],
'preis' => 39.9,
'bestell_datum' => new MongoDate(strtotime("2012-06-09T17:00:46")),
'rabatt' => new MongoCode("function (p) { return p*0.8 }")
];

```

14.2.4 Entwerfen des Datenschemas

Ein Datenschema für eine dokumentorientierte Datenbank wie MongoDB folgt anderen Prinzipien als das Datenschema einer relationalen Datenbank. Während Tabellen und Kollektionen sich in etwa entsprechen, gibt es in MongoDB kein *JOIN*. Eine normalisierte Datenhaltung ohne Redundanzen hat daher viele Abfragen zur Folge. Auch gibt es bei MongoDB weder atomare Operationen über Dokumentengrenzen hinweg noch Transaktionen, dafür können Sie aber in MongoDB Dokumente ineinander verschachteln.

Das Design eines Datenschemas wird vor allem vom Verhältnis der Lese- und Schreibzugriffe geleitet. Wenn die Lesezugriffe ein Vielfaches der Schreibzugriffe sind, sollte das Schema für den Lesezugriff optimiert sein, was typischerweise eine Denormalisierung der Daten bedeutet. Sind Lese- und Schreibzugriffe ausgewogen, bietet sich eher ein normalisiertes Datenschema an.

Eingebettete Dokumente

Die Verschachtelung von Dokumenten bietet sich dann an, wenn zwischen den Dokumenten eine *besteht-aus*-Beziehung bzw. eine *ist*- oder *hat*-Beziehung besteht. Listing 14.3 zeigt ein Beispiel für ein Produktdokument. In einem relationalen, normalisierten Datenschema würde die Zuordnung zu den Kategorien über eine eigene Tabelle abgebildet werden, auch die Lagerinformation *Lagerort* würde in eine eigene Tabelle ausgelagert sein.

Die gezeigte Dokumentstruktur bietet den Vorteil, dass beispielsweise im Webshop die Kategorien ohne zusätzlichen Lesezugriff mitangezeigt werden können und die Lagerinformation für die Bearbeitung der Bestellung unmittelbar bereitsteht. Abfragen nach allen Produkten einer bestimmten Kategorie sind dank der Möglichkeit, verschachtelte Dokumente und Arrays zu indizieren, trotzdem effizient möglich.

Listing 14.3 Eingebettete Arrays und Dokumente

```

{ _id: 'produkt-36918',
  titel: 'Russische Schapka, schwarz',
  preis: 69.9,
  kategorie: ['winter', 'mütze', 'warm', 'kaninchenfell'],
  lagerbestand: 182,
  lager: [ { anzahl: 120, zone: '12b', regal: 1223, ebene: 4 },
           { anzahl: 60, zone: '12b', regal: 1223, ebene: 3 },
           { anzahl: 2, zone: '12a', regal: 1218, ebene: 1 } ]
}

```

Die in Listing 14.3 gezeigte Struktur bietet auch den Vorteil, den Lagerbestand und die Anzahl für das entsprechende Lagerregal gemeinsam als atomare Operation zu erhöhen oder zu erniedrigen, da beide Zahlen im gleichen Dokument – dem Produkt – sind.

Wann sollten Sie besser Dokumente besser nicht einbetten? Einerseits, wenn es sich bei den eingebetteten Daten um eigenständige Datenobjekte erster Klasse handelt. Benutzerdaten in andere Objekte einzubetten, macht beispielsweise wenig Sinn. Andererseits, wenn Sie zusätzliche Flexibilität benötigen, denn eigenständige Dokumente können leichter referenziert werden und bieten eine breitere Palette an Abfrage- und Kombinationsmöglichkeiten. Wenn die Anzahl der einzubettenden Dokumente sehr groß wird, sollten Sie ebenfalls eher auf getrennte Kollektionen ausweichen.

Beispiel: Kommentare zu Einträgen

Ein häufig genanntes Beispiel sind Kommentare zu Blog-Einträgen, Produkten etc. Sollen Kommentare in einer eigenen Kollektion gespeichert werden oder direkt dem Produkt zugeordnet werden, wie in Listing 14.4 zu sehen?

Listing 14.4 Eingebettete Kommentare

```
{ _id: 'produkt-36918',
  titel: 'Russische Schapka, schwarz', preis: 69.9,
  kommentare: [{ benutzer: 'Christina', text: 'Super warm und praktisch' },
                { benutzer: 'Max', text: 'Hat genau die passende Farbe' },
                { benutzer: 'Eva', text: 'Ein Must in dieser Saison' }]
}
```

Die Antwort hängt von Ihrer Anwendung ab: Sind die Kommentare – wie in einem Webshop – immer einem Produkt zugeordnet und werden nicht als eigenständige Objekte verwendet? Rechnen Sie nicht mit mehr als 100 Kommentaren pro Produkt?

Wenn Sie beide Fragen mit Ja beantworten, dann sollten Sie die Kommentare einbetten. Sind aber in Ihrer Anwendung Kommentare Datenobjekte erster Klasse, und bieten Sie viele Sonderfunktionen für Kommentare an, dann sollten Sie die Kommentare in eine eigene Kollektion auslagern. Auch wenn Sie viele Kommentare pro Produkt erwarten, sollten Sie vom Einbetten Abstand nehmen. Dann wird nämlich das Produktdokument sehr groß (und stößt vielleicht sogar an das Größenlimit für Dokumente), was zur Folge hat, dass für alle Such- und Abfrageoperationen ein großes Dokument in den Arbeitsspeicher transferiert wird, selbst wenn die Kommentare gar nicht benötigt werden. Das kann letztlich dazu führen, dass Sie Leistungseinbußen hinnehmen müssen.

Felder im Voraus berechnen

MongoDB ist für die Analyse und Aggregation von Daten gut geeignet, allerdings sollten Sie – so weit möglich – auf die Berechnung von Inhalten zum Abfragezeitpunkt verzichten. Es besteht zwar die Möglichkeit, serverseitiges JavaScript in MongoDB auszuführen, allerdings ist das im Vergleich zum Zugriff auf ein (möglicherweise indiziertes) Feld wesentlich langsamer. Im Voraus berechnete Felder benötigen meist nicht viel Speicherplatz, erlauben aber im Gegenzug, dass mit allen gewöhnlichen Operatoren darauf zugegriffen werden kann.

Listing 14.5 zeigt ein Beispieldokument zum Protokollieren von Zugriffsstatistiken. Das Dokument deckt einen Zeitraum von 24 Stunden ab und enthält als zusätzliches Feld auch die

Summe der Zugriffe im genannten Zeitraum; dieses *gesamt*-Feld ist redundant und kann jederzeit aus dem *stunden*-Array berechnet werden. Wenn Sie aber rasch wissen wollen, welche die populärsten URLs an einem bestimmten Tag waren, ist das Feld die beste Lösung, da Sie es auch leicht mit einem Index versehen können. Da das *gesamt*-Feld im selben Dokument wie der Stundenzähler ist, können die Werte auch synchron und atomar hochgezählt werden.

Listing 14.5 Dokument für Zugriffsstatistiken

```
{ url: "http://shop.xmp.site/produkt/36918/schapka-schwarz",  
  gesamt: 849,  
  stunden: [ 12, 17, 33, ... 120, 70, 17] // 24 Einträge  
}
```

Resümee

Eine dokumentorientierte Datenbank, wie MongoDB, bietet wesentlich mehr Spielraum zur Datenmodellierung als eine relationale Datenbank. Generell sollten Sie von der Möglichkeit Gebrauch machen, Dokumente ineinander zu verschachteln. Dadurch erreichen Sie eine größere Lokalität der Daten und können viele Anfragen mit einem einzigen Dokument beantworten. Achten Sie aber darauf, dass primäre Datenobjekte in einer eigenen Kollektion gespeichert werden sollen, und dass die Dokumente in Summe nicht zu groß werden.

Obwohl eine schemafreie Datenbank das Mischen unterschiedlicher Dokumenttypen in einer Kollektion zulässt, sollten sie Kollektionen möglichst homogen gestalten. Das Anlegen einer weiteren Kollektion ist billig und benötigt wenige Ressourcen.

Versuchen Sie, Dokumente, so weit als möglich, als abgeschlossene Datenmengen zu definieren. Insbesondere sollten Sie Felder, die sich häufig im Gleichschritt ändern, im selben Dokument unterbringen, da Sie so einfach eine atomare Aktualisierung der Felder umsetzen können.

Letztlich sollten Sie sich vor allem an den Suchabfragen Ihrer Anwendung orientieren, um ein maßgeschneidertes Datenmodell für Ihre spezifischen Anforderungen zu erhalten. In den meisten Fällen sollten Sie auch nicht zu weit vorausplanen und vorab unnötige Komplexität in das Datenmodell einbringen. Anforderungen wandeln sich rasch, und die Struktur einer dokumentorientierten Datenbank lässt sich leicht ändern. Lassen Sie sich beim Design vom YAGNI-Prinzip leiten (*You Ain't Gonna Need It*, siehe <http://c2.com/cgi/wiki?YouArentGonnaNeedIt>).

■ 14.3 CRUD-Operationen

Die CRUD-Operationen zum Erzeugen (*Create*), Lesen (*Read*), Aktualisieren (*Update*) und Löschen (*Delete*) von Dokumenten sind das Basiswerkzeug, um Daten in MongoDB zu verwalten. Die dafür vorhandenen Funktionen sind allesamt Methoden auf Ebene der Kollektionen, denn Dokumente werden in Kollektionen gespeichert. Wie bereits in Abschnitt 14.2.2 erwähnt, wird eine neue Kollektion angelegt, sobald das erste Dokument in die Kollektion eingefügt wird.

14.3.1 Einfügen von Dokumenten

Zum Einfügen neuer Dokumente stehen Ihnen in MongoDB die Befehle `db.<kollektion>.insert()` (in PHP `MongoCollection::insert()`) und `db.<kollektion>.batchInsert()` (in PHP `MongoCollection::batchInsert()`) zur Verfügung. Letzteren Befehl können Sie dazu verwenden, mehrere Dokumente mit einem einzigen Befehl einzufügen. Listing 14.6 zeigt ein typisches Szenario. Die zu speichernden BSON-Dokumente werden als PHP-Arrays erzeugt, die MongoDB-PHP-Erweiterung wandelt die Datenstruktur passend um.

Listing 14.6 Einfügen von neuen Dokumenten mit `insert()` und `batchInsert()`

```
<?php
$mng = new Mongo('mongodb://127.0.0.1:27017');
$db = $mng->selectDB('styleShop');
$coll = $db->selectCollection('produkte');

$produkte = [
    [ 'typ' => 'hut', 'farbe' => 'blau', 'preis' => 49.9,
      'beschreibung' => 'Hut für sonnige Tage' ],
    [ 'typ' => 'hut', 'kategorie' => [ 'Freizeit', 'Sommer' ],
      'preis' => 24.9, 'beschreibung' => 'Modischer Freizeithut' ],
    [ 'typ' => 'hut', 'kategorie' => [ 'Freizeit', 'Winter' ],
      'material' => 'Kaninchenfell', 'preis' => 29.9,
      'beschreibung' => 'Russische Schapka, sehr warm' ]];
$coll->batchInsert($produkte);

$produkt = [ 'typ' => 'handtasche', 'material' => 'Wolle, geknüpft',
              'herstellung' => 'Handarbeit, Steiermark',
              'beschreibung' => 'Handtasche für jeden Anlass' ];
$coll->insert($produkt);
?>
```

Bild 14.3 zeigt das Ergebnis der Einfügeoperationen. MongoDB erstellt für jedes Dokument automatisch einen primären Schlüssel und speichert diesen ins Feld `_id`. Mehr zu primären Schlüsseln in Abschnitt 14.3.3. Vorher wenden wir uns noch der Fehlerbehandlung zu.

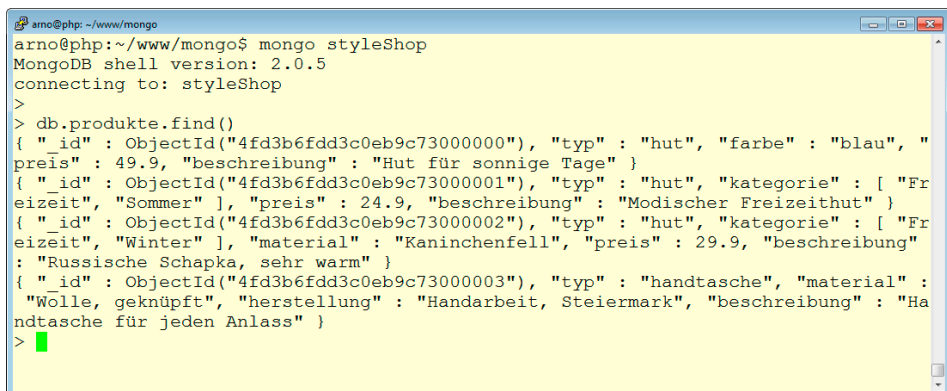


Bild 14.3 Eingefügte Dokumente in der Kollektion „produkte“

14.3.2 Fehlerbehandlung

Es ist zu Beginn etwas ungewohnt, dass MongoDB die Einfügeoperation asynchron durchführt. Sie bekommen nach dem Einfügen keine Statusmeldung über Erfolg oder Misserfolg zurück. Diesem Verhalten liegen zwei Gedanken zugrunde: Einerseits ist das asynchrone Einfügen wesentlich schneller, die Performanz Ihrer Anwendung erhöht sich. Andererseits geht MongoDB davon aus, dass einige Datensätze durchaus verloren gehen können. Je nachdem, um welche Daten es sich dabei handelt, mag diese Annahme auch für Ihre Anwendung zutreffen. Wenn Sie MongoDB zum Beispiel zur Protokollierung einsetzen, dann macht es im Regelfall nichts aus bzw. verfälscht die Statistiken nicht, wenn einige Datensätze nicht gespeichert worden sind.

Wenn Sie aber Daten in MongoDB speichern, die nicht verloren gehen dürfen, dann sollten Sie die sichere Variante des Einfügens von Dokumenten verwenden. MongoDB stellt dafür den Befehl `db.getLastError()` zur Verfügung, mit dem das Ergebnis der letzten Operation abgefragt werden kann. In PHP gibt es zwei Varianten, wie Sie `db.getLastError()` verwenden können: Über die Methode `MongoDB::lastError()` oder über die `safe`-Option direkt beim Einfügen.

Abfrage über `MongoDB::lastError()`

Listing 14.7 zeigt die Fehlerabfrage mithilfe von `MongoDB::lastError()`. Die Methode muss aufgerufen werden, bevor weitere Befehle an MongoDB gesendet werden, da ansonsten der Status des neuen Befehls den vorherigen Status überschreibt.

Listing 14.7 Fehlerabfrage mit `MongoDB::lastError()`

```
$db = $mng->selectDB('styleShop');
$coll = $db->selectCollection('produkte');
$coll->insert($document);
$error = $db->lastError();
if (isset($error['code']) || !$error['ok']) {
    // Fehler aufgetreten
}
```

Die Felder des von `MongoDB::lastError()` retour gelieferten Arrays sind in Tabelle 14.2 aufgeführt. Da auch der Befehl `db.getLastError()` selbst fehlschlagen kann, sollte auch das Feld `$error['ok']` immer mit überprüft werden.

Tabelle 14.2 Felder des Fehlerarrays von `MongoDB::lastError()`

Feld	Beschreibung
<code>\$error['err']</code>	<code>NULL</code> , wenn kein Fehler aufgetreten ist, ansonsten ein String mit Beschreibung des Fehlers
<code>\$error['code']</code>	Enthält einen numerischen Fehlercode, falls ein Fehler aufgetreten ist, ansonsten ist das Feld nicht vorhanden
<code>\$error['n']</code>	Anzahl der Dokumente, die in einer <code>update</code> -Operation verändert worden sind; für <code>insert/remove</code> immer Null
<code>\$error['connectionId']</code>	ID der Verbindung zu MongoDB

Tabelle 14.2 Felder des Fehlerarrays von `MongoDB:lastError()` (Fortsetzung)

Feld	Beschreibung
<code>\$error['ok']</code>	Gibt an, ob der Befehl <code>getLastError()</code> erfolgreich ausgeführt wurde (<i>true</i>) oder bei der Ausführung Probleme aufgetreten sind (<i>false</i>). In letzterem Fall kann über das Ergebnis der vorherigen Operation nichts ausgesagt werden, weshalb Sie auch diesen Wert in die Überprüfung miteinbeziehen sollten.

Safe-Option beim Einfügen

Alternativ zur Abfrage mit `MongoDB::lastError()` können Sie auch direkt beim Einfügen die *safe*-Option angeben. In diesem Fall wird aus dem asynchronen Einfügen eine synchrone Operation, die auf die Antwort von MongoDB wartet. Tritt ein Fehler auf, wird eine Exception geworfen.

Listing 14.8 zeigt den Aufruf einer sicheren Einfügeoperation. `MongoCollection::insert()` können drei Optionen übergeben werden: *safe*, um die Antwort von MongoDB abzuwarten, *fsync*, falls Sie zusätzlich festlegen wollen, dass Ihre Daten unmittelbar in die Datenbankdatei geschrieben werden, und eine maximale Zeitdauer (in Millisekunden) mit *timeout*.

Listing 14.8 Ausführen einer sicheren Einfügeoperation

```
try {
    $coll->insert($document, array('safe' => true, 'fsync' => false,
                                   'timeout' => 500));
}
catch (MongoCursorTimeoutException $e) {
    // Zeitüberschreitung
}
catch (MongoCursorException $e) {
    // Fehler beim Einfügen
}
?>
```

Über das *Exception*-Objekt können Sie im Fehlerfall mit den Methoden *Exception::getCode()* und *Exception::getMessage()* Fehlercode und -beschreibung abfragen. Über das Attribut `MongoCursorException::doc` ist auch das Array aus Tabelle 14.2 abrufbar.



PRAXISTIPP: Auch der *Safe*-Modus verwendet intern den Befehl `db.getLastError()`. Aus technischer Sicht ist also die Verwendung von `MongoDB::lastError()` und der sicheren Einfügeoperation identisch. Die Auswahl zwischen den beiden Methoden bleibt Ihrem persönlichen Programmierstil überlassen.

14.3.3 Primäre Schlüssel

MongoDB erstellt automatisch primäre Schlüssel, wenn neue Dokumente in die Datenbank eingefügt werden, und speichert diese ins Feld `_id`, wie in Abschnitt 14.3.1 zu sehen war. Der erzeugte Schlüssel ist 12 Bytes lang und setzt sich aus einem Zeitstempel, einem Hash der Client-Adresse, der Nummer des MongoDB-Prozesses sowie einer fortlaufenden Nummer zusammen. Diese Zusammensetzung ergibt eindeutige Schlüssel, auch wenn viele MongoDB-Instanzen zusammenarbeiten.

Ein angenehmer Nebeneffekt dieser Struktur ist, dass über den Schlüsselwert auch der Zeitpunkt der Erstellung des Dokumentes abgefragt werden kann, ein eigenes Feld mit Zeitstempel ist damit überflüssig. Zur Abfrage wird die Methode `getTimestamp()` der PHP-Klasse *MongoId* verwendet, wie in Listing 14.9 zu sehen.

Listing 14.9 Abfrage des Erstellungszeitpunktes einer MongoId

```
<?php
$id = new MongoId('4fcd18acd4c0eb705a000001');
echo "Id: $id\n";
echo "Timestamp: ", gmdate(DATE_RFC2822, $id->getTimestamp()), "\n";
?>
```

Listing 14.10 zeigt das Ergebnis der Abfrage.

Listing 14.10 Enthaltener Zeitstempel der MongoId

```
> php id-time.php
Id: 4fcd18acd4c0eb705a000001
Timestamp: Mon, 04 Jun 2012 20:21:00 +0000
```



PRAXISTIPP: Da der Zeitstempel der erste Bestandteil der von MongoDB erstellten ID ist, erzeugt eine Sortierung nach dem primären Schlüssel eine zeitliche Sortierung nach Erstellungszeitpunkt der Dokumente. Auf diese Weise können beispielsweise die zehn aktuellsten Dokumente leicht abgefragt werden.

Selbst gewählter primärer Schlüssel

MongoDB legt den primären Schlüssel im Feld `_id` ab. Wenn Sie Ihrem Dokument dieses Feld selbst mitgeben, übernimmt MongoDB diesen Eintrag als primären Schlüssel für das Dokument. Damit haben Sie die Möglichkeit, selber den primären Schlüssel Ihrer Dokumente bestimmen zu können. Listing 14.11 zeigt ein Beispiel; der primäre Schlüssel wird auf `34001` gesetzt.

Listing 14.11 Einfügen eines Dokuments mit selbst gewählter ID

```
$produkt = [ '_id' => 34001, // Produktnummer als ID
              'typ' => 'krawatte', 'material' => 'Seide',
              'herstellung' => 'Handarbeit, Steiermark',
              'beschreibung' => 'Grüne Seidenkrawatte mit Muster' ];
$coll->insert($produkt);
```

Mit der Abfrage in Listing 14.12 ist zu sehen, dass MongoDB den Wert von `_id` unverändert als primären Schlüssel übernommen hat.

Listing 14.12 Erstelltes Objekt mit selbst gewählter ID

```
> db.produkte.find()
{ "_id" : 34001, "typ" : "krawatte", "material" : "Seide",
  "herstellung" : "Handarbeit, Steiermark",
  "beschreibung" : "Grüne Seidenkrawatte mit Muster" }
```

MongoDB akzeptiert jedes Objekt als Wert für die Dokument-ID, wie in Listing 14.13 zu sehen. Aus Performanzgründen sollten Sie aber davon Abstand nehmen und wo immer möglich einfache Datentypen als ID verwenden. Insbesondere bietet sich die Verwendung einer selbst gewählten ID an, wenn die zu speichernden Dokumente ein Feld enthalten, das eindeutig innerhalb der Kollektion ist.

Listing 14.13 IDs können beliebige Werte sein (nicht empfehlenswert)

```
> db.test.insert({"_id" : {"drop":12, "it":34},
                  "text": "Ketchup zur Bodenpflege"});
> db.test.find();
{ "_id" : { "drop" : 12, "it" : 34 },
  "text" : "Ketchup zur Bodenpflege" }
```

14.3.4 Finden von Dokumenten

Um Dokumente aus der MongoDB-Datenbank abzurufen, wird der Befehl `db.<kollektion>.find()` verwendet. Die PHP-Erweiterung bietet den Befehl als Methoden `MongoCollection::find()` und `MongoCollection::findOne()` an. Dem `find()`-Befehl können zwei Parameter übergeben werden: Die Definition der genauen Abfrage und eine Liste der zurückzuliefernden Dokumentenfelder.

Listing 14.14 zeigt ein typisches Beispiel: In der Kollektion `produkte` sollen alle Einträge vom Typ `hut` gefunden werden. Dazu wird im ersten Parameter einfach der gewünschte Wert des Feldes übergeben. Im zweiten Parameter werden die gewünschten Felder des Resultats aufgeführt, in der Form `$feld => true`. Anstatt alle gewünschten Felder einzeln aufzuführen, können Sie alternativ über `$feld => false` Felder aus dem Ergebnis ausschließen.

Listing 14.14 Suchabfrage von Produkten vom Typ „hut“

```
$db = $mng->selectDB('styleShop');
$coll = $db->selectCollection('produkte');

$res = $coll->find(['typ' => 'hut'],
                  ['preis' => true, 'beschreibung' => true]);
foreach ($res as $hut) {
    echo "Hut (€ $hut[preis]0): $hut[beschreibung] (Id: $hut[_id])\n";
}
```

Listing 14.15 zeigt das Ergebnis der Suchabfrage. Die Ergebniseinträge enthalten alle zusätzlich den primären Schlüssel (Feld `_id`), ohne dass dieser bei den gewünschten Dokumentfeldern angegeben werden muss.

Listing 14.15 Ergebnis der Suchabfrage

```
> php find.php
Hut (€ 49.90): Hut für sonnige Tage (Id: 4fcd18acd3c0eb705a000000)
Hut (€ 24.90): Modischer Freizeithut (Id: 4fcd18acd3c0eb705a000001)
Hut (€ 29.90): Russische Schapka, sehr warm (Id: 4fcd18acd3c0eb705a000002)
```



Wenn Sie den primären Schlüssel im Feld `_id` nicht im Ergebnis retourniert haben wollen, müssen Sie das Feld explizit mit `,'_id' => false` im zweiten Parameter von `find()` unterdrücken.

Suche über primären Schlüssel

Wenn Sie ein Dokument über dessen primären Schlüssel suchen wollen, müssen Sie die ID über die Klasse *MongoId* angeben, in der MongoDB-Konsole über die Funktion *ObjectId()*, wie in Listing 14.16 zu sehen.

Listing 14.16 Abfrage eines Dokumentes über dessen ID

```
MongoDB-Konsole
> db.produkte.find({'_id' : ObjectId("4fcd18acd3c0eb705a000000")})

PHP
$coll = $db->selectCollection('produkte');
$doc = $coll->findOne(['_id' => new MongoId('4fcd18acd3c0eb705a000000')]);
```

MongoDB bietet sehr umfangreiche Möglichkeiten der Suche und Abfrage. Die vollständige Syntax der Abfragesprache wird in Abschnitt 14.4 beschrieben.

Resultat-Cursor

Der `find()`-Befehl liefert einen Cursor (Klasse *MongoCursor*) zurück, mit dem das Ergebnis iteriert werden kann. Die vollständige Resultatmenge befindet sich also nie gleichzeitig im Speicher, was besonders bei großen Datenmengen von Vorteil ist. Die *MongoCursor*-Klasse implementiert die *Iterator*-Schnittstelle, weshalb mit *foreach* die einzelnen Ergebniseinträge einfach durchgegangen werden können.

Mit der Funktion `count()` können Sie die Anzahl der Ergebniseinträge abrufen, beispielsweise:

```
$coll->find(['typ' => 'hut']).count();
```

Die `count()`-Funktion beeinflusst die Position des Cursors nicht, kann also jederzeit aufgerufen werden. Vorsicht bei großen Resultatmengen: Um die Anzahl der Einträge zu ermitteln, muss MongoDB auf der Serverseite alle Resultate durchgehen; das kann sehr lange dauern und verschwendet Ressourcen, insbesondere, wenn Sie mit `limit()` die Trefferanzahl anschließend beschränken (siehe Abschnitt 14.4.4).

14.3.5 Aktualisieren von Dokumenten (Update)

Zum Verändern von Dokumenten stellt MongoDB die Funktion `db.<kollektion>.update()` zur Verfügung, in PHP entsprechend `MongoCollection::update()`. Anders als bei relationalen Datenbanken wird bei MongoDB standardmäßig das vollständige Dokument ausgetauscht!

Wenn sie nur einzelne Felder eines Dokumentes verändern wollen, müssen Sie eine spezielle Syntax verwenden.

Listing 14.17 zeigt den Fall, dass ein Dokument vollständig ausgetauscht wird. Als Suchbegriff wird der primäre Schlüssel des Dokumentes (Feld `_id`) verwendet. Analog zu `insert()` und `update()` könnten in einem zweiten Parameter die Optionen `safe`, `fsync` und `timeout` verwendet werden (siehe Abschnitt 14.3.1). Im Beispiel wird `update()` asynchron verwendet, eine Fehlerprüfung findet anschließend mit `lastError()` statt.

Listing 14.17 Ersetzen eines Dokumentes mit `update()`

```
<?php
$mng = new Mongo(); // Standardverbindung zu localhost:27017
$db = $mng->selectDB('styleShop');
$coll = $db->selectCollection('produkte');

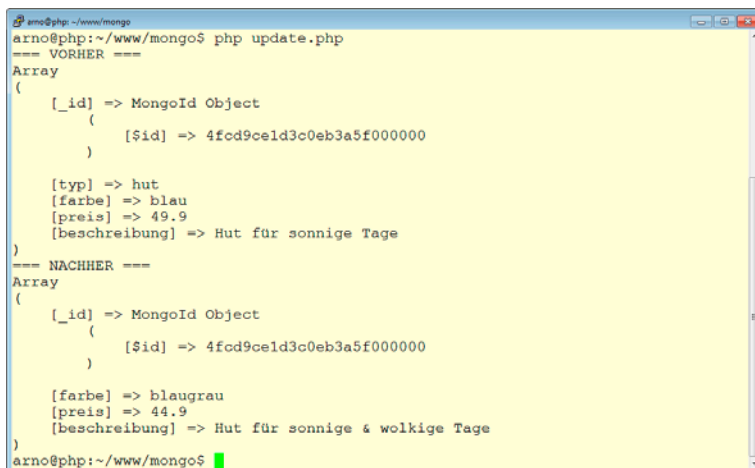
$newerHut = [ 'farbe' => 'blaugrau', 'preis' => 44.9,
              'beschreibung' => 'Hut für sonnige & wolkige Tage' ];

$hut = $coll->findOne(['farbe' => 'blau']);
echo "=== VORHER ===\n";
print_r($hut);

$coll->update(['_id' => $hut['_id']], $newerHut);
$error = $db->lastError();
if (isset($error['code']) || !$error['ok']) {
    // Fehlerbehandlung...
}

$hut = $coll->findOne(['_id' => $hut['_id']]);
echo "=== NACHHER ===\n";
print_r($hut);
?>
```

Bild 14.4 zeigt das Ergebnis der Update-Operation. Das alte Dokument wurde vollständig durch das neue Dokument ersetzt, was unter anderem daran erkennbar ist, dass das Feld `typ` nicht mehr vorhanden ist.



```
arno@php: ~/www/mongo
arno@php:~/www/mongo$ php update.php
=== VORHER ===
Array
(
    [_id] => MongoId Object
        (
            [$id] => 4fcd9ce1d3c0eb3a5f000000
        )

    [typ] => hut
    [farbe] => blau
    [preis] => 49.9
    [beschreibung] => Hut für sonnige Tage
)
=== NACHHER ===
Array
(
    [_id] => MongoId Object
        (
            [$id] => 4fcd9ce1d3c0eb3a5f000000
        )

    [farbe] => blaugrau
    [preis] => 44.9
    [beschreibung] => Hut für sonnige & wolkige Tage
)
arno@php:~/www/mongo$
```

Bild 14.4 Ergebnis der `update()`-Operation

Verändern oder alternativ Einfügen

Wenn das mit `update()` zu verändernde Dokument nicht existiert, wird ein Fehler retourniert. Mit der Option `upsert` im zweiten Parameter von `update()`, können Sie angeben, dass in diesem Fall statt eines Fehlers das übergebene Dokument neu eingefügt werden soll. *Upsert* entspricht damit der aus MySQL bekannten Logik `INSERT INTO table ... ON DUPLICATE KEY UPDATE`.



MongoDB bietet für diesen durchaus häufig vorkommenden Fall (`update()`, sonst `insert()`) die vereinfachte Funktion `MongoCollection::save($dokument)` an. Bei Verwendung dieser Funktion sollten Sie aber darauf achten, dass das Dokument bereits einen primären Schlüssel (Feld `_id`) enthält.

Ändern einzelner Felder

Wenn sie nur einzelne Felder eines Dokumentes verändern wollen, nicht das gesamte Dokument ersetzen, müssen Sie auf die spezielle Syntax der `$`-Befehle zurückgreifen (siehe auch Abschnitt 14.3.6).

Listing 14.18 zeigt, wie die Preise ausgewählter Produkte mit `$inc` erhöht werden können. `$inc` ist keine PHP-Variable, sondern eine spezielle MongoDB-Anweisung, weshalb sie auch in einfachen Anführungszeichen eingeschlossen ist. Da standardmäßig bei `update()` nur ein einziges Dokument verändert wird, selbst wenn die Suchabfrage auf mehrere Dokumente zutrifft, wird mit der Option `'multiple' => true` im zweiten Parameter die Änderung aller Dokumente ermöglicht.

Listing 14.18 Verändern eines einzigen Dokumentfeldes in mehreren Dokumenten

```
<?php
$mng = new Mongo(); // Standardverbindung zu localhost:27017
$db = $mng->selectDB('styleShop');
$coll = $db->selectCollection('produkte');

printPreise($coll, 'VORHER');
try {
    $coll->update(['typ' => 'hut'],
                 ['$inc' => ['$preis' => 5.00]],
                 ['safe' => true, 'upsert' => false, 'multiple' => true]);
}
catch (MongoCursorException $e) {
    // Fehlerbehandlung...
}
printPreise($coll, 'NACHHER');

function printPreise($coll, $txt) {
    echo "=== $txt ===\n";
    $res = $coll->find(['typ' => 'hut']);
    foreach ($res as $hut) {
        echo "$hut[beschreibung]: € $hut[preis]\n";
    }
}
?>
```

Bild 14.5 zeigt das Ergebnis des Beispiels: Alle Preise wurden um € 5,00 erhöht.

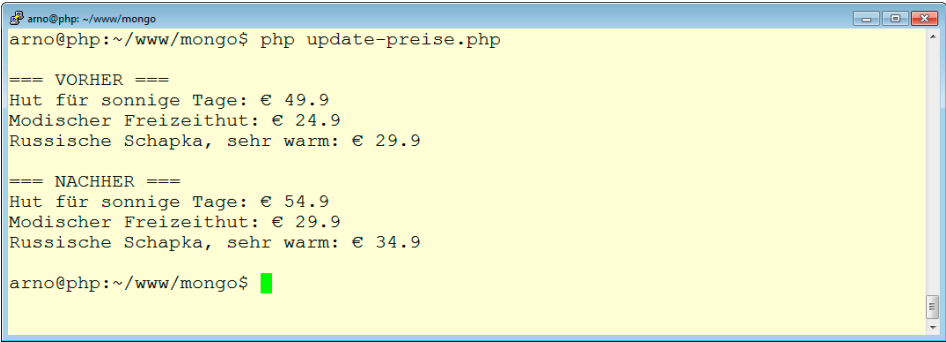


Bild 14.5 Veränderung der Preise aller Produkte

14.3.6 Aktualisieren verschachtelter Dokumente

MongoDB bietet einige Operationen an, um mit *update()* Felder in Dokumenten gezielt zu verändern (siehe Tabelle 14.3). Neben dem Setzen, Löschen und Berechnen von Feldwerten, gibt es Operationen, die Arrays in Dokumentfeldern manipulieren können. Diese Array-Operationen sind besonders nützlich, wenn Sie verschachtelte Dokumente (siehe Abschnitt 14.2.4) verwenden.

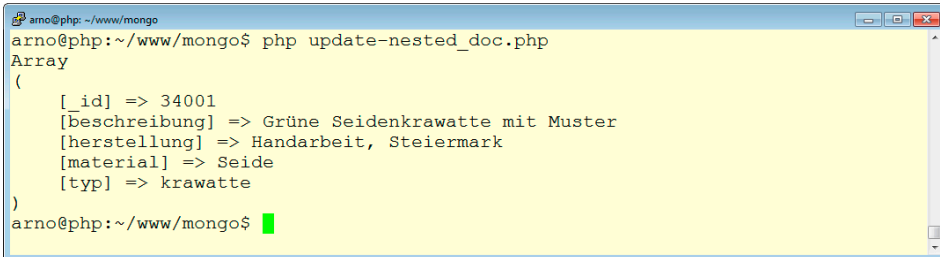
Tabelle 14.3 \$-Befehle für Änderungen einzelner Felder bei update()

Operation	Beschreibung
'\$set' => ['feld' => \$wert]	Setzt das angegebene Feld auf den übergebenen Wert (\$set) bzw. löscht dieses Feld (\$unset)
'\$unset' => ['feld' => 1]	
'\$inc' => ['feld' => \$zahl]	Addiert \$zahl zum angegebenen Feld, die Zahl kann auch negativ sein
'\$bit' => ['feld' => \$bitop]	Führt eine boolesche Verknüpfung durch, \$bitop kann ['\$and' => \$wert] für UND oder ['\$or' => \$wert] für ODER sein
'\$addToSet' => ['feld' => \$wert]	Fügt den angegebenen Wert nur dann dem Array im Feld hinzu, wenn der Wert noch nicht im Array existiert. Um mehrere Werte hinzuzufügen kann \$wert den Inhalt ['\$each' => \$array] haben.
'\$push' => ['feld' => \$wert]	Fügt den angegebenen Wert dem Array im Feld am Ende hinzu. Falls das Feld kein Array ist, wird ein Fehler ausgelöst. Mit \$pushAll können mehrere Werte hinzugefügt werden.
\$pushAll => ['feld' => \$array]	
'\$pop' => ['feld' => 1]	Entfernt das erste (-1) bzw. letzte (1) Element des Arrays im angegebenen Feld
'\$pop' => ['feld' => -1]	

Operation	Beschreibung
'\$pull' => ['feld' => \$wert]	Entfernt den Wert aus dem Array im angegebenen Feld; <i>\$wert</i> kann auch ein Suchkriterium sein, beispielsweise ['\$lte' => 100] oder ['\$nin' => [1, 2, 3]]
'\$pullAll' => ['feld' => \$array]	Entfernt alle Werte in <i>\$array</i> aus dem Array im angegebenen Feld
'\$rename' => ['feld' => \$name]	Benennt das angegebene Feld nach <i>\$name</i> um

Beispiel: Kommentare zu Produkten hinzufügen

Den Produkten in der *styleShop*-Datenbank sollen Kommentare als verschachtelte Dokumente hinzugefügt werden. Als Beispieldokument wird die Krawatte (ID: 34001) aus Abschnitt 14.3.1 verwendet (siehe Bild 14.6).



```

arno@php: ~/www/mongo
arno@php:~/www/mongo$ php update-nested_doc.php
Array
(
    [_id] => 34001
    [beschreibung] => Grüne Seidenkrawatte mit Muster
    [herstellung] => Handarbeit, Steiermark
    [material] => Seide
    [typ] => krawatte
)
arno@php:~/www/mongo$

```

Bild 14.6 Ausgangsdokument #34001 („Krawatte“)

Einzufügende Kommentare bestehen aus einem Benutzernamen, einer Punktebewertung und einem Text. Listing 14.19 zeigt die gewählte Dokumentstruktur für zwei Beispielskommentare.

Listing 14.19 Kommentare zu Produkten

```

$kommentar1 = ['benutzer' => 'Max',
               'bewertung' => 5,
               'text' => 'Sehr geschmackvolles Muster'];
$kommentar2 = ['benutzer' => 'Evi',
               'bewertung' => 4,
               'text' => 'Die Farbe passt ihm!'];

```

Um diese Kommentare dem Produkt hinzuzufügen, wird *update()* mit der Operation *\$push* verwendet, die einem Array den angegebenen Wert hinzufügt oder das Array anlegt, falls der Wert noch nicht existiert. Parallel dazu soll auch die Anzahl der Kommentare mithilfe der *\$inc*-Operation mitgezählt werden.

Listing 14.20 zeigt, wie die beiden Kommentare eingefügt werden. Werden, wie im Beispiel, mehrere Kommentare gleichzeitig eingefügt, lässt sich auch die *\$pushAll*-Operation verwenden, der ein Array der einzufügenden Dokumente übergeben wird.

Listing 14.20 Kommentare mit update() zum Produkt hinzufügen

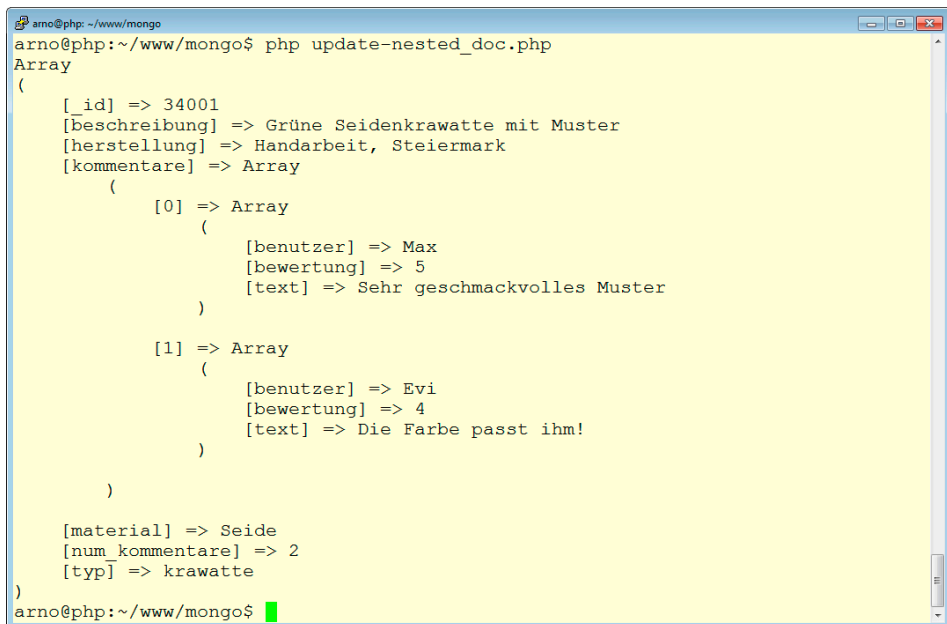
```
// Als einzelne Updates
$coll->update(['_id' => 34001],
             ['$inc' => ['num_kommentare' => 1],
              '$push' => ['kommentare' => $kommentar1]]);
$coll->update(['_id' => 34001],
             ['$inc' => ['num_kommentare' => 1],
              '$push' => ['kommentare' => $kommentar2]]);

// Alternative mit $pushAll
$coll->update(['_id' => 34001],
             ['$inc' => ['num_kommentare' => 2],
              '$pushAll' => ['kommentare' => [$kommentar1, $kommentar2]]]);
```

Bild 14.7 zeigt das Ergebnis der *update()*-Operation. Im Produkt wurden die Felder *kommentare* und *num_kommentare* neu angelegt. Das Kommentarfeld enthält ein Array mit den beiden Kommentardokumenten.

Beispiel: Kommentare entfernen

Verschachtelte Dokumente lassen sich mit den in Tabelle 14.3 aufgeführten Operationen auch wieder gezielt entfernen. Listing 14.21 zeigt, wie im *update()*-Befehl der Operation *\$pull* eine Suchabfrage übergeben wird, die alle Kommentare löscht, die eine Bewertung kleiner 5 haben. Um auch die Anzahl der Kommentare in *num_kommentare* auf den passenden Wert zurückzusetzen, muss die Zahl der zutreffenden Kommentare separat mit *findOne()* ermittelt werden (der Aufruf gibt nur die Bewertungsfelder der Kommentare zurück).



```
arno@php: ~/www/mongo$ php update-nested_doc.php
Array
(
    [_id] => 34001
    [beschreibung] => Grüne Seidenkrawatte mit Muster
    [herstellung] => Handarbeit, Steiermark
    [kommentare] => Array
        (
            [0] => Array
                (
                    [benutzer] => Max
                    [bewertung] => 5
                    [text] => Sehr geschmackvolles Muster
                )
            [1] => Array
                (
                    [benutzer] => Evi
                    [bewertung] => 4
                    [text] => Die Farbe passt ihm!
                )
        )
    [material] => Seide
    [num_kommentare] => 2
    [typ] => krawatte
)
arno@php: ~/www/mongo$
```

Bild 14.7 Ergebnis nach dem Einfügen der Kommentare mit update()

Listing 14.21 Alle Kommentare mit einer Bewertung niedriger als 5 entfernen

```
// Anzahl der Kommentare mit niedriger Bewertung berechnen
$obj = $coll->findOne(['_id' => 34001],
                    ['kommentare.bewertung' => 1, '_id' => 0]);

$count = 0;
foreach ($obj['kommentare'] as $k) {
    if ($k['bewertung'] < 5)
        $count++;
}

// Kommentare entfernen, Anzahl entsprechend reduzieren
$coll->update(['_id' => 34001],
            ['$pull' => ['kommentare' => ['bewertung' => ['$lt' => 5]]],
            ['$inc' => ['num_kommentare' => -$count]],
            ['safe' => true]);
```



HINWEIS: Informationen, wie die Anzahl der Kommentare, können beispielsweise für die Indizierung für Suchanfragen praktisch sein, bei Veränderungen ist das Feld aber manuell neu zu berechnen. Das kann – wie im Beispiel – zu einer zusätzlich notwendigen Suchanfrage führen.

Alternativ hätten die Kommentare auch als eigenständige Dokumente gespeichert werden können. Allerdings wären dann für die Anzeige des Produktes gemeinsam mit Kommentaren zumindest zwei Lesebefehle notwendig. Da bei Webanwendungen üblicherweise die Lesezugriffe im Vergleich zu den Schreibzugriffen stark überwiegen, macht es Sinn, die zusätzliche Arbeit beim Schreiben in Kauf zu nehmen.

Bild 14.8 zeigt das Ergebnis der Operation: Der Kommentar von Evi wurde entfernt und der Kommentarzähler passend reduziert.

```
arno@php: ~/www/mongo
arno@php:~/www/mongo$ php update-nested_doc.php
Array
(
    [_id] => 34001
    [beschreibung] => Grüne Seidenkrawatte mit Muster
    [herstellung] => Handarbeit, Steiermark
    [kommentare] => Array
        (
            [0] => Array
                (
                    [benutzer] => Max
                    [bewertung] => 5
                    [text] => Sehr geschmackvolles Muster
                )
        )
    [material] => Seide
    [num_kommentare] => 1
    [typ] => krawatte
)
arno@php:~/www/mongo$
```

Bild 14.8 Ergebnis nach dem Löschen eines Kommentars mit update()

14.3.7 Löschen von Dokumenten und Kollektionen

Zum Löschen eines Dokumentes wird der Befehl `db.<kollektion>.remove()` – in PHP `MongoCollection::remove()` – verwendet. Dem Befehl wird, gleich wie `find()`, als erster Parameter ein Suchbegriff übergeben, mit dem die zu löschenden Dokumente ausgewählt werden. Wird dem Befehl kein Suchkriterium übergeben, werden alle Dokumente der Kollektion gelöscht! Der PHP-Methode können zusätzlich Optionen im zweiten Parameter übergeben werden. Zum einen die Optionen `safe`, `fsync` und `timeout`, die dieselbe Bedeutung wie beim `insert()`-Befehl haben. Zum anderen die Option `justOne`, die MongoDB anweist, maximal ein Dokument zu löschen, egal wie viele Dokumente auf den Suchbegriff zutreffen. Diese Option stellt ein Sicherheitsnetz gegen Programmierfehler dar.

Listing 14.22 zeigt einen beispielhaften Aufruf, der ein Dokument vom Typ `hut` mit einem Preis von `24.9` löscht (nur ein Dokument, wegen der Option `justOne`). Da es sich um eine sichere Ausführung handelt (Option `safe`), würde im Falle eines Fehlers eine `MongoCursorException` geworfen werden.

Listing 14.22 Löschen von Dokumenten mit `remove()`

```
<?php
$mng = new Mongo(); // Standardverbindung zu localhost:27017
$db = $mng->selectDB('styleShop');
$coll = $db->selectCollection('produkte');

try {
    $coll->remove(['typ' => 'hut', 'preis' => 24.9],
                 ['justOne' => true, 'safe' => true]);
}
catch (MongoCursorException $e) {
    // Fehlerbehandlung
}
?>
```



HINWEIS: Beachten Sie, dass bei einer asynchronen Ausführung die Reihenfolge der Befehle nicht notwendigerweise der Reihenfolge in PHP entspricht bzw. ein vorhergehender Befehl noch nicht beendet sein muss, bevor der nachfolgende ausgeführt wird.

So kann es beispielsweise bei einem asynchron ausgeführten `remove()` dazu kommen, dass unmittelbar *nachfolgend* eingefügte Dokumente, auf die der Suchbegriff auch zutrifft, ebenfalls gelöscht werden.

Löschen einer Kollektion

Zum Löschen einer Kollektion stellt MongoDB den Befehl `db.<kollektion>.drop()` zur Verfügung, in PHP `MongoCollection::drop()`, der ohne Parameter aufgerufen wird. Wenn Sie eine Kollektion löschen, werden alle darin enthaltenen Dokumente sowie etwaig vorhandene Indexe gelöscht. In PHP wird `drop()` immer synchron ausgeführt; der Rückgabewert ist ein Array, dessen Feld `$result['ok']` Auskunft über die erfolgreiche Ausführung des Befehls gibt.

Löschen einer Datenbank

Um eine Datenbank zu löschen, können Sie den Befehl `db.dropDatabase()` verwenden, in PHP `MongoDB::drop()`. Auch dieser Befehl wird ohne Parameter aufgerufen und löscht alle enthaltenen Dokumente und Indexe. Der Rückgabewert in PHP – ein Array mit den Feldern `ok` und `dropped` – liefert keine Anhaltspunkte, ob die Datenbank existiert hat. Lediglich bei Fehlern in der Ausführung ist der Rückgabewert von `$result['ok'] == 0`.

■ 14.4 MongoDB-Abfragesprache

MongoDB bietet eine reichhaltige Abfragesprache, um Dokumente in Kollektionen auszuwählen. Die Abfragesprache umfasst dabei nicht nur vordefinierte Operatoren und Methoden, um auf verschachtelte Dokumente zuzugreifen, sondern sie bietet auch die Möglichkeit, selbst JavaScript-Methoden serverseitig ausführen zu lassen. MongoDB berücksichtigt bei allen Abfragen auch den Datentyp des jeweiligen Dokumentfeldes und ermöglicht damit fein granulare Suchabfragen. Funktionen zum Sortieren und Gruppieren der Ergebniseinträge sind ebenso vorhanden und werden in den Abschnitten 14.4.4 und 14.4.5 vorgestellt.

14.4.1 Operatoren


MongoDB unterstützt eine breite Palette an Operatoren, um Felder auf bestimmte Inhalte zu prüfen. Tabelle 14.4 führt die Operatoren auf. Die angegebenen, zu vergleichenden Werte müssen immer Konstanten sein. Ein Vergleich mit Werten in anderen Feldern ist mit diesen Operatoren nicht möglich.

Tabelle 14.4 Operatoren für Suchabfragen

Operator	Beschreibung
<pre>{ '\$gt' => wert } { '\$gte' => wert } { '\$lt' => wert } { '\$lte' => wert }</pre>	Vergleichsoperatoren: Feld muss größer (<i>\$gt</i>), größer-gleich (<i>\$gte</i>), kleiner (<i>\$lt</i>) oder kleiner-gleich (<i>\$lte</i>) als der angegebene Wert sein
<pre>{ '\$and' => exprarr } { '\$nor' => exprarr } { '\$or' => exprarr } { '\$not' => expression }</pre>	Boolesche Operatoren UND (<i>\$and</i>), ODER (<i>\$or</i>) und NICHT-ODER (<i>\$nor</i>); <i>exprarr</i> ist ein Array von Suchausdrücken. Dem NICHT-Operator (<i>\$not</i>) kann nur ein einziger Suchausdruck übergeben werden.
<pre>{ '\$all' => array } { '\$in' => array } { '\$nin' => array }</pre>	Erfüllt, wenn das Feld alle (<i>\$all</i>), zumindest einen (<i>\$in</i>) oder keinen (<i>\$nin</i>), der im Array aufgeführten Werte enthält
<pre>{ '\$exists' => true/false }</pre>	Überprüft, ob das Feld im Dokument existiert (<i>true</i>) oder nicht existiert (<i>false</i>)

Tabelle 14.4 Operatoren für Suchabfragen (Fortsetzung)

Operator	Beschreibung
<code>{ '\$mod' => {div, rest}}</code>	Führt eine Modulo-Berechnung mit dem Feldwert durch; übereinstimmende Dokumente haben einen Feldwert $x \text{ mod } \text{div} = \text{rest}$
<code>{ '\$ne' => wert}</code>	Erfüllt, wenn der Feldwert ungleich dem angegebenen Wert ist
<code>{ '\$size' => wert}</code>	Berechnet die Größe (Anzahl der Einträge) des Arrays im Feld und vergleicht diese mit dem angegebenen Wert; Einträge in Objekten (assoziativen Arrays) können nicht gezählt werden.
<code>{ '\$type' => wert}</code>	Überprüft, ob der Datentyp des Feldes mit dem angegebenen Datentyp übereinstimmt
<code>new MongoRegex('/regex/opt')</code> <code>{ '\$regex' => string,</code> <code>'\$options' => string}</code>	Reguläre Ausdrücke können auf zwei Arten angegeben werden: Gekapselt in der Klasse <i>MongoRegex</i> oder als <i>\$regex</i> -Operator. MongoDB benutzt die PCRE-Bibliothek, die Syntax der regulären Ausdrücke ist dieselbe wie in den <i>preg_*</i> -PHP-Funktionen.



PRAXISTIPP: Sie können das `$`-Zeichen als Kennzeichen für Operatoren mit der Konfigurationsoption *mongo.cmd* auf ein beliebiges anderes Zeichen setzen, beispielsweise den Doppelpunkt. Das hilft, Operatoren von PHP-Variablen zu unterscheiden und zeitraubende Fehler, wie `"$lte"` (doppelte Anführungsstriche statt einfache), zu verhindern.

Beispiel: `$and`, `$lte`

Um in der *produkte*-Kollektion alle Hüte bis maximal € 40,00 zu suchen:

```
PHP: $coll->find(['$and' => [ ['typ' => 'hut'],
                             ['preis' => ['$lte' => 40]] ]]);
Shell: db.produkte.find({ $and : [ {typ : 'hut'},
                                   {preis : {$lte : 40}} ]})
```

Da *\$and* die Standardeinstellung ist, kann der Ausdruck vereinfacht werden zu:

```
PHP: $coll->find(['typ' => 'hut', 'preis' => ['$lte' => 40]]);
Shell: db.produkte.find({typ : 'hut', preis : {$lte : 40}})
```

Beispiel: `$or`, `$in`, `$exists`

Um Produkte zu suchen, die entweder kein Farbfeld enthalten oder deren Farbe Blau, Türkis oder Rot ist:

```
PHP: $coll->find(['$or' => [
    ['farbe' => ['$exists' => false]],
    ['farbe' => ['$in' => ['blau', 'türkis', 'rot']]] ]]);
Shell: db.produkte.find({$or : [{farbe : {$exists : false}},
                                {farbe : {$in : ['blau', 'türkis', 'rot']}} ]})
```




Die booleschen Operatoren können nicht innerhalb von Feldwerten verwendet werden. Der folgende Suchausdruck ist ein ungültiger Ausdruck und liefert einen Fehler retour:

```
db.produkte.find({farbe : { $or : [ { $exists : false },
                                   { $in : ['blau', 'türkis', 'rot']} ] } })
```

Beispiel: Reguläre Ausdrücke

Um alle Produkte zu suchen, deren Beschreibung mit der Zeichenfolge *modisch* beginnt:

```
PHP: $coll->find(['beschreibung' => new MongoRegex('/^modisch/i')]);
Shell: db.produkte.find({'beschreibung' : /^modisch/i })
```

Alternativ können Sie die Langform verwenden.

```
PHP: $coll->find(['beschreibung' => ['$regex' => '^modisch',
                                   '$options' => 'i']]);
Shell: db.produkte.find({'beschreibung' : { $regex : "^modisch",
                                             $options : "i" } })
```



PRAXISTIPP: In PHP macht die Langform vor allem dann Sinn, wenn Sie viele Schrägstriche im regulären Ausdruck verwenden, beispielsweise für Suchen nach definierten URLs. *MongoRegex* lässt derzeit als Begrenzungszeichen nur den Schrägstrich zu, Schrägstriche innerhalb des Ausdrucks müssen also mit `||/` maskiert werden.

14.4.2 Verschachtelte Dokumente und Arrays

MongoDB ist sehr flexibel, was die Konvertierung zwischen Datentypen betrifft. Das gilt insbesondere für Arrays in Feldern: MongoDB extrahiert daraus die Werte, um auch Vergleiche mit einfachen Datentypen zu ermöglichen. Dieses Vorgehen wird in MongoDB *MultiKeys* genannt.

Listing 14.23 zeigt ein Beispiel mit zwei Produkten, die ein Array von Strings im *kategorie*-Feld enthalten. Die Suche mit `{kategorie: 'Freizeit'}` – ein Vergleich mit einem String – liefert beide Produkte als Ergebnis, da die Arrays diesen String als Wert enthalten. Analog werden auch die anderen Operatoren aus Tabelle 14.4 direkt auf die Werte im Array angewandt, nicht auf das Array selbst, wie in Listing 14.23 anhand der Abfragen mit *\$in* und dem regulären Ausdruck zu sehen.

Listing 14.23 Vergleich von Werten in Arrays

```
// Zwei Produkte
{ _id: 1, typ: 'hut', kategorie: ['Freizeit', 'Sommer'], preis: 24.9,
  beschreibung: 'Modischer Freizeithut' }
{ _id: 2, typ: 'hut', kategorie: ['Freizeit', 'Winter'], preis: 29.9,
  beschreibung: 'Russische Schapka, sehr warm' }
```

```
> db.produkte.find({kategorie: 'Freizeit'}, ['_id'])
{ "_id" : 1 }
{ "_id" : 2 }
> db.produkte.find({kategorie: {$in: ['Herbst', 'Winter']}}, ['_id'])
{ "_id" : 2 }
> db.produkte.find({kategorie: /^Somm/}, ['_id'])
{ "_id" : 1 }
```

Zugriff auf verschachtelte Dokumente

MongoDB bietet eine einfache – von JavaScript entlehnte – Syntax, um auf Felder, innerhalb eingebetteter Dokumente, zuzugreifen: Den Punkt. Listing 14.24 zeigt ein Beispiel mit in Produkten eingebetteten Kommentaren. Um Produkte auszuwählen, die Kommentare mit einer 5-Punkte-Bewertung enthalten, kann auf das Bewertungsfeld mit *kommentare.bewertung* zugegriffen werden. Die zweite Beispielabfrage im Listing gibt alle Produkte aus, die keinen Kommentar von *Max* oder *Moritz* enthalten. Da Produkte ohne Kommentare dann ebenfalls automatisch ausgewählt würden, wird mit *\$exists* noch geprüft, ob das Produkt Kommentare enthält.

Listing 14.24 Zugriff auf verschachtelte Dokumente mit Punktnotation

```
{ _id: 17, beschreibung: 'Goldene Seidenkrawatte mit Muster',
  kommentare: [
    { benutzer: 'Max', bewertung: 5, text: 'Sehr geschmackvolles Muster' },
    { benutzer: 'Evi', bewertung: 4, text: 'Die Farbe passt ihm!' },
    { benutzer: 'Christina', bewertung: 4, text: 'Passt gut zu schwarz.' }
  ]
}
{ _id: 46, beschreibung: 'Blaue Seidenkrawatte, gemustert',
  kommentare: [
    { benutzer: 'Arno', bewertung: 5, text: 'Für alle Anlässe passend' },
    { benutzer: 'Christina', bewertung: 5, text: 'Himmlisch!' }
  ]
}

> db.produkte.find({ 'kommentare.bewertung': 5 }, ['_id'])
{ "_id" : 17 }
{ "_id" : 46 }
> db.produkte.find({ $and: [
  { kommentare: { $exists: true } },
  { 'kommentare.benutzer': {$nin: ['Max', 'Moritz']} } ], ['_id'])
{ "_id" : 46 }
```



PRAXISTIPP: Die Punktnotation kann auch eingesetzt werden, um auf das n-te Element in einem Array zuzugreifen. Die Abfrage, ob beispielsweise der erste Kommentar von Max geschrieben wurde, lautet: `{ ,kommentare.0.benutzer': 'Max' }`

Kombinierte Abfragen auf verschachtelte Dokumente: \$elemMatch

Wie soll eine Abfrage aussehen, wenn wir alle Produkte auswählen wollen, die Kommentare von *Christina* mit einer Bewertung von 5 Punkten enthalten? Listing 14.25 zeigt einen Versuch, bei dem beide Bedingungen mit *\$and* verknüpft werden. Das Ergebnis sind aber beide

Produkte, nicht nur Produkt 46. Warum? Der Grund dafür ist, dass jede Teilabfrage (auf Benutzer und auf Bewertung) eigenständig auf das *gesamte* Produktdokument angewandt wird. Auch Produkt 17 hat einen Kommentar von *Christina* und eine Bewertung von 5 Punkten, weshalb es ebenfalls in das Ergebnis aufgenommen wird.

Listing 14.25 Ungewollte Kombinationsabfrage

```
> db.produkte.find({ $and: [ { 'kommentare.benutzer': 'Christina' },
                             { 'kommentare.bewertung': 5 } ] }, ['_id'])
{ "_id" : 17 }
{ "_id" : 46 }
```

Für die Verknüpfung der beiden Abfragen, sodass sie auf einen einzigen Kommentar zutreffen müssen, bietet MongoDB den Operator *\$elemMatch* an. Listing 14.26 zeigt die Abfrage für Produkte, die einen mit 5 Punkten bewerteten Kommentar von *Christina* enthalten.

Listing 14.26 Kombinationsabfrage von Subdokumenten mit \$elemMatch

```
> db.produkte.find({ kommentare: { '$elemMatch': {
                                     benutzer: 'Christina', bewertung: 5 } } }, ['_id'])
{ "_id" : 46 }
```

14.4.3 Komplexe Bedingungen (\$where)

Wenn für Ihre Anwendung die von MongoDB zur Verfügung gestellten Operatoren nicht ausreichen, weil Sie beispielsweise eine Berechnung, basierend auf den Feldwerten, durchführen wollen oder Felder untereinander vergleichen wollen, können Sie die *\$where*-Klausel verwenden. Dort geben Sie eine JavaScript-Funktion an, die serverseitig von MongoDB ausgeführt wird. Liefert die Funktion *true* zurück, wird das aktuelle Dokument in die Resultatmenge aufgenommen, sonst verworfen.

Listing 14.27 fügt die Beispielprodukte in die Datenbank ein, die im Folgenden für die Abfragen mit *\$where* verwendet werden sollen. Die Kommentare bestehen aus Benutzernamen, Punktebewertung und Datum. Um von PHP aus, den MongoDB-Datums-Datentyp zu erzeugen, wird die Klasse *MongoDate* verwendet.

Listing 14.27 Beispielprodukte für folgende Abfragen

```
$produkte = [
    [ '_id' => 5, 'beschreibung' => 'Hemd, einfarbig, schwarz',
      'kommentare' => [
        ['benutzer' => 'Max', 'bewertung' => 5,
          'datum' => new MongoDate(strtotime('2012-06-26 12:08:26'))],
        ['benutzer' => 'Evi', 'bewertung' => 3,
          'datum' => new MongoDate(strtotime('2012-08-27 14:18:27'))],
        ['benutzer' => 'Christina', 'bewertung' => 3,
          'datum' => new MongoDate(strtotime('2012-08-27 17:00:46'))]]],
    [ '_id' => 6, 'beschreibung' => 'Hemd, einfarbig, weiß',
      'kommentare' => [
        ['benutzer' => 'Arno', 'bewertung' => 5,
          'datum' => new MongoDate(strtotime('2012-07-26 12:08:26'))],
        ['benutzer' => 'Christina', 'bewertung' => 5,
```

```
'datum' => new MongoDBDate(strtotime('2012-08-01 17:46:08'))],
['benutzer' => 'Pia', 'bewertung' => 3,
'datum' => new MongoDBDate(strtotime('2012-08-03 11:12:13'))]]];
$collection->batchInsert($produkte);
```

Listing 14.28 zeigt, wie Sie mithilfe der *\$where*-Klausel die durchschnittliche Bewertung eines Produktes berechnen können und nur jene Produkte auswählen, die eine höhere Bewertung als 4 Punkte haben. Die JavaScript-Funktion prüft zuerst, ob Kommentare vorhanden sind, berechnet dann die Durchschnittsbewertung und liefert für alle Produkte mit einer Bewertung größer als *cutoff true* zurück.

Die Variable *cutoff* ist in der JavaScript-Funktion selbst nicht definiert. Beim Erstellen des Funktionsparameters mit der Klasse *MongoCode* können Sie im zweiten Parameter Variablen binden – eine saubere Möglichkeit, um von PHP berechnete Werte an den JavaScript-Code zu übergeben. Schließlich wird mit *MongoCollection::find()* die Suchabfrage durchgeführt. Als Ergebnis wird nur das Produkt mit ID 6 (siehe Listing 14.27) retourniert.

Listing 14.28 Abfrage mit Berechnung der durchschnittlichen Bewertung

```
<?php
$mng = new Mongo();
$db = $mng->selectDB('styleShop');
$coll = $db->selectCollection('produkte');

$jsfunc = <<<JSFUNC
function() {
    if (!this.kommentare) { // Falls Produkt keine Kommentare hat
        return false;
    }
    var sum=0;
    for (var i=0; i < this.kommentare.length; i++) {
        sum += this.kommentare[i].bewertung;
    }
    return (sum/this.kommentare.length >= cutoff);
}
JSFUNC;

$clause = new MongoCode($jsfunc, ['cutoff' => 4]);
$res = $coll->find(['$where' => $clause], ['beschreibung']);
foreach ($res as $p) {
    echo "Id $p[_id]: $p[beschreibung]\n";
}
?>
```

Listing 14.29 zeigt ein weiteres Beispiel, bei dem die durchschnittliche Bewertung aller Kommentare vor einem bestimmten Zeitpunkt berechnet wird. Im Aufruf von *MongoCollection::find()* werden zusätzlich zur *\$where*-Klausel auch weitere Operatoren verwendet, um das Suchergebnis einzuschränken. Obwohl diese Bedingungen auch direkt in die JavaScript-Funktion aufgenommen werden könnten, ist es effizienter, dies über die vordefinierten MongoDB-Operatoren zu tun.

So kann beispielsweise der reguläre Ausdruck in der Abfrage von einem Index auf dem Feld *beschreibung* Gebrauch machen und damit die Menge der Kandidaten bei einer großen Kollektion vorab einschränken. Analog wird mit *kommentare.datum* überprüft, ob überhaupt Kommentare vor dem angegebenen Datum vorhanden sind.

Listing 14.29 Durchschnitt aller Bewertungen vor einem bestimmten Datum

```

$jsfunc = <<<JSFUNC
function() {
    var sum=0, cnt=0;
    for (var i=0; i < this.kommentare.length; i++) {
        if (this.kommentare[i].datum > beforeDate) {
            continue;
        }
        sum += this.kommentare[i].bewertung;
        ++cnt;
    }
    return (sum/cnt >= cutoff);
}
JSFUNC;

$mdate = new MongoDate(strtotime('2012-08-01'));
$clause = new MongoCode($jsfunc, ['cutoff' => 4, 'beforeDate' => $mdate]);
$res = $coll->find(['kommentare.datum' => ['$lt' => $mdate],
    'beschreibung' => new MongoRegex('/^Hemd/'),
    '$where' => $clause], ['beschreibung']);

```

Die serverseitige JavaScript-Berechnung ist ein mächtiges Instrument, es sollte aber sehr behutsam eingesetzt werden, um keine Performanz zu verlieren. Wo immer möglich, sollten Sie Abfragen mit den MongoDB-Operatoren formulieren, da diese eine höhere Geschwindigkeit bieten.

14.4.4 Sortieren

MongoDB bietet analog zu relationalen Datenbanken die Möglichkeit, Resultate zu sortieren. Die Definition erfolgt über den Cursor des Resultats, in PHP über die Funktion *MongoCursor::sort()*. Sie können nach mehreren Feldern sortieren, auf- und absteigend.

Listing 14.30 zeigt, wie nach dem Feld *beschreibung* aufsteigend sortiert werden kann. Die Zuweisung 'beschreibung' => 1 bedeutet dabei aufsteigendes Sortieren, 'beschreibung' => -1 bedeutet absteigendes Sortieren. Der *sort()*-Funktion können im Array mehrere Felder übergeben werden, die Sortierhierarchie der Felder ist von links nach rechts (das heißt, nachfolgende Felder werden nur berücksichtigt, wenn das vorangestellte Feld einen identischen Wert hat).

Listing 14.30 Sortieren des Ergebnisses

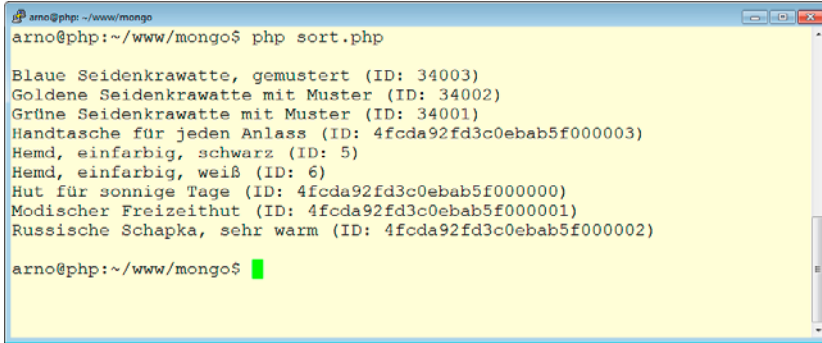
```

<?php
$mng = new Mongo();
$db = $mng->selectDB('styleShop');
$coll = $db->selectCollection('produkte');

$crsr = $coll->find([])->sort(['beschreibung' => 1]);
foreach ($crsr as $prod) {
    echo "$prod[beschreibung] (ID: $prod[_id])\n";
}
?>

```

Bild 14.9 zeigt das Ergebnis der Sortierung. MongoDB verwendet derzeit für den Vergleich bei Strings eine einfache Sortierung nach dem Unicode-Codepoint des Zeichens (infolge der verwendeten UTF-8-Kodierung). Daraus ergibt sich, dass beispielsweise Großbuchstaben vor den Kleinbuchstaben kommen, Umlaute erst nach dem z etc. Eine eigene Sortierreihenfolge kann nicht angegeben werden – leider hat die Internationalisierung derzeit keinen hohen Stellwert bei MongoDB.



```

arno@php: ~/www/mongo
arno@php:~/www/mongo$ php sort.php

Blaue Seidenkrawatte, gemustert (ID: 34003)
Goldene Seidenkrawatte mit Muster (ID: 34002)
Grüne Seidenkrawatte mit Muster (ID: 34001)
Handtasche für jeden Anlass (ID: 4fcda92fd3c0ebab5f000003)
Hemd, einfarbig, schwarz (ID: 5)
Hemd, einfarbig, weiß (ID: 6)
Hut für sonnige Tage (ID: 4fcda92fd3c0ebab5f000000)
Modischer Freizeithut (ID: 4fcda92fd3c0ebab5f000001)
Russische Schapka, sehr warm (ID: 4fcda92fd3c0ebab5f000002)

arno@php:~/www/mongo$

```

Bild 14.9 Sortierte Ergebnismenge



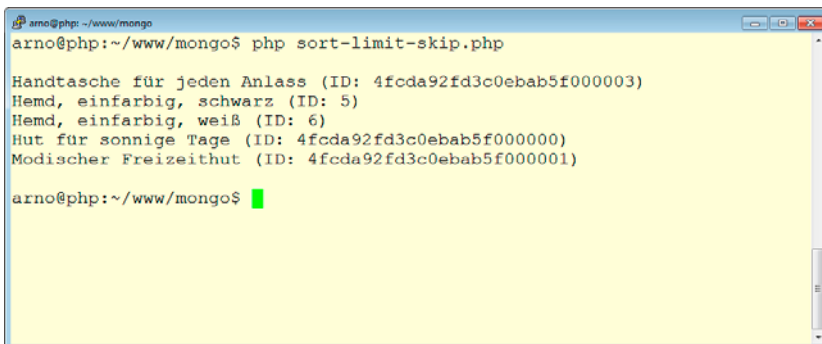
PRAXISTIPP: MongoDB setzt ein hartes Limit von 32 MB Arbeitsspeicher zum Sortieren, größere Resultatmengen lassen sich nicht sortieren. Um nicht an das Speicherlimit zu stoßen, sollten Sie entweder einen Index zum Sortieren verwenden oder die Resultatmenge mit *limit()* beschränken.

limit() und skip()

Über den Cursor des Resultats können Sie MongoDB auch anweisen, nur eine definierte Anzahl an Ergebnissen zurückzuliefern, ausgehend von einem Offset. Damit können Sie beispielsweise Funktionen zum Blättern durch Resultatlisten bequem umsetzen.

Bild 14.10 zeigt das Resultat, wenn die Ergebnisse wie folgt eingeschränkt werden.

```
$crsr = $coll->find([])->sort(['beschreibung'=>1])->skip(3)->limit(5);
```



```

arno@php: ~/www/mongo
arno@php:~/www/mongo$ php sort-limit-skip.php

Handtasche für jeden Anlass (ID: 4fcda92fd3c0ebab5f000003)
Hemd, einfarbig, schwarz (ID: 5)
Hemd, einfarbig, weiß (ID: 6)
Hut für sonnige Tage (ID: 4fcda92fd3c0ebab5f000000)
Modischer Freizeithut (ID: 4fcda92fd3c0ebab5f000001)

arno@php:~/www/mongo$

```

Bild 14.10 Sortierte Ergebnisse, limitiert auf fünf Einträge

14.4.5 Gruppieren

MongoDB bietet eine *group()*-Funktion an, die ähnlich funktioniert wie die *GROUP BY*-Funktion von relationalen Datenbanken. In PHP ist die Funktion als *MongoCollection::group()* verfügbar. Die *group()*-Funktion ist der kleine Bruder von Map/Reduce (siehe Abschnitt 14.6), beim Aufruf müssen Sie eine Reduce-Funktion angeben, welche die aggregierten Ergebnisse berechnet.

group() und seine Parameter werden nachfolgend im Kontext des Webshop-Beispiels erklärt. Als Beispieldaten verwenden wir dazu die Produkte aus Listing 14.31. Die angeführten Produkte enthalten Informationen zum Typ, zum Preis und zum Lagerbestand.

Listing 14.31 Beispieldaten zur Gruppierung

```
$produkte = [
    ['_id' => 100, 'typ' => 'Hemd', 'lagernd' => 20, 'preis' => 29.9],
    ['_id' => 101, 'typ' => 'Hemd', 'lagernd' => 75, 'preis' => 29.9],
    ['_id' => 102, 'typ' => 'Hemd', 'lagernd' => 5, 'preis' => 34.9],
    ['_id' => 110, 'typ' => 'Krawatte', 'lagernd' => 15, 'preis' => 14.9],
    ['_id' => 111, 'typ' => 'Krawatte', 'lagernd' => 23, 'preis' => 19.9],
    ['_id' => 120, 'typ' => 'Hut', 'lagernd' => 9, 'preis' => 33.9],
    ['_id' => 121, 'typ' => 'Hut', 'lagernd' => 13, 'preis' => 27.9],
    ['_id' => 122, 'typ' => 'Hut', 'lagernd' => 2, 'preis' => 69.9],
    ['_id' => 123, 'typ' => 'Hut', 'lagernd' => 4, 'preis' => 59.9]];
$coll->batchInsert($produkte);
```

Aggregieren der Resultate (reduce)

Die Funktionsweise von *group()* ist ähnlich der PHP-Funktion *array_reduce()*. Eine Funktion wird der Reihe nach mit allen Dokumenten aufgerufen und aggregiert in einem Summenobjekt (Akkumulatorobjekt) die gewünschten Ergebnisfelder. Listing 14.32 zeigt so eine JavaScript-Funktion. Der Funktion werden zwei Parameter übergeben, das aktuelle Element (*prod*, Produkt im Webshop) und das Akkumulatorobjekt *acc*. Die Funktion zählt einerseits in *acc.cnt* mit, wie viele Produkte übergeben worden sind, addiert den Lagerbestand auf und hält fest, bei wie vielen Produkten der Lagerbestand unter *lowInventory* liegt.

Listing 14.32 Funktion zum Aggregieren der Eigenschaften

```
function(prod, acc) {
    acc.cnt++;
    acc.inv += prod.lagernd;
    if (prod.lagernd < lowInventory) {
        acc.low++;
    }
}
```

Ablauf von group()

group() teilt zu Beginn der Berechnung die Dokumente der Suchabfrage in Teilgruppen nach einem Gruppierungsschlüssel ein. Der Gruppierungsschlüssel kann ein Feld, eine Liste von Feldern oder eine Funktion sein; Dokumente mit gleichem Schlüssel werden in dieselbe Teilgruppe eingefügt. Im Anschluss ruft *group()* für jede Teilgruppe die *Reduce*-

Funktion auf und fügt das so entstandene Summenobjekt ins Ergebnis ein. Am Ende erhalten Sie auf diesem Wege ein Summenobjekt für jede Teilgruppe.

MongoCollection::group() übernimmt für die Funktion genau drei Parameter: Den Gruppierungsschlüssel, einen Initialwert für das Summenobjekt und die *Reduce*-Funktion. In einem vierten optionalen Parameter können Sie eine Suchanfrage angeben sowie eine *Finalize* genannte Funktion, die das Summenobjekt abschließend manipulieren kann, bevor es ins Ergebnis eingefügt wird.

Beispiel: Niedriger Lagerbestand

Listing 14.33 zeigt ein Beispiel zur Anwendung von *group()*. Es werden pro Produkttyp (Gruppierungsschlüssel *typ*) die Anzahl der Produkte, der Lagerbestand und Anzahl der Produkte mit niedrigem Lagerbestand berechnet. Als *Reduce*-Funktion dient die Funktion aus Listing 14.32. Das Summenobjekt wird mit allen Zählern (*cnt*, *inv*, *low*) gleich Null initialisiert, eine zusätzlich angegebene *Finalize*-Funktion berechnet Durchschnittswerte. Als Suchanfrage (im Feld *condition*) wird sichergestellt, dass nur Produkte in die Berechnung aufgenommen werden, die das Feld *lagernd* enthalten.

Listing 14.33 Gruppieren nach Produkttyp und Berechnung des Lagerbestandes

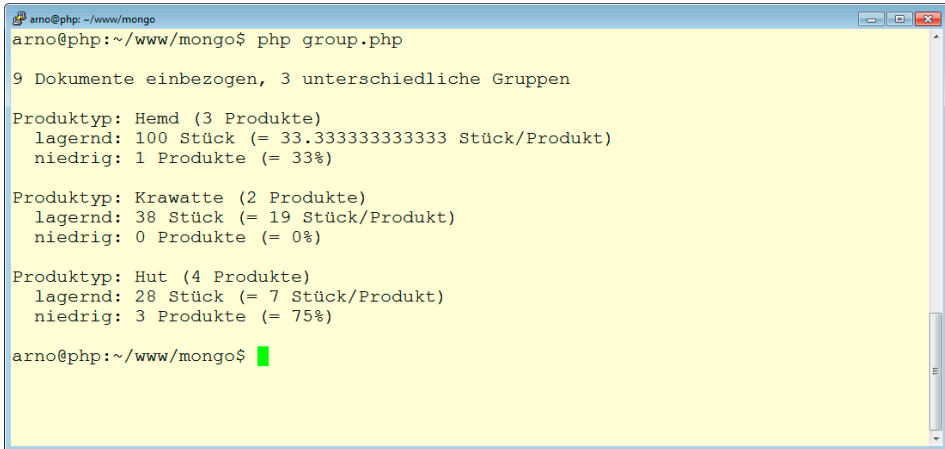
```
<?php
$mng = new Mongo();
$db = $mng->selectDB('styleShop');
$coll = $db->selectCollection('produkte');

$reducejs = 'function (prod, acc) { ... }'; // siehe oben
$finalizejs = <<<FINALIZE
function(acc) {
    acc.avg = acc.inv / acc.cnt;
    acc.low_percent = Math.round(acc.low / acc.cnt * 100);
}
FINALIZE;

$res = $coll->group(['typ' => true],
    ['cnt' => 0, 'inv' => 0, 'low' => 0],
    new MongoCode($reducejs, ['lowInventory' => 10]),
    ['condition' => ['lagernd' => ['$exists' => true]],
    'finalize' => new MongoCode($finalizejs)]);

echo "$res[count] Dokumente einbezogen, ",
    "$res[keys] unterschiedliche Gruppen\n\n";
foreach ($res['retval'] as $item) {
    echo <<<EOF
Produkttyp: $item[typ] ($item[cnt] Produkte)
    lagernd: $item[inv] Stück (= $item[avg] Stück/Produkt)
    niedrig: $item[low] Produkte (= $item[low_percent]%) \n\n
EOF;
}
?>
```

Bild 14.11 zeigt das Ergebnis der Berechnung. *MongoCollection::group()* liefert als Ergebnis ein Array zurück, das unter dem Schlüssel *count* die Anzahl der durchsuchten Dokumente enthält, unter *keys* die Anzahl der durch den Gruppierungsschlüssel entstandenen Gruppen und unter *retval* die Ergebnis-Arrays der einzelnen Gruppen.



```

arno@php: ~/www/mongo$ php group.php

9 Dokumente einbezogen, 3 unterschiedliche Gruppen

Produkttyp: Hemd (3 Produkte)
  lagernd: 100 Stück (= 33.333333333333 Stück/Produkt)
  niedrig: 1 Produkte (= 33%)

Produkttyp: Krawatte (2 Produkte)
  lagernd: 38 Stück (= 19 Stück/Produkt)
  niedrig: 0 Produkte (= 0%)

Produkttyp: Hut (4 Produkte)
  lagernd: 28 Stück (= 7 Stück/Produkt)
  niedrig: 3 Produkte (= 75%)

arno@php: ~/www/mongo$

```

Bild 14.11 Ergebnis der Abfrage mit `group()`

Beispiel: Funktion als Gruppierungsschlüssel

Den Gruppierungsschlüssel durch eine Funktion berechnen zu lassen, ist nützlich, wenn die gewünschten Gruppenkategorien berechnet werden müssen. Wenn Sie zum Beispiel die Produkte in Intervallen von 20,00 € gruppieren wollen, können Sie die Funktion in Listing 14.34 verwenden. Die Funktion muss ein Objekt *retour* liefern, nachdem gruppiert wird. Im Listing würde die Angabe von *preis_von* ausreichen, da *preis_bis* sich direkt daraus errechnet; der besseren Codestruktur wegen, wird *preis_bis* aber gleich hier mitberechnet und nicht erst später im PHP-Code.

Listing 14.34 Funktion zur Berechnung des Gruppierungsschlüssels

```

function (prod) {
    var key = Math.floor(prod.preis/20) * 20;
    return { preis_von : key,
            preis_bis : key+20 };
}

```

Listing 14.35 zeigt die Anwendung der Funktion. Die Funktion aus Listing 14.34 wird `MongoCollection::group()` als erster Parameter übergeben, als dritter Parameter wieder die *Reduce*-Funktion. Da die Gruppierungsschlüssel in das Summenobjekt aufgenommen werden, kann das Preisintervall in PHP bequem verwendet werden.

Listing 14.35 Gruppierung und Ausgabe

```

$reduceJS = <<<REDUCE
function (prod, acc) {
    var total = prod.lagernd * prod.preis;
    acc.sum += total;
    if (total > acc.max) {
        acc.max = total;
        acc.produkt = prod.typ + ", Id: " + prod._id
                    + ", Stückpreis: " + prod.preis;
    }
}

```

```


}
REDUCE;
$reduceCode = new MongoCode($reduceJS);
$keyCode = new MongoCode('function (prod) { ... }'); // siehe oben

$res = $coll->group($keyCode, ['sum' => 0, 'max' => 0], $reduceCode,
    ['condition' => ['lagernd' => ['$exists' => true]]]);

foreach ($res['retval'] as $grp) {
    echo "Produkte von $grp[preis_von] € bis $grp[preis_bis] €\n";
    echo "Lagerbestand: $grp[sum]€\n";
    echo "Größter Bestand ($grp[max]€): $grp[produkt]\n\n";
}
?>

```

Bild 14.12 zeigt das Ergebnis der Gruppierung. Die Produkte wurden in die gewünschten Preisintervalle gruppiert und Kennzahlen in *Reduce* berechnet.



```

arno@php: ~/www/mongo
arno@php:~/www/mongo$ php group-keyfunc.php

Produkte von 0 € bis 20 €
Lagerbestand: 681.2€
Größter Bestand (457.7€): Krawatte, Id: 111, Stückpreis: 19.9

Produkte von 20 € bis 40 €
Lagerbestand: 3682.8€
Größter Bestand (2242.5€): Hemd, Id: 101, Stückpreis: 29.9

Produkte von 40 € bis 60 €
Lagerbestand: 239.6€
Größter Bestand (239.6€): Hut, Id: 123, Stückpreis: 59.9

Produkte von 60 € bis 80 €
Lagerbestand: 139.8€
Größter Bestand (139.8€): Hut, Id: 122, Stückpreis: 69.9

arno@php:~/www/mongo$ █

```

Bild 14.12 Ergebnis der Gruppierung nach Preiskategorien

■ 14.5 Indizieren von Feldern

Ein herausragendes Merkmal von MongoDB ist die Möglichkeit, Felder zu indizieren. Es ist daher nicht – wie bei anderen NoSQL-Datenbanken – notwendig, selbst eine Indexstruktur anzulegen. Standardmäßig legt MongoDB bereits einen Index für den primären Schlüssel an, die Suche nach Objekten, basierend auf ihrem primären Schlüssel, ist also sehr schnell.

Das Anlegen eines Index geschieht mit dem Befehl *ensureIndex()*, dem als Parameter ein Objekt übergeben wird, dessen Einträge die zu indizierenden Felder des Dokumentes darstellen. Listing 14.36 zeigt einige Beispiele. Ein Index hat pro Feld eine Sortierrichtung, aufsteigend +1, absteigend -1. Die Sortierrichtung ist nur bei Indexen mit zwei oder mehr Feldern relevant und bestimmt darüber, ob der Index auch zur Sortierung mit *sort()* verwendet werden kann.

Listing 14.36 Beispiele zum Anlegen von Indexen

```
// Index auf einem Feld, aufsteigend
PHP: $coll->ensureIndex(['beschreibung' => 1]);
Shell: db.produkte.ensureIndex({ Beschreibung: 1 })

// Kombierter Index auf zwei Feldern, beide aufsteigend
PHP: $coll->ensureIndex(['typ' => 1, 'beschreibung' => 1]);
Shell: db.produkte.ensureIndex({ typ: 1, beschreibung: 1 })

// Kombierter Index, auf- und absteigend sortiert
PHP: $coll->ensureIndex(['preis' => -1, 'typ' => 1, 'beschreibung' => 1]);
Shell: db.produkte.ensureIndex({ preis: -1, typ: 1, beschreibung: 1 })
```

14.5.1 Indexarten

MongoDB unterstützt vier unterschiedliche Indexarten, die über einen Parameter der Methode *ensureIndex()* ausgewählt werden können:

- **Gewöhnlicher Index:** Der gewöhnliche Index unterliegt keiner Einschränkung, betreffend den Werten der Felder, insbesondere dürfen identische Werte mehrfach vorkommen.
- **Eindeutiger Index (*unique index*):** Wenn in einem Feld bzw. bei der Kombination von Feldern nur eindeutige Werte vorkommen, dann bietet es sich an, einen eindeutigen Index zu verwenden. Unter anderem kann dieser Index dazu verwendet werden, beim Einfügen sicherzustellen, dass ein Wert nicht bereits existiert.
- **Multikey-Index:** MongoDB indiziert die einzelnen Werte von Array-Feldern (zum Beispiel: Kategorie, Schlagwort etc.). Dadurch wird dasselbe Dokument mehrfach unter verschiedenen Indexschlüsseln im Index abgelegt. Die Möglichkeit, Werte von Arrays zu indizieren, ist eine Stärke von MongoDB.
- **2D-Geoindex:** Zum Indizieren von Geopositionen bietet MongoDB einen Geoindex an. Mit ihm können einfache Positionsabfragen performant umgesetzt werden.

Bei eindeutigen Indexen können fehlende Felder ein Problem darstellen. Da fehlende Felder in Dokumenten als *NULL* indiziert werden, können keine zwei Dokumente, welche die Indexfelder nicht haben, eingefügt werden, da sonst für *NULL* zwei Einträge existieren würden. MongoDB bietet für diesen Fall die Option *sparse* an. Ein *sparse*-Index erlaubt fehlende Felder in den Dokumenten.

Wenn Sie einen eindeutigen Index für bereits bestehende Daten anlegen wollen und dort einige Dubletten vorhanden sind, können Sie MongoDB mit der Option *dropDups* anweisen, nachfolgende Dokumente mit identischem Indexschlüssel *zu löschen*. Sie sollten vorab sicherstellen, dass die zu löschenden Dokumente keine wertvollen Daten mehr enthalten.

Listing 14.37 zeigt, wie Sie von PHP aus die unterschiedlichen Indexarten anlegen können. Dem zweiten Parameter von *ensureIndex()* können noch Optionen zur asynchronen Erstellung des Index im Hintergrund (*'background' => true*) zur Vergabe eines Namens für den Index (*'name' => \$idx_name*) sowie die bereits bekannten Optionen *safe* und *timeout* übergeben werden, um das Resultat der Indexerstellung zu überprüfen.

Listing 14.37 Erstellen der unterschiedlichen Indexarten

```
// Gewöhnlicher Index
$coll->ensureIndex(['beschreibung' => 1]);
// Unique Index
$coll->ensureIndex(['produkt' => 1], ['unique' => true]);
// Unique Index, _Löschen_ der doppelten Dokumente
$coll->ensureIndex(['produkt_nr' => 1], ['unique' => true,
                                     'dropDups' => true]);
// Sparse Unique Index (Indexfelder können im Dokument fehlen)
$coll->ensureIndex(['lager_id' => 1], ['unique' => true,
                                     'sparse' => true]);
// Multikey Index; vorausgesetzt 'kategorie' ist ein Array-Feld
$coll->ensureIndex(['kategorie' => 1]);
// Geindex : Feld sollte ein Array vom Typ [x, y] sein, kein Objekt!
$coll->ensureIndex(['geschaeft_pos' => '2d']);
```

14.5.2 Verwenden von Indexen

Indexe in MongoDB haben die gleichen Eigenschaften wie Indexe in relationalen Datenbanken. Sie erhöhen die Lesegeschwindigkeit, verlangsamen aber das Schreiben, da die Indexstruktur zusätzlich aktualisiert werden muss. Eine wichtige Einschränkung in MongoDB ist, dass pro Abfrage nur ein einziger Index verwendet werden kann! Einzige Ausnahme: Wenn verschiedene Abfragen mit *\$or* verknüpft sind, verwendet MongoDB für jede der Teilabfragen den passenden Index.

Analog zu relationalen Datenbanken ist bei kombinierten Indexen die Reihenfolge der Felder im Index wichtig. Nur bei Suchabfragen, die auf einen Präfix der indizierten Felder abzielen, wird der Index verwendet. Folgende Beispielsuchabfragen für einen Index der Felder *typ*, *farbe*, *preis* (in dieser Reihenfolge) verdeutlichen diesen Punkt:

- {typ: "hut", farbe: "blau", preis: 39.9}: Index wird verwendet.
- {typ: "hut"} oder {typ: "hut", farbe: "blau"}: Index wird verwendet, da Suchanfragen ein Präfix der indizierten Felder darstellen.
- {farbe: "rot"} oder {preis: 39.9}: Index wird nicht verwendet, da die Abfragen kein Präfix der Indexfelder darstellen (*typ* wird nicht abgefragt).
- {typ: "hut", farbe: {\$gt: "Gelb"}} oder {typ: "hut", farbe: "blau", preis: {\$lt: 50}}: Index wird für Bereichsabfragen verwendet, hier ist die Bereichsangabe im letzten Teil des Index.
- {typ: {\$gt: "hut"}, farbe: "Gelb"} oder {typ: "hut", preis: {\$lt: 50}}: Auch hier kann der Index verwendet werden, allerdings müssen Indexeinträge unnötig durchsucht werden, was – je nach Verteilung der Daten in der Kollektion – Leistungseinbußen nach sich ziehen kann.

Indexabdeckung (covering index)

Wenn Sie in ein Ergebnis einer Suchanfrage nur Felder einbinden, die auch im Index erscheinen, der Index also die Ergebnisfelder vollständig abdeckt, muss MongoDB zur Erstellung des Ergebnisses nicht mehr auf die eigentlichen Dokumente zurückgreifen. Da der Index meist im Arbeitsspeicher gehalten wird, kann das Ergebnis ausschließlich mit Informationen im Arbeitsspeicher berechnet werden, dementsprechend schnell und performant ist die Berechnung. In Ihren eigenen Anwendungen sollten Sie daher versuchen, für wichtige Suchabfragen den Index so zu gestalten, dass alle Ergebnisfelder im Index vorkommen. Achten Sie dabei jedoch auf die Gesamtgröße des Index, sodass dieser immer noch vollständig im Arbeitsspeicher gehalten werden kann.

Listing 14.38 zeigt ein Beispiel: Für den angegebenen Index werden die ersten drei angeführten Suchabfragen nicht vollständig vom Index abgedeckt. Die dritte Suchabfrage ist nicht abgedeckt, weil standardmäßig auch noch der primäre Schlüssel (Feld `_id`) mit in die Ergebnisobjekte aufgenommen wird. Erst wenn der primäre Schlüssel – wie in der Suchabfrage 4 – explizit ausgenommen wird, kann das Ergebnis ausschließlich mit Informationen des Index erstellt werden.

Listing 14.38 Vom Index abgedeckte Suchabfrage

```
// Index
$coll->ensureIndex(['typ' => 1, 'farbe' => 'preis' => -1]);

// nicht vom Index vollständig abgedeckte Suchabfragen
$coll->find(['typ' => 'hut', 'farbe' => 'blau']);
$coll->find(['typ' => 'hut', 'farbe' => 'blau'],
  ['typ' => true, 'farbe' => true, 'beschreibung' => true]);
$coll->find(['typ' => 'hut', 'farbe' => 'blau'],
  ['typ' => true, 'farbe' => true, 'preis' => true]);

// vom Index abgedeckte Suchabfrage
$coll->find(['typ' => 'hut', 'farbe' => 'blau'],
  ['typ' => true, 'farbe' => true, 'preis' => true,
   '_id' => false]);
```



PRAXISTIPP: Wenn Sie den primären Schlüssel im Ergebnis benötigen, nicht aber für die Formulierung der Suchabfrage, können Sie den primären Schlüssel am Ende des Index anhängen, damit Sie eine vom Index vollständig abgedeckte Suchabfrage mit allen Geschwindigkeitsvorteilen haben.

14.5.3 Analysieren von Abfragen mit `explain()`

Das wichtigste Werkzeug zur Steigerung der Performanz bzw. zum Analysieren von Abfragen während der Entwicklung ist `explain()`. Es zeigt, ob für die Abfrage ein Index verwendet wird und wenn ja, welcher Index. Zudem lässt sich ablesen, wie viele Einträge und Dokumente MongoDB durchsuchen musste, um das Ergebnis zu erstellen. Tabelle 14.5 gibt einen Überblick über die wichtigsten Informationen zu `explain()`.

Tabelle 14.5 Informationen zu Suchabfragen von explain()

Feld	Beschreibung
<i>cursor</i>	Gibt an, ob und wenn ja, welcher Index verwendet wird. <i>BasicCursor</i> bedeutet, dass kein Index verwendet wird, <i>BtreeCursor index_name</i> , dass der genannte Index verwendet wird.
<i>indexOnly</i>	Falls <i>true</i> , ist das Ergebnis vollständig aus dem Index erstellbar (abdeckender Index, covering index)
<i>isMultiKey</i>	Gibt an, ob es sich bei dem Index um einen Multikey-Index handelt, der Index also, der indizierte Array-Felder enthält
<i>n</i>	Anzahl der zutreffenden, im Resultat enthaltenen Dokumente
<i>nscanned</i>	Anzahl der Indexeinträge, die durchsucht wurden
<i>nscannedObjects</i>	Anzahl der Dokumente, die durchsucht wurden
<i>Millis</i>	Benötigte Zeit in Millisekunden
<i>scanAndOrder</i>	Falls <i>true</i> , konnte der Index nicht verwendet werden, um das Resultat entsprechend der mit <i>sort()</i> angegebenen Reihenfolge zu sortieren.

Für die folgenden Beispiele werden 10 000 Produkteinträge, wie in Listing 14.39 beschrieben, verwendet. Jeder Eintrag hat drei Tags.

Listing 14.39 Aufbau der Produkteinträge für die folgenden Beispiele

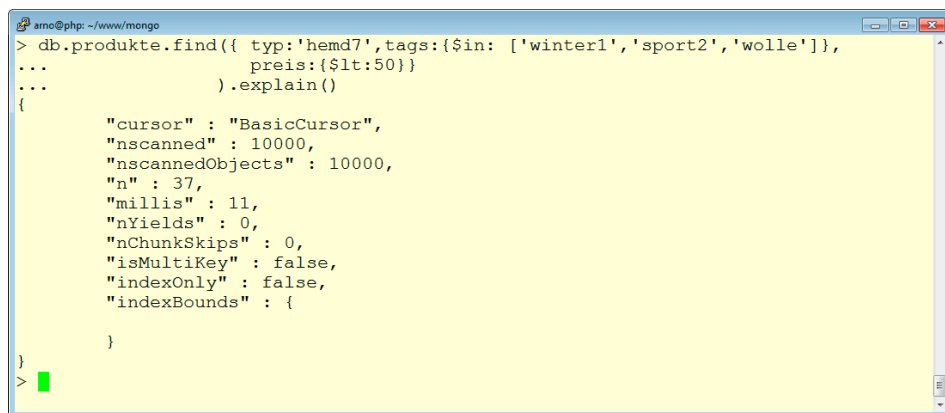
```
Dokumentaufbau:
{ _id: Zahl, typ: Typ, farbe: Farbe, preis: Preis, tags: Array(Tag) }
Mit:
Typ   = hut_, hemd_, krawatte_, schuhe_
Farbe = rot_, gelb_, blau_, violett_, schwarz_
Tag   = frühling_, sommer_, herbst_, winter_, freizeit_, elegant_,
        sport_, wolle_, polyester_, seide_
Preis = 12 € bis 83 €
("_" steht für eine Ziffer 0-9)
```

Suche ohne Index

Angenommen, wir suchen in der Produktkollektion nach Produkten eines bestimmten Typs, mit zumindest einem von mehreren angegebenen Tags und einem Preis unter einem Schwellwert. Eine mögliche Abfrage könnte dann so aussehen.

```
db.produkte.find({ typ: 'hemd7', preis: { $lt:50 }
                  tags: { $in: ['winter1','sport2','wolle'] } })
```

Das Ergebnis der Analyse ist in Bild 14.13 zu sehen. MongoDB musste alle 10 000 Produkte durchsuchen (*nscannedObjects*), um die 37 zutreffenden Einträge zu ermitteln.



```

> db.produkte.find({ typ:'hemd7',tags:{$in: ['winter1','sport2','wolle']},
...               preis:{$lt:50}}
...               ).explain()
{
  "cursor" : "BasicCursor",
  "nscanned" : 10000,
  "nscannedObjects" : 10000,
  "n" : 37,
  "millis" : 11,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {
  }
}
>

```

Bild 14.13 Analyse des Suchbefehls

Suche mit Indexen

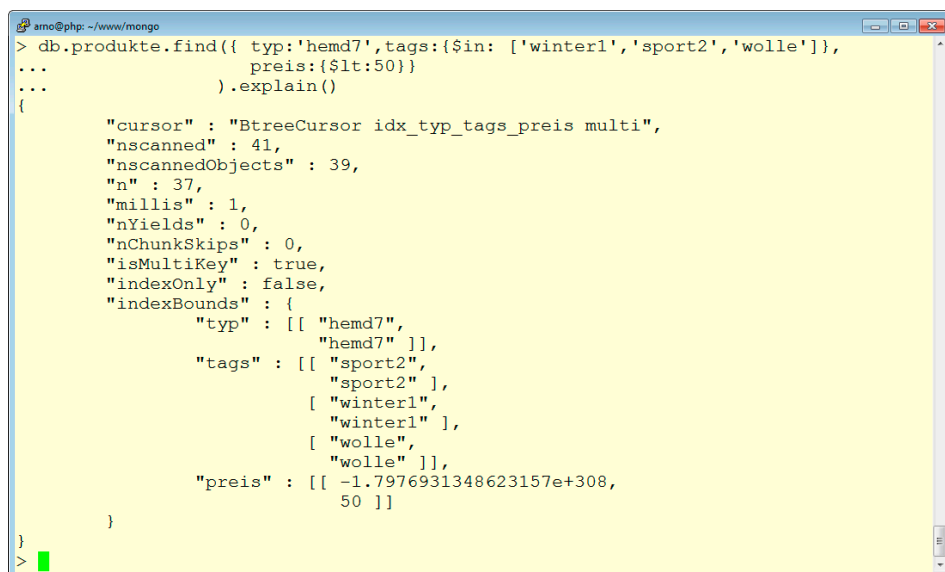
Wenn die Suchabfrage so gestaltet ist, sollten Felder, die mit Fixwerten verglichen werden, weit vorne stehen und Felder, die Bereichsabfragen enthalten, weiter hinten. Es bietet sich deshalb an, folgenden Index anzulegen.

```

db.produkte.ensureIndex({ typ: 1, tags: 1, preis: 1 },
{ name: 'idx_typ_tags_preis' })

```

Mit diesem Index zeigt *explain()* für dieselbe Suchanfrage eine deutliche Verbesserung, wie in Bild 14.14 zu sehen. Nur 41 Indexeinträge (*nscanned*) und 39 Dokumente (*nscannedObjects*) wurden untersucht, um die 37 Ergebniseinträge zu finden. Ein besseres Verhältnis zwischen untersuchten Einträgen und Resultat ist bei einer solchen Suchanfrage kaum möglich.



```

> db.produkte.find({ typ:'hemd7',tags:{$in: ['winter1','sport2','wolle']},
...               preis:{$lt:50}}
...               ).explain()
{
  "cursor" : "BtreeCursor idx_typ_tags_preis multi",
  "nscanned" : 41,
  "nscannedObjects" : 39,
  "n" : 37,
  "millis" : 1,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : true,
  "indexOnly" : false,
  "indexBounds" : {
    "typ" : [[ "hemd7",
               "hemd7" ]],
    "tags" : [[ "sport2",
               "sport2" ],
               [ "winter1",
               "winter1" ],
               [ "wolle",
               "wolle" ]],
    "preis" : [[ -1.7976931348623157e+308,
                 50 ]]
  }
}
>

```

Bild 14.14 Suchanfrage, die passenden Index verwendet

Zum Vergleich wird noch der gegenteilige Index angelegt, bei dem das *preis*-Feld an erster Stelle steht.

```
db.produkte.ensureIndex({ preis: 1, tags: 1, typ: 1 },
  { name: 'idx_preis_tags_typ' })
```

Wie in Bild 14.15 zu sehen, beträgt nun das Verhältnis zwischen Ergebniseinträgen und durchsuchten Indexeinträgen 1:7. Während bei niedriger Zugriffszahl das Verhältnis durchaus noch akzeptabel sein kann, kann sich bei hohen Zugriffszahlen daraus ein Performanzproblem ergeben. Wenn Sie sich unsicher sind oder mit dem Resultat eines Index nicht zufrieden sind, sollten Sie daher immer mehrere Indexvarianten ausprobieren. Es lohnt sich jedenfalls, die Struktur, Verteilung und Zusammensetzung der Daten zu kennen, um bessere Abfragen und zielgenauere Indexe zu schreiben.



Bild 14.15 Suchanfrage bei etwas ungünstigerem Index

Erzwingen des passenden Index mit hint()

Gelegentlich kann es passieren, dass MongoDB nicht den passenden Index auswählt. Wenn in der *produkte*-Kollektion beispielsweise nur folgende beiden Indexe angelegt sind.

```
db.produkte.ensureIndex({ preis: 1, typ: 1, tags: 1 }, { name: 'index_1' })
db.produkte.ensureIndex({ typ: 1, preis: 1, _id: 1 }, { name: 'index_2' })
```

Dann wählt MongoDB für die folgende Abfrage den ungünstigeren Index aus, bei der nach einem fixen Preis und einem bestimmten Textpräfix in *Typ* gesucht wird. Als Antwort soll nur der primäre Schlüssel der Ergebnisse retourniert werden.

```
db.produkte.find({preis: 19, typ: /^hemd/}, { _id: 1 })
```

Wie in Bild 14.16 zu sehen, wählt MongoDB den Index *index_1*, der einerseits ein Verhältnis von Ergebniseinträgen zu durchsuchten Indexeinträgen von 1:3 aufweist und auch das Ergebnis (das Feld *_id*) nicht im Index enthält (*indexOnly: false*).

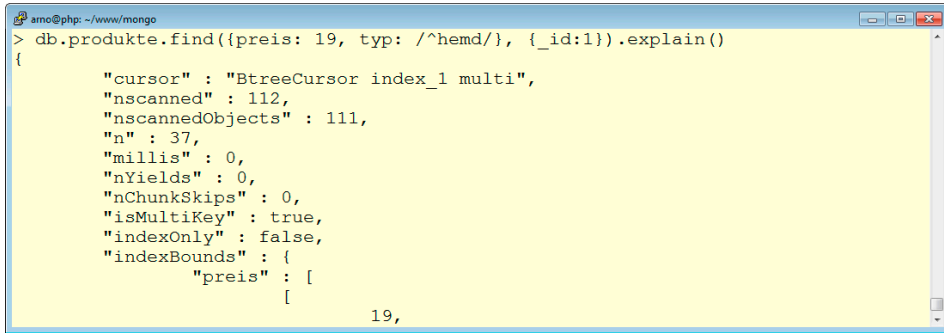


Bild 14.16 MongoDB wählt den ungünstigeren der beiden Indexe

Mit der Funktion *hint()* kann man MongoDB anweisen, einen anderen Index zu benutzen. Die Funktion wird als verketteter Aufruf an die *find()*-Funktion angehängt.

```

db.produkte.find({ preis: 19, typ: /^hemd/ },
  { preis: 1 }).hint("index_2").explain()

```

Bild 14.17 zeigt das Resultat: Nicht nur werden weniger Indexeinträge (*nscanned*) und Dokumente (*nscannedObjects*) durchsucht, sondern es muss kein Zugriff mehr auf die Dokumente stattfinden, *index_2* deckt die Suchanfrage vollständig ab (*indexOnly: true*). Bei hohen Zugriffszahlen kann das eine Abfrage wesentlich beschleunigen.

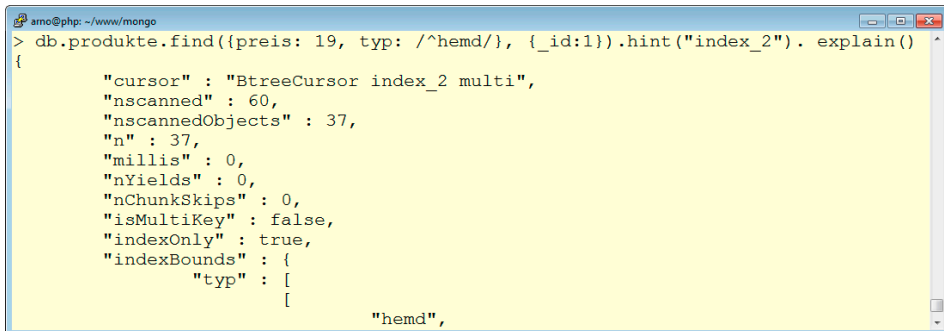


Bild 14.17 Mit *hint()* die Verwendung des besseren Index erzwingen

14.6 Map/Reduce

Map/Reduce ist ein zentrales Werkzeug zum Verarbeiten großer Datenmengen, da es sich gut parallelisieren lässt, fehlerresistent umgesetzt werden kann und auf eine Vielzahl von Problemstellungen anwendbar ist. MongoDB bietet eine leistungsstarke Umsetzung von Map/Reduce an, das sich gut für die Datenanalyse bzw. -berechnung eignet.

14.6.1 Prinzip

Map/Reduce hat seine Wurzeln in der funktionalen Programmierung und wurde unter anderem durch Google zur Bearbeitung von großen Datenmengen populär. Das zugrunde liegende Prinzip ist eine Aufteilung der Berechnung auf zwei Rechenschritte:

1. **Map:** Anwenden derselben Funktion auf die Elemente eines Arrays und Zwischenspeichern der Ergebnisse. Da die Map-Funktion das Ergebnis eines Elements unabhängig von den anderen berechnen kann, lässt sich die Rechenarbeit problemlos parallelisieren.
2. **Reduce:** Zusammenfassen der Zwischenergebnisse zu einem einzigen Resultat. Wenn die Reduce-Funktion idempotent ist, kann auch sie parallelisiert werden. Idempotent bedeutet hier, dass das Resultat am Ende immer dasselbe ist, egal in welcher Reihenfolge die Zwischenergebnisse abgearbeitet werden.

Beispiel: Summe von Quadraten

Wenn Sie beispielsweise die Summe der Quadrate von einer Liste von Zahlen bilden wollen, dann können Sie das, wie in Listing 14.40, mit einer *foreach*-Schleife tun. Während das Vorgehen bei kleinen Listen das praktikabelste ist, sind Listen mit einigen Hundert Millionen Zahlen damit nicht mehr effizient zu verarbeiten.

Listing 14.40 Bilden der Summe von Quadraten

```
<?php
$zahlen = [ 4, 3, 8, 7, 1, 3, 5, 9, 2, 6 ];
$summe = 0;
foreach ($zahlen as $x) {
    $summe += $x*$x;
}
echo "$summe\n";
?>
```

Map/Reduce teilt diese Berechnung in zwei Schritte auf: Zuerst die Berechnung der Quadrate, dann die Berechnung der Summe. Bild 14.18 zeigt den prinzipiellen Ablauf.

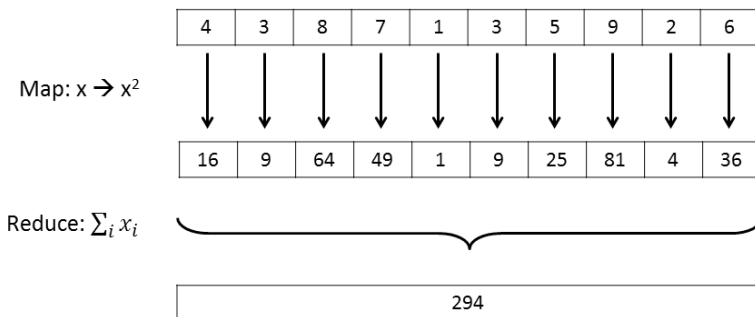


Bild 14.18 Prinzip von Map/Reduce

Listing 14.41 zeigt eine mögliche Umsetzung in PHP, welche die Funktionen *array_map()* und *array_reduce()* verwendet. Die Reduce-Funktion arbeitet dabei mit einem Akkumulator (*\$result*), in dem sukzessive die Summe gebildet wird. Der dritte Parameter von *array_*

reduce() ist der Initialwert von *\$resultat* und sollte neutral für die Berechnung sein; da eine Summe gebildet wird, ist die neutrale Zahl 0 (bei einer Multiplikation wäre es beispielsweise 1).

Listing 14.41 Map/Reduce für die Summe der Quadrate von Zahlen

```
<?php

function mapFun($x) {
    return $x*$x;
}

function reduceFun($result, $x) {
    return $result + $x;
}

$zahlen = [ 4, 3, 8, 7, 1, 3, 5, 9, 2, 6 ];
$mapped = array_map('mapFun', $zahlen);
$summe = array_reduce($mapped, 'reduceFun', 0);
echo "$summe\n";
?>
```

Parallelisieren

Der Vorteil von Map/Reduce ist, wie bereits erwähnt, die Möglichkeit, die Berechnungsschritte parallel auszuführen. Bild 14.19 zeigt eine beispielhafte Parallelisierung: Die Prozesse 1 bis 3 berechnen die Quadrate, die Prozesse 4 bis 7 berechnen die Summen. Prozesse 1 und 4 könnten beispielsweise auf einem eigenständigen Rechner arbeiten und Prozess 7 nur die Summe melden, analog für Prozess 3 und 6. Die Verteilung der Aufgaben auf (fast) beliebig viele Rechner ist bequem möglich.

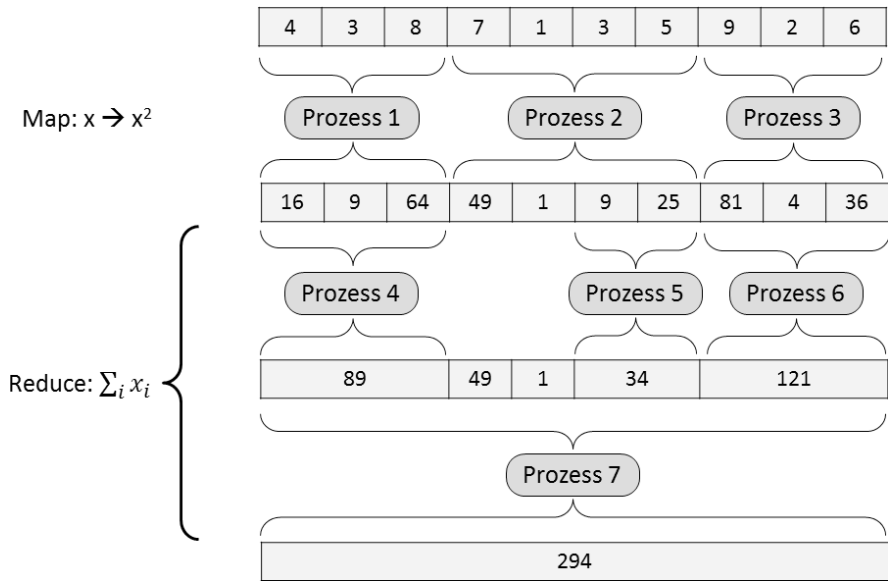


Bild 14.19 Parallelisierte Verarbeitung mit Map/Reduce

Bild 14.19 verdeutlicht auch, warum die Reduce-Funktion in beliebiger Reihenfolge der Array-Felder abgearbeitet werden können muss: Eine Parallelisierung ist nur möglich, wenn das Ergebnis von Prozess 6 nicht von Prozess 5 oder 4 abhängt. Auch die hierarchische Summierung (Prozesse 4 bis 6 in der ersten Stufe, Prozess 7 in der Zweiten) erfordert, dass die Reduce-Funktion idempotent ist, also in beliebiger Reihenfolge abgearbeitet werden kann.

Der gezeigte Aufbau bedingt auch, dass die Reduce-Funktion sowohl Daten der Map-Funktion als auch Daten zuvor ausgeführter Reduce-Funktionen verarbeiten kann. Am einfachsten ist das umzusetzen, wenn beide Funktionen (Map und Reduce) denselben Datentyp verwenden.

14.6.2 Ablauf und Parameter

MongoDB bietet ein fertiges Map/Reduce-Framework an, dem Sie nur noch die Map- und Reduce-Funktionen übergeben müssen. Sind die Daten auf mehrere Server verteilt, findet auch die Berechnung verteilt auf diesen Servern statt.

Ablauf

Bild 14.20 zeigt den Ablauf von Map/Reduce in MongoDB. Die *map()*-Funktion übergibt mithilfe von *emit()* den berechneten Wert an den nächsten Schritt. Jeder Wert wird dabei einem frei gewählten bzw. für die Problemstellung passenden Schlüssel zugeteilt (hier: die Zahlen 1 bis 3). *emit()* darf auch mehrfach pro Element in der Ausgangsmenge aufgerufen werden (im Bild zum Beispiel für die Buchstaben *n* und *d*), um weitere Resultate unter anderen Schlüsseln abzulegen. Wird *emit()* nicht aufgerufen, wird das Element ausgefiltert (im Beispiel wird Buchstabe *x* ausgefiltert).

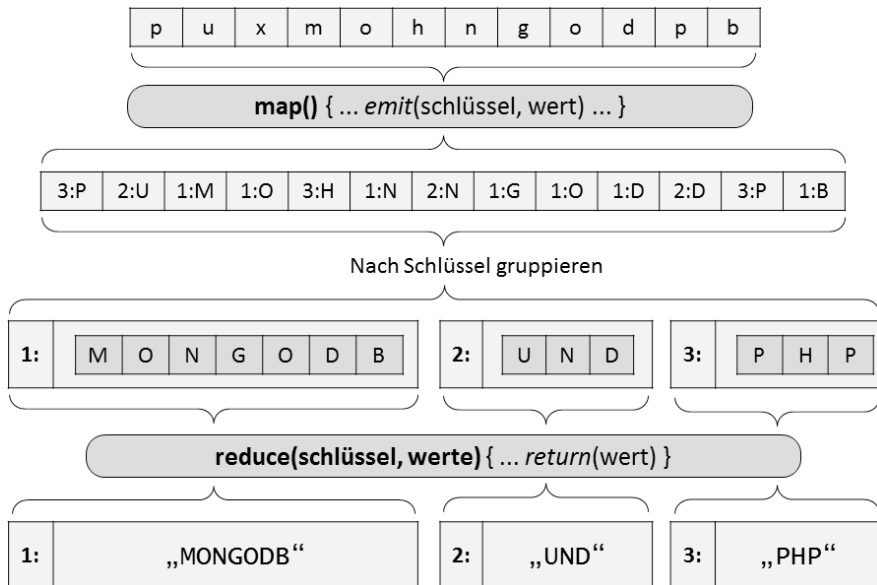


Bild 14.20 Ablauf von Map/Reduce in MongoDB

Die Felder werden dann nach dem Schlüssel gruppiert. Real findet kein eigener Gruppierungsschritt statt, sondern die Gruppierung ergibt sich durch die passenden Aufrufe der *reduce()*-Funktion. Die *reduce()*-Funktion bekommt Schlüssel und ein Array von Werten übergeben, aus denen *reduce()* ein aggregiertes Resultat berechnet. Aufgrund der möglichen hierarchischen Verarbeitung sollte das Resultat vom selben Datentyp sein wie die einzelnen Werte des vorhergehenden Schritts. Das Ergebnis ist eine Liste von Objekten, die gruppiert nach Schlüsseln die aggregierten Resultate enthalten.

Parameter

Dem *mapreduce*-Befehl werden beim Aufruf die notwendigen Parameter für die Berechnung übergeben. Eine Liste ausgewählter Parameter finden Sie in Tabelle 14.6. Mit dem Parameter *out* bestimmen Sie, ob und wie das Ergebnis der Berechnung gespeichert werden soll. Da auch das Ergebnis eine große Liste von Dokumenten sein kann, wird standardmäßig das Ergebnis in einer von Ihnen wählbaren Kollektion abgelegt.

Tabelle 14.6 Ausgewählte Parameter für den *mapreduce*-Befehl

Parameter	Beschreibung
<i>map</i>	JavaScript-Funktion für den Map-Schritt. Das zu bearbeitende Objekt kann über <i>this</i> adressiert werden, das Berechnungsergebnis wird mit <i>emit(schlüssel, wert)</i> ausgegeben, <i>emit()</i> kann beliebig oft (auch 0-mal) aufgerufen werden (siehe auch Abschnitt 14.6.3).
<i>reduce</i>	JavaScript-Funktion für den Reduce-Schritt. Bekommt Schlüssel als ersten Parameter und Array von Werten als zweiten Parameter übergeben und muss Ergebnis mit <i>return</i> zurückliefern.
<i>out</i>	Gibt an, wie das Ergebnis gespeichert werden soll, die wichtigsten Optionen sind: <ul style="list-style-type: none"> ▪ <i>{replace: <kollektion>}</i> oder <i><kollektion></i>: Speichert das Ergebnis in die angegebene Kollektion, vorhandene Daten in der Kollektion werden vollständig gelöscht ▪ <i>{merge <kollektion>}</i>: Fügt das Ergebnis in die angegebene Kollektion ein, alte Werte werden durch neue Werte überschrieben ▪ <i>{reduce <kollektion>}</i>: Fügt das Ergebnis in die angegebene Kollektion ein, alte und neue Werte werden mit der Reduce-Funktion kombiniert. Erlaubt inkrementelles Berechnen! ▪ <i>{inline: true}</i>: Das Ergebnis wird direkt an das Resultatobjekt zurückgegeben. Nur verwendbar, wenn das Ergebnis klein genug ist, um in ein MongoDB-BSON-Objekt zu passen.
<i>finalize</i>	Optional: JavaScript-Funktion, die am Ende des Reduce-Schritts für jedes Resultatelement aufgerufen wird
<i>query</i>	Optional: Suchabfrage, um nur definierte Elemente aus der Kollektion auszuwählen und in die Berechnung aufzunehmen

Beispiel

Als erstes Beispiel werden wir die Gruppierung aus Abschnitt 14.4.5 mit Map/Reduce umsetzen. Für Produkte, deren Lagerbestand und Preise bekannt sind, soll der Wert der lagernden Ware nach Preisen der Einzelstücke gestaffelt berechnet werden.

Listing 14.42 zeigt die dafür verwendete Map-Funktion. Das zu bearbeitende Objekt ist über *this* abrufbar. Als Schlüssel wird die Preiskategorie verwendet, nach der die Ergebnisse gruppiert werden sollen. Das mit *emit()* ausgegebene Objekt enthält den Wert des lagernden Produktes doppelt: Einmal als Summe, einmal als Maximum. Der einfache Trick ermöglicht es, dass die Ergebnisse von Map- und Reduce-Funktion genau dieselben Daten enthalten, eine spezielle Logik innerhalb der Reduce-Funktion entfällt also.

Listing 14.42 Map-Funktion – Schlüssel ist die Preiskategorie

```
function() {
    if ((typeof this.lagernd == 'undefined')
        || (typeof this.preis == 'undefined')) {
        return;
    }
    var key = Math.floor(this.preis/20) * 20;
    var txt = this.typ + ", Id: " + this._id + ", Stückpreis: "+this.preis;
    var total = this.lagernd * this.preis;
    emit(key, { sum: total, max: total, produkt: txt });
}
```

Der in Listing 14.43 gezeigten Reduce-Funktion werden Werte aus dem Map-Schritt als Array im Parameter *vals* übergeben. Das mit *return* weitergegebene Resultat summiert den Wert der lagernden Ware auf und führt parallel den maximalen Warenwert eines Produktes mit.

Listing 14.43 Reduce-Funktion – Aggregieren der Elemente

```
function (key, vals) {
    var res = { sum: 0, max: 0, produkt: '' };
    for (var i=0; i < vals.length; i++) {
        res.sum += vals[i].sum;
        if (vals[i].max > res.max) {
            res.max = vals[i].max;
            res.produkt = vals[i].produkt;
        }
    }
    return res;
}
```

Listing 14.44 enthält das vollständige Programm zur Berechnung des Lagerwertes, gruppiert nach Preiskategorie. PHP stellt keine eigene Methode für Map/Reduce zur Verfügung, stattdessen wird der Befehl *MongoDB::command()* verwendet, der als ersten Parameter die Funktion *mapreduce* und die zu verwendende Kollektion enthält. Die Ausgabe des Resultats erfolgt in der Kollektion *produkte_nach_preis* (Modus: *replace*), aus der nachfolgend mit *find()* und *sort()* die Werte, sortiert nach der Preiskategorie, ausgelesen und ausgegeben werden.

Listing 14.44 Vollständiges Programm zur Berechnung des Lagerwertes

```
<?php
$mng = new Mongo();
$db = $mng->selectDB('styleShop');
$coll = $db->selectCollection('produkte');
```

```

$mapJS = ' ... '; // siehe Listing oben
$reduceJS = ' ... '; // siehe Listing oben

$res = $db->command(['mapreduce' => 'produkte',
                    'map' => new MongoCode($mapJS),
                    'reduce' => new MongoCode($reduceJS),
                    'out' => 'produkte_nach_preis']);

print_r($res);

$res = $db->selectCollection('produkte_nach_preis')
        ->find([])->sort(['_id'=>1]);
foreach ($res as $grp) {
    $val = $grp['value'];
    echo "Produkte ab $grp[_id] €\n";
    echo "Lagerbestand: $val[sum]€\n";
    echo "Größter Bestand ($val[max]€): $val[produkt]\n\n";
}
?>

```

Bild 14.21 zeigt das Resultat des *mapreduce*-Befehls, den berechneten Lagerbestand nach Preiskategorie sowie den größten Bestand eines einzelnen Produkts. Am Ergebnisobjekt lässt sich ablesen, dass für jedes Eingangsdokument (*input*) genau ein Zwischenergebnis (*emit*) erstellt worden ist und vier Preiskategorien (*output*) erzeugt worden sind.

```

arno@php: ~/www/mongo
arno@php:~/www/mongo$ php mapreduce1.php
Array
(
    [result] => produkte_nach_preis
    [timeMillis] => 40
    [counts] => Array
        (
            [input] => 9
            [emit] => 9
            [reduce] => 2
            [output] => 4
        )
    [ok] => 1
)

Produkte ab 0 €
Lagerbestand: 681.2€
Größter Bestand (457.7€): Krawatte, Id: 111, Stückpreis: 19.9

Produkte ab 20 €
Lagerbestand: 3682.8€
Größter Bestand (2242.5€): Hemd, Id: 101, Stückpreis: 29.9

Produkte ab 40 €
Lagerbestand: 239.6€
Größter Bestand (239.6€): Hut, Id: 123, Stückpreis: 59.9

Produkte ab 60 €
Lagerbestand: 139.8€
Größter Bestand (139.8€): Hut, Id: 122, Stückpreis: 69.9

arno@php:~/www/mongo$

```

Bild 14.21 Resultat des mapreduce-Befehls und berechnete Gruppierung

14.6.3 Inkrementelle Berechnung

Dadurch, dass MongoDB die Resultate der Map/Reduce-Berechnung in einer Kollektion speichert, kann eine Berechnung stückweise mit neuen Daten fortgesetzt werden, ohne dass die gesamte Berechnung erneut durchgeführt werden muss. Das ist besonders bei großen Datenmengen, die mit der Zeit wachsen, ein nicht zu unterschätzender Vorteil.

Als Beispiel für eine inkrementelle Berechnung soll eine Liste jener Personen erstellt werden, die im Webshop zu Produkten die meisten Kommentare abgegeben haben. Zudem soll die durchschnittliche Punkteanzahl der Bewertung ausgegeben werden. Das Programm wird in regelmäßigen Intervallen aufgerufen, um die neuste Statistik zu erhalten.

Mehrfaches emit()

Listing 14.45 zeigt die für das Beispiel wesentlichen Felder der verwendeten Produktdokumente. Die Kommentare befinden sich als eingebettete Dokumente im Feld *kommentare*. Da für die Kommentarstatistik nach dem Benutzer gruppiert wird, muss *emit()* in der Map-Funktion pro Produktdokument mehrfach aufgerufen werden, pro Kommentar einmal.

Listing 14.45 Aufbau der Produktdokumente in MongoDB

```
{ _id: 5, beschreibung: 'Hemd, einfarbig, schwarz',
  'kommentare' : [
    { benutzer: 'Max', bewertung: 5,
      datum: ISODate('2012-06-26T10:08:26Z') },
    { benutzer: 'Evi', bewertung: 3,
      datum: ISODate('2012-08-27T12:18:27Z') },
    { benutzer: 'Christina', bewertung: 3,
      datum: ISODate('2012-08-27T15:00:46Z') } ]
}
```

Für die inkrementelle Berechnung der Kommentarstatistik wird ein Zeitraum angegeben, in dem die Kommentare liegen müssen, damit kein Kommentar doppelt gezählt wird. Da in einem Produkt sowohl Kommentare innerhalb als auch außerhalb des Zeitraums liegen können, muss in der Map-Funktion selbst überprüft werden, ob ein Kommentar im gewünschten Zeitraum liegt. Die Funktion in Listing 14.46 enthält deshalb eine entsprechende Abfrage, die Variablen *from* und *to* werden von PHP aus an Werte gebunden. Für jeden Kommentar innerhalb des Zeitraums wird *emit()* mit dem Benutzer als Schlüssel aufgerufen.

Listing 14.46 Map-Funktion – Pro Kommentar wird ein emit() ausgelöst

```
function() {
  this.kommentare.forEach(function(i) {
    if (i.datum >= from && i.datum < to) {
      emit(i.benutzer, { punkte: i.bewertung, anzahl: 1 });
    }
  });
}
```

Die Reduce-Funktion in Listing 14.47 gestaltet sich einfach: In ihr werden lediglich die Punkte und die Anzahl der Kommentare zusammengezählt. Die einfache Logik ergibt sich daraus, dass die von Map erzeugten Objekte bereits das Feld *anzahl* enthalten und deshalb einfach aufsummiert werden können.

Listing 14.47 Reduce-Funktion – Zusammenzählen der Punkte und Kommentaranzahl

```
function (key, vals) {
  var res = { punkte: 0, anzahl: 0 };
  vals.forEach(function(i) {
    res.punkte += i.punkte;
    res.anzahl += i.anzahl;
  });
  return res;
}
```

Zur Berechnung der durchschnittlichen Punkteanzahl pro Kommentar wird am Ende des Reduce-Schrittes die *finalize()*-Methode wie in Listing 14.48 aufgerufen. Für die inkrementelle Berechnung ist es wichtig, dass die Gesamtpunkteanzahl und die Gesamtanzahl der Kommentare erhalten bleiben. Aus diesem Grund wird der Durchschnitt in einer eigenen Variable abgelegt.

Listing 14.48 Finalize – Durchschnittspunkte pro Kommentar berechnen

```
function (key, val) {
  val.durchschnitt = val.punkte / val.anzahl;
  return val;
}
```

Listing 14.49 enthält das vollständige Programm zur inkrementellen Berechnung der Kommentarstatistik. Die Datums Grenzen werden über Kommandozeilenparameter übergeben. Der *mapreduce*-Befehl enthält nun auch einen *query*-Parameter, um nur jene Produkte auszuwählen, die Kommentare später als das Startdatum aufweisen. Auf eine genaue Eingrenzung wurde hier verzichtet, da im Regelfall die inkrementelle Berechnung nur für neuere Kommentare durchgeführt wird.

Wichtig für die inkrementelle Berechnung ist das Setzen des *out*-Parameters auf *reduce*. Diese Option weist *mapreduce()* an, vorhandene Werte in der Kollektion *kommentar_statistik* mit den neu berechneten Werten über die angegebene Reduce-Funktion zusammenzuführen.

Listing 14.49 Programm zur inkrementellen Berechnung der Kommentarstatistik

```
$mapJS = ' ... '; // siehe Listing oben
$reduceJS = ' ... '; // siehe Listing oben
$finalizeJS = ' ... '; // siehe Listing oben


$fromDt = new MongoDate(strtotime($_SERVER['argv'][1]));
$toDt = new MongoDate(strtotime($_SERVER['argv'][2]));
$res = $db->command(['mapreduce' => 'produkte2',
  'query' => ['kommentare.datum' => ['$gte' => $fromDt]],
  'map' => new MongoCode($mapJS, ['from' => $fromDt,
    'to' => $toDt]),
  'reduce' => new MongoCode($reduceJS),
  'finalize' => new MongoCode($finalizeJS),
  'out' => ['reduce' => 'kommentar_statistik']]);
print_r($res['counts']);
$res = $db->selectCollection('kommentar_statistik')
  ->find([])->sort(['value.anzahl'=>-1])->limit(3);
foreach ($res as $grp) {
```

```

$val = $grp['value'];
echo "Benutzer: $grp[_id]\n",
    "$val[anzahl] Kommentare, ",
    "durchschnittlich $val[durchschnitt] Punkte.\n\n";
}
?>

```

Bild 14.22 zeigt das Ergebnis des ersten Durchlaufs der Berechnung der Kommentarstatistik. Aus dem ausgegebenen Ergebnisobjekt kann man ablesen, dass für vier Produktdokumente (*input*) *emit()* sechsmal aufgerufen wurde, da sechs Kommentare im gewünschten Zeitraum lagen. In Summe gab es fünf unterschiedliche Benutzer (*output*), die in diesem Zeitraum Kommentare abgegeben haben.



```

arno@php: ~/www/mongo$ php mapreduce2.php 2012-06-01 2012-09-01
Array
(
    [input] => 4
    [emit] => 6
    [reduce] => 1
    [output] => 5
)
Benutzer: Christina
2 Kommentare, durchschnittlich 4 Punkte.

Benutzer: Arno
1 Kommentare, durchschnittlich 5 Punkte.

Benutzer: Evi
1 Kommentare, durchschnittlich 3 Punkte.

arno@php: ~/www/mongo$

```

Bild 14.22 Erster Durchlauf

Das Ergebnis des zweiten Durchlaufs ist in Bild 14.23 zu sehen. Im angegebenen Zeitraum wurden bei zwei Produkten (*input*) in Summe sechs Kommentare (*emit*) abgegeben. Es gab keine neuen Benutzer in diesem Zeitraum, da *output* unverändert auf 5 geblieben ist.



```

arno@php: ~/www/mongo$ php mapreduce2.php 2012-09-01 2012-10-31
Array
(
    [input] => 2
    [emit] => 6
    [reduce] => 2
    [output] => 5
)
Benutzer: Christina
4 Kommentare, durchschnittlich 3.75 Punkte.

Benutzer: Max
3 Kommentare, durchschnittlich 2.3333333333333 Punkte.

Benutzer: Evi
2 Kommentare, durchschnittlich 3 Punkte.

arno@php: ~/www/mongo$

```

Bild 14.23 Zweiter Durchlauf

■ 14.7 Replikation und Verfügbarkeit

MongoDB kann Server in Replikationsgruppen (*Replication Set*) anordnen, damit bei Ausfall eines Servers die Verfügbarkeit der Datenbank trotzdem gewährleistet ist. Replikationsgruppen sind auch für die Datensicherheit essenziell: Da dieselben Daten auf mehreren Rechnern gespiegelt sind, gehen keine Daten bei einem Serverausfall verloren. Zwar speichert MongoDB die Datenbank auf das Laufwerk, es gibt aber gelegentlich Berichte, dass nach einem Crash, die Datendateien teilweise korrupt sind.

14.7.1 Funktionsweise

In MongoDB basiert die Replikation auf einem Master-Slave-Prinzip, mit der Besonderheit, dass der Master nicht fix ist, sondern von den beteiligten Servern gewählt wird. Aus diesem Grund heißt der Master in MongoDB auch *Primary* und die Slaves *Secondary*. Schreibzugriffe finden ausschließlich auf dem Primary statt, üblicherweise die Lesezugriffe auch, es sei denn, Sie erlauben Lesezugriffe auf die Secondaries mit der Funktion *setSlaveOkay()*, die auf Ebene der Verbindung, Datenbank, Kollektion oder des Cursors aufgerufen werden kann und in diesem Kontext Gültigkeit behält.

Server- und Netzwerkfehler

Die Server einer Replikationsgruppe überwachen sich gegenseitig. Wenn der Primary ausfällt bzw. über das Netzwerk nicht erreichbar ist, wird jener Secondary zum Primary gewählt, der den aktuellsten Datenstand enthält; allerdings nur, wenn die Mehrheit der Secondaries dafür stimmen. Mit dieser Mehrheitsregel wird verhindert, dass bei einer Netzwerkteilung zwei Server als Primary ernannt werden können, da die Mehrheit der Server nur in einer Netzwerkhälfte sein kann. Aus diesem Grund sollte auch immer eine ungerade Anzahl an Servern in der Replikationsgruppe sein. Hat in einem getrennten Netzwerk keine der verbleibenden Teilgruppen eine absolute Mehrheit, wird kein Primary gewählt, es sind dann nur noch lesende Zugriffe möglich.

Verwendung

Replikationsgruppen dienen hauptsächlich der Verfügbarkeit und Datensicherheit, erst in zweiter Linie der Skalierung von Lesezugriffen, denn dafür empfiehlt MongoDB Sharding zu verwenden (siehe Abschnitt 14.8). Bei Lesezugriffen auf die Secondaries können nämlich noch alte Datensätze retourniert werden, da es bei der Replikation von Primary kleine Verzögerungen geben kann.

14.7.2 Anlegen einer Replikationsgruppe

Um eine Replikationsgruppe anzulegen, werden die *mongod*-Server mit der Option *-replSet* gestartet, wie in Listing 14.50 zu sehen. Zu Entwicklungszwecken können Sie mehrere Instanzen am selben Server starten, in der Produktion sollten Sie die Instanzen auf verschiedene Server verteilen.

Listing 14.50 Start von fünf lokalen mongod-Instanzen als Replikationsgruppe

```

mongod -replSet myApp --port 27017 --fork --dbpath /var/lib/mongodb/srv1
mongod -replSet myApp --port 27018 --fork --dbpath /var/lib/mongodb/srv2
mongod -replSet myApp --port 27019 --fork --dbpath /var/lib/mongodb/srv3
mongod -replSet myApp --port 27020 --fork --dbpath /var/lib/mongodb/srv4
mongod -replSet myApp --port 27021 --fork --dbpath /var/lib/mongodb/srv5

```

Zu Beginn darf nur einer der Server bereits Daten enthalten oder alle leer sein. Verbinden Sie sich über die Mongo-Shell zu dem Server, der bereits Daten enthält, sonst zu irgendeinem Server, und initialisieren Sie die Gruppe wie in Listing 14.51 gezeigt. Die *_id* muss den gleichen Wert wie der *replSet*-Kommandozeilenparameter haben.

Listing 14.51 Replikationsgruppe starten

```

rs.initiate({_id: 'myApp', members: [
    { _id: 0, host: 'localhost:27017' },
    { _id: 1, host: 'localhost:27018' },
    { _id: 2, host: 'localhost:27019' },
    { _id: 3, host: 'localhost:27020' },
    { _id: 4, host: 'localhost:27021' } ]});

```

Die Server nehmen nun Kontakt miteinander auf, etablieren die Gruppe und wählen den Primary-Server aus. Sie können mit *rs.status()* den Zustand der Replikationsgruppe prüfen.

14.7.3 Verwenden in PHP

In PHP können Sie die Replikationsgruppe wie einen gewöhnlichen Server ansprechen. Es ist dabei ausreichend, sich zu irgendeinem Server zu verbinden, denn dieser übermittelt dem PHP-Treiber die Daten aller Server der Gruppe, insbesondere, welcher der Server der Primary ist. Infolge verbindet sich der PHP-Treiber mit dem Primary.

Als Option sollten Sie bei der Verbindung mit PHP die Replikationsgruppe angeben, mehrere Server können mit Beistrich getrennt werden, wie in Listing 14.52 zu sehen.

Listing 14.52 Verbinden zu einer Replikationsgruppe von PHP aus

```

$mng = new Mongo('mongodb://localhost:27017,localhost:27020',
    ['replicaSet' => 'myApp']);

```

Wenn Sie mit *setSlaveOkay()* Leseverbindungen zu den Secondaries zulassen, wählt der PHP-Treiber jenen Server aus, der am schnellsten (mit kleinster Round-Trip-Time) über das Netzwerk zu erreichen ist.

Schreiboperationen

Standardmäßig liefert der PHP-Treiber eine Erfolgsmeldung, sobald die Schreiboperation auf dem Primary erfolgreich ausgeführt worden ist. Wenn Sie sicherstellen wollen, dass die Daten auf einer definierten Anzahl von Secondaries repliziert worden ist, können Sie den *w*-Parameter verwenden, der für die Datenbank oder für die Kollektion gesetzt werden kann. Alternativ können Sie direkt bei der Schreiboperation eine Zahl als *safe*-Parameter angeben. Die drei Varianten werden in Listing 14.53 gezeigt.

Listing 14.53 Angabe der notwendigen erfolgreichen Replikationen

```
$db = $mng->selectDB('styleShop');
$db->w = 2; // Primary + 1 Secondary
$coll = $db->selectCollection('produkte'); // übernimmt Einstellung von DB
$coll->w = 3; // Primary + 2 Secondaries
$coll->insert(['typ' => 'hut']); // übernimmt Einstellung von Kollektion
$coll->insert(['typ' => 'hut'], ['safe' => true]); // ditto
$coll->insert(['typ' => 'hut'], ['safe' => 4]); // Primary + 3 Secondaries
```



PRAXISTIPP: Üblicherweise ist es ausreichend, wenn die Bestätigung von einem zusätzlichen Secondary vorhanden ist ($w=2$), um Datenverlust zu vermeiden. Wenn Sie sicherstellen wollen, dass bei einem Netzwerksplit jedenfalls aktuelle Daten in der größeren Hälfte verfügbar sind, müssen Sie w auf die Mehrheit der Server setzen, bei fünf Servern in der Gruppe auf 3, bei sieben Servern auf 4. Allerdings kann die Performanz von Schreiboperationen bei hohen w -Werten abnehmen.

Fehlerbehandlung

Solange Sie nicht die *safe*-Option angeben oder unmittelbar nachfolgend mit *db.getLastError()* einen Fehler abfragen, können Sie nicht sicher sein, dass die Schreiboperation erfolgreich war. Insbesondere kann es dann passieren, dass PHP – bei bestehender oder persistenter Verbindung – den Ausfall des Primary-Servers nicht mitbekommt. Alle nachfolgenden Schreiboperationen gehen dann ins Leere. Es ist daher anzuraten, bei wichtigen Schreiboperationen auf Fehler zu prüfen.

Benötigt bei hohem w -Wert die Bestätigung der Secondary zu lange, erhalten Sie eine *MongoCursorException*. Achtung: Die Exception ist nur ein Indiz dafür, dass innerhalb des Zeitlimits nicht die gewünschte Anzahl an Servern die Schreiboperation bestätigt haben. Sie ist kein Indiz dafür, dass die Schreiboperation vollständig fehlgeschlagen ist. Die Daten können also beispielsweise im Primary gespeichert sein, allerdings bestätigten die Secondary die Operation nicht rechtzeitig.

14.8 Sharding und Skalierung

Die problemlose Skalierung auf viele Hundert Server, um Daten und Zugriffe zu verteilen, ist ein typisches Merkmal von NoSQL-Datenbanken – MongoDB macht hier keine Ausnahme. MongoDB wendet horizontales Partitionieren an: Die Daten einer Kollektion werden entsprechend eines definierten Merkmals – dem Sharding-Schlüssel – auf die Server verteilt. Passend konfiguriert weist MongoDB keinen Single Point of Failure auf, und die Daten werden entsprechend ihrer Struktur und Zugriffe auf den Servern stetig ausbalanciert. Hinzufügen und Entfernen von Servern ist für die Anwendung transparent.

14.8.1 Funktionsweise

Beim Sharding sind neben den *Shard* genannten Datenservern auch Konfigurationsserver und Distributionsserver (*mongos*) eingebunden. Bild 14.24 zeigt den Aufbau einer Sharding-Infrastruktur in MongoDB. Die Server *Shard 1* bis *Shard 4* sind die Datenserver, auf welche die zu speichernden Daten aufgeteilt werden. Der Konfigurationsserver speichert, welche Daten auf welcher *Shard* liegen. Es kann auch mehrere Konfigurationsserver geben, selten sind es aber mehr als drei. Die Zugriffe der Clients werden über einen leicht gewichtigen Distributionsserver (*mongos*) auf die passende *Shard* weitergeleitet. Auch *mongos* sollten mehrfach vorhanden sein; typischerweise könnte ein *mongos*-Prozess auf jedem Webserver laufen, dann kann sich PHP lokal verbinden.

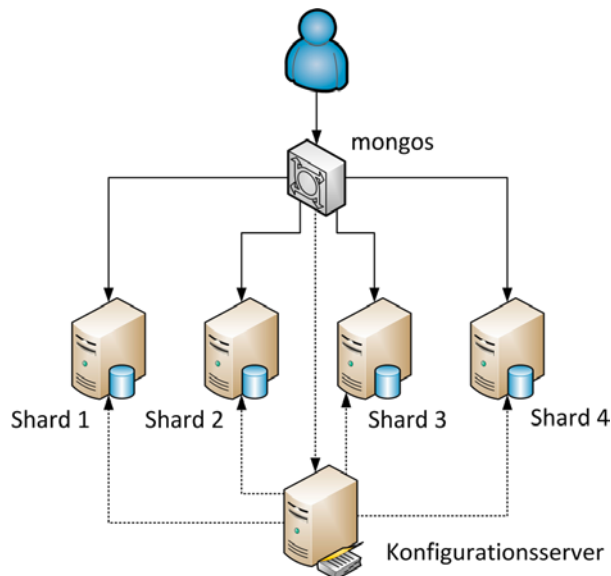


Bild 14.24 Aufbau einer Sharding-Infrastruktur

Für die Verfügbarkeit und Datensicherheit ist es möglich, dass jede *Shard* nicht ein einzelner Rechner, sondern eine Replikationsgruppe ist. Damit können auch große Installationen, bei denen mit dem Ausfall von Servern gerechnet werden muss, sicher betrieben werden.

14.8.2 Sharding konfigurieren

Um Sharding aufzusetzen, müssen *Shard*-Server, Konfigurations- und Distributionsserver gestartet werden. Listing 14.54 zeigt die notwendigen Kommandozeilenparameter zum Starten von zwei *Shard*-Servern und je einem Konfigurations- und Distributionsserver auf einem einzigen Entwicklungsrechner. Der Parameter *chunkSize* (in MB) gibt die Größe der Blöcke an, die auf die *Shards* verteilt werden. MongoDB verteilt nicht einzelne Datensätze, sondern Daten in Blöcken (*chunks*) auf die *Shards*. Standardmäßig liegt dieser Wert bei 64 MB, für das Beispiel – um Sharding zu erzwingen – wird der Wert auf 1 MB gesetzt.

Listing 14.54 Starten der Shard-, Konfigurations- und Distributionsserver

```

mongod --shardsvr --port 27017 --fork --dbpath /src/mongodb/data/shard1
mongod --shardsvr --port 27018 --fork --dbpath /src/mongodb/data/shard2
mongod --configsvr --port 27019 --fork --dbpath /src/mongodb/data/shard_cfg
mongos --configdb localhost:27019 --chunkSize 1 --port 27020 --fork

```

Damit die Server in einem Sharding-Verbund zusammenarbeiten, müssen infolge die Shard-Server konfiguriert werden. Listing 14.55 zeigt die notwendigen Mongo-Shell-Befehle, um die beiden Server als Shards hinzuzufügen. Die Kommandos müssen in eine Shell zum *mongos*-Distributionsserver (im Beispiel Verbindung mit `mongo localhost:27020` hergestellt) eingegeben werden.

Nach der Konfiguration der Server muss Sharding sowohl auf Ebene der Datenbank als auch für die einzelne Kollektion konfiguriert werden – nur dann wird eine Kollektion auf die Shards verteilt. Beim Befehl *shardcollection* wird der Sharding-Schlüssel angegeben, auf dessen Basis die Datensätze in Blöcken gruppiert und auf die Shards verteilt werden. Im Beispiel wird dazu Produkttyp und Preis genommen. Die Wahl des Sharding-Schlüssels bestimmt wesentlich über das Verhalten von MongoDB bei hohen Zugriffszahlen (siehe Abschnitt 14.8.4).

Listing 14.55 Konfiguration des Shardings (Shell-Verbindung mit mongos)

```

db.adminCommand({ addShard : "localhost:27017"})
db.adminCommand({ addShard : "localhost:27018"})
db.adminCommand({ enablesharding : 'styleShop'})
db.adminCommand({ shardcollection : 'styleShop.produkte',
                    key : {typ: 1, preis: 1} })

```

14.8.3 Verwendung

Zur Verwendung müssen Sie nichts weiter tun, als sich zur *mongos*-Instanz zu verbinden und die gewünschten Befehle zu senden. Mongos verhält sich wie ein normaler MongoDB-Server, es sind keine speziellen Parameter oder Optionen zu berücksichtigen.

Listing 14.56 zeigt die Aufteilung bei einer leeren Kollektion: Es existiert nur ein Chunk, der alle Einträge enthalten kann.

Listing 14.56 Initiale Shard-Aufteilung (Shell-Verbindung zu mongos)

```

> use config
> db.chunks.find({}, {_id:0, shard:1, min:1, max:1})
{ "min" : { "typ" : { $minKey : 1 }, "preis" : { $minKey : 1 } },
  "max" : { "typ" : { $maxKey : 1 }, "preis" : { $maxKey : 1 } },
  "shard" : "shard0000" }

```

Listing 14.57 fügt nun einige Datensätze in die Kollektion ein, um die beim Start angegebene Blockgröße von 1 MB zu überschreiten. Wie Sie sehen, müssen Sie sich lediglich zur *mongos*-Instanz verbinden (im Beispiel: *localhost:27020*), das Sharding ist transparent für die PHP-Applikation.

Listing 14.57 Einfügen von 2500 Datensätzen je 1 KB groß

```
<?php
$mng = new Mongo('mongodb://127.0.0.1:27020');
$db = $mng->selectDB('styleShop');
$coll = $db->selectCollection('produkte');
$typ = ['hut', 'krawatte', 'hemd', 'schuhe'];
for ($i = 0; $i < 25; $i++) {
    $prod = [];
    for ($j = 1; $j < 100; $j++) {
        $prod[] = ['typ' => $typ[$j % 4], 'preis' => $i%100,
                    'beschreibung' => str_repeat('viel text', 100)];
    }
    $coll->batchInsert($prod);
}
?>
```

Listing 14.58 zeigt den Zustand der Shards nach dem Einfügen der Beispieldatensätze. Vier Chunks sind angelegt worden, jeweils zwei auf jedem Shard-Server. Wenn Sie *db.chunks.find()* kurz nach dem Einfügen aufrufen, kann es sein, dass noch alle Chunks auf dem ersten Shard-Server liegen. Nach kurzer Zeit balanciert MongoDB dieses Ungleichgewicht aber aus.

Listing 14.58 Sharding nach dem Einfügen der Datensätze

```
> use config
> db.chunks.find({}, {_id:0, shard:1, min:1, max:1})
{ "min" : { "typ" : { $minKey : 1 }, "preis" : { $minKey : 1 } },
  "max" : { "typ" : "hemd", "preis" : 0 },
  "shard" : "shard0001" }
{ "min" : { "typ" : "hemd", "preis" : 0 },
  "max" : { "typ" : "hut", "preis" : 8 },
  "shard" : "shard0001" }
{ "min" : { "typ" : "schuhe", "preis" : 5 },
  "max" : { "typ" : { $maxKey : 1 }, "preis" : { $maxKey : 1 } },
  "shard" : "shard0000" }
{ "min" : { "typ" : "hut", "preis" : 8 },
  "max" : { "typ" : "schuhe", "preis" : 5 },
  "shard" : "shard0000" }
```

Für die PHP-Anwendung macht es keinen Unterschied, wo die Daten liegen und auf wie viele Shard-Server diese verteilt sind. Es ist also nicht notwendig, das Programm anzupassen, wenn Sie die Anzahl der Shard-Server erhöhen oder Konfigurationsserver hinzufügen.

14.8.4 Sharding-Schlüssel auswählen

Die Leistungsfähigkeit einer auf Shards verteilten Kollektion hängt wesentlich von der Wahl des Shard-Schlüssels, auf dessen Basis die Daten verteilt werden, ab. Um hohe Zugriffszahlen zu ermöglichen, sollten sich die Zugriffe möglichst auf die Shards verteilen. Das ist aber nur möglich, wenn die Anfragen den Shard-Schlüssel oder zumindest Teile davon enthalten. In diesem Sinne verhalten sich Shard-Schlüssel gleich wie Indexe: Nur wenn die Anfrage ein Präfix des Schlüssels enthält, kann sie gezielt einzelne Shards auswählen. In allen ande-

ren Fällen geht die Anfrage an *alle* Shards, und der Nutzen des Shardings zur Verteilung der Zugriffe geht verloren.

Wenn – wie in den Beispielen oben – der Shard-Schlüssel Typ und Preis eines Produktes umfasst, dann haben die angeführten Anfragen mit *find()* folgende Charakteristika:

- `{typ: "hut", preis: 39.9}`: Es wird genau eine Shard angesprochen.
- `{typ: "hut"}` oder `{typ: "hut", farbe: "blau"}`: Je nach Aufteilung der Daten werden ein oder mehrere Shards angesprochen (alle die Hüte enthalten), nicht aber alle Shards.
- `{farbe: "rot"}` oder `{preis: 39.9}`: Anfrage geht parallel an alle Shards.
- `sort({typ:1, preis:1})`: Die Shards werden sequenziell abgefragt, wird die Ausgabe limitiert, sind nur jene Shards betroffen, von denen Resultate abgefragt worden sind.
- `sort({farbe: 1, material: 1})`: In allen Shards wird parallel sortiert und dann das Ergebnis kombiniert. Kann wünschenswert sein, um große Datenmengen zu sortieren, sollte aber nur selten aufgerufen werden.
- `ensureIndex({farbe: 1, material: 1})`: Das Anlegen eines Index bezieht ebenfalls alle Shards parallel mit ein.

Da eine Kollektion nur auf eine Art auf Shards aufgeteilt werden kann, sollten Sie für Ihre Anwendung das Sharding so wählen, dass die aufwändigste und häufigste Anfrage gezielt einzelne Shards anspricht. Nur so sind hohe Zugriffszahlen zu bewältigen.

■ 14.9 Zusammenfassung

MongoDB zählt zu Recht zu den Stars der NoSQL-Welt. Die mächtige Abfragesprache, Indexe auf Dokumentinhalte, Map/Reduce, Replikation und einfaches Sharding machen die Anwendung von MongoDB attraktiv. Einen Freibrief für ein unüberlegtes Datenschema erhält man dennoch nicht: Auch MongoDB kann die Gesetze der Informatik nicht außer Kraft setzen und setzt deshalb auf bewährte Strategien und Verfahren.

MongoDB ist eine gut einzusetzende Datenbank, wenn die Daten Ihrer Anwendung sich gut in eine JSON-Struktur einbetten lassen. Typische Beispiele dafür sind Blogs, Content-Management-Systeme und Session-Speicher. Allerdings ist MongoDB für Anwendungen, die Transaktionssicherheit quer über Dokumentgrenzen hinweg benötigen, nicht die erste Wahl, da atomare Operationen nur innerhalb eines Dokuments möglich sind.

Bei NoSQL-Datenbanken gilt es daher genau zu prüfen, ob deren Eigenschaften für die eigene Anwendung stimmig sind. Bei der breiten Auswahl an unterschiedlichen Datenbanken sollte es aber leicht fallen, eine passende für Ihre Bedürfnisse auszuwählen. Stöbern Sie also im Internet und unterhalten Sie sich mit Kollegen und Freunden, um Ihren Horizont zu erweitern und auf dem Laufenden zu bleiben. In diesem Sinne: Happy Coding!