Workshop Mongodb

Contents

Installatie van de software

Uit praktische overwegingen gebruiken we de windows versie van mongoDB.
Liefhebbers kunnen de linuxversie downloaden. De hele workshop is dan praktisch 1 op 1 te volgen met putty in plaats van een command shell of Powershell.

Dubbelklik op de aangeleverde mongodb-win32-x86_64-2008plus-ssl-3.4.1-signed.mis. Hij wordt dan geïnstalleerd in C:\Program Files\MongoDB\Server\3.4\bin

Ga in je verkenner naar This PC en open de properties.
Ga naar system->advanced system settings->environment variables->system variables en kies path. Klik op Path en dan Edit. Voeg het pad C:\Program Files\MongoDB\Server\3.0\bin toe aan het bestaande pad en sla dat op.

Maak een folder aan:
c:\data\db

Dit is het standaard pad van MongoDB waarin alle datafiles, indexen, metadata enzovoort wordt opgeslagen. Hoe je daarvan afwijkt komt later aan de orde.

Open een command prompt (ook wel Dos box genoemd) of Powershell. De laatste heeft (op mijn pc tenminste) het voordeel dat commando's in een history bewaard blijven, ook na sluiten van de shell.
Tik in

```
mongod
```

en voilà: de database is geopend. De logging ervan blijft door het beeld lopen. Laat deze box open, want als je hem sluit, stop je de database. Met ^C stop je hem ook.

In een meer professionele omgeving wil je Mongod als een service starten. Hoe dat moet, vind je hier

Je ziet ook meteen in de scherm logging op welke poort hij luistert. Default is dat 27017. Later komen we nog uitgebreider op poortnummers terug.

In principe heb je nu genoeg om vanuit een server te connecten, bv met weblogic. Dat komt hier niet aan de orde, we werken vandaag uitsluitend lokaal met de mongo shell.

Open een nieuwe dos box. Ga naar de directory waar je products.json hebt opgeslagen.
Tik in

`mongoimport --db pcat --collection products < products.json` (let op de dubbele streepjes)

--db is de database naam (die je bij deze dus aanmaakt!)
--collection is de naam van de collection binnen die database die je dus ook bij deze aanmaakt.
Je kan ook files importeren in het formaat csv (comma separated) of tsv (tab separated).

Open nu een shell door een nieuwe command prompt (of Powershell) te openen en het volgende in te tikken:

```
C:\>mongo
```

```
MongoDB shell version v3.4.1
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.1
Server has startup warnings:
2017-02-22T20:58:18.515+0100 I CONTROL  [initandlisten]
2017-02-22T20:58:18.515+0100 I CONTROL  [initandlisten] ** WARNING: Access control
is not enabled for the database.
2017-02-22T20:58:18.515+0100 I CONTROL  [initandlisten] **          Read and write
access to data and configuration is unrestricted.
2017-02-22T20:58:18.515+0100 I CONTROL  [initandlisten]
```

Kijk welke databases je hebt:

```
show dbs
```

Welke database gebruik ik op dit moment?

```
db
```

Kies de zojuist aangemaakt pcat database:

```
use pcat
```

Welke collections zitten hier in:

```
show collections
```

Welke documents (rows, zal ik maar zeggen) zitten hier in?

```
db.products.find()
```

Het enige dat iedere rij verplicht heeft is een veld _id. Dit moet uniek zijn. Je mag het zelf toekennen; als je dat niet doet, doet mongo het voor je.
De id's moeten altijd uniek zijn binnen de collection en zijn om die reden geïndexeerd by default. Controleer dat met het volgende commando:

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

Gebruik altijd dubbele quotes voor keys en voor string values om uit de problemen te blijven. De mongoshell is erg coulant, maar dat kweekt slechte gewoontes.

Select queries.

Net als in SQL kan je de collection queryen. In het algemeen gebruiken we daarvoor

```
db.<collection-name>.find()
```

Tussen de haken () kan nader gespecificeerd worden welke velden in de output getoond worden (SELECT in sql) en welke selectie (WHERE in sql) er op plaats vindt.
Welke producten zijn er van het merk ACME?

```
db.products.find({"brand":"ACME"})
```

De output is niet erg fraai. Er zijn in mongo shell wat mogelijkheden dit op te leuken:

```
db.products.find({"brand":"ACME"}).pretty()
db.products.find({"brand":"ACME"}).toArray()
```

Ook is het mogelijk slechts een aantal keys te laten zien. Selecteer alle documents van brand ACME maar toon alleen de velden 'available' en 'type':

```
db.products.find({"brand":"ACME"},{available:1,type:1})
```

Selecteer alle documents van brand ACME maar toon het veld 'available' NIET:

```
db.products.find({"brand":"ACME"},{available:0})
```

Let op: beide soorten weergave (exclusie en inclusie) kunnen niet gemengd worden.
Let op 2: Het veld _id is altijd zichtbaar tenzij het expliciet wordt uitgesloten met _id:0. Dit mag zelfs gecombineerd met een inclusie en is de enige uitzondering op bovenstaande regel.

```
db.products.find({},{name:1,_id:0})
```

Er is een hele reeks geavanceerde commando's om precies die data te selecteren die je nodig hebt. Dat zijn:
$ne -> Not Equal. Bijvoorbeeld `db.products.find({name:{$ne:"ACME"}})`
$gt -> Greater Then. Bijvoorbeeld `db.products.find({monthly_price:{$gt:40}})`
$gte- > Greater Then or Equal. Bijvoorbeeld `db.products.find({monthly_price:{$gte:40}})`
$lt -> Less Then. Bijvoorbeeld `db.products.find({monthly_price:{$lt:60}})`
$lte -> Less Then or Equal. Bijvoorbeeld `db.products.find({monthly_price:{$lte:60}})`
$or -> OR. Bijvoorbeeld `db.products.find({$or: [{monthly_price:60},{brand:"ACME"}]})`
$in -> IN. Bijvoorbeeld `db.products.find({monthly_price:{$in:[40,90]})`

Ook is een toevoeging mogelijk:
limit() beperkt het aantal documents dat teruggegeven wordt, bv

```
db.products.find().limit(4)
```

sort() sorteert de gegevens oplopend of aflopend, bv

```
db.products.find({},{brand:1,price:1,_id:0}).sort({brand:1,price:-1})
```

Probeer de output te begrijpen!

DML queries.

DML staat voor Data Manipulation Language. Dit zijn statements die de data aanpassen. We kunnen data aan bestaande documents toevoegen, hele documents toevoegen, data veranderen en data verwijderen.
We gaan een document toevoegen aan een collectie:

```
db.products.insert({name:"Pom",lastName:"Bleeksma",gender:"Male"})
```

Ik heb expres niet gekeken of dit wel enigszins bij de rest van de collection past, immers, dat is voor NoSQL niet belangrijk.

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

We zien nu alle names uit alle documents.

Na vakantie in Thailand wil ik mijn gender aangepast zien:

```
db.products.update({name:”Pom”},{gender:”Female”})
```

Aan de output zie je dat het geslaagd is:

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

Even kijken naar het resultaat:

```
db.products.find({name:"Pom"})
```

Wat nu? Niets gevonden?
We hebben het verkeerde update commando gebruikt: het update statement heeft het gehele document vervangen! Dat was niet de bedoeling.
Dat draaien we even terug op een omslachtige manier:

```
db.products.remove({gender:"Female"})
db.products.insert({name:"Pom",lastName:"Bleeksma",gender:"Male"})
db.products.find({name:"Pom"})
```

De update had moeten gebeuren met een zogenaamd $ commands. In dit geval $set:

```
db.products.update({name:”Pom”},{$set: {gender:”Female”}})
db.products.find({name:"Pom"})
```

Als het veld niet bestaat, wordt het toegevoegd:

```
db.products.update({name:"Pom"},{$set: {transport:"Motorcycle"}})
db.products.find({name:"Pom"})
```

Andere vergelijkbare $ commands zijn:
$unset -> Om een veld uit een document te verwijderen.
$inc -> om een numeriek veld te verhogen met een bepaalde waarde
Als het veld niet bestaat wordt het toegevoegd (met de waarde van de verhoging)
$push -> Om een waarde aan een array toe te voegen.

Replica sets

Replica sets zijn meerdere mongod's die dezelfde data delen. Ze kunnen op dezelfde host draaien maar ook op verschillende.
Iedere mongod van de set heet een member.
Replicatie heet ook wel vertical scaling en het is bedoeld als redundantie.
De members bepalen onderling welk member Primary is en welke secundary zijn. Dit gebeurt met meerderheid van stemmen dus moet er altijd een oneven aantal members zijn. Kom je met je infrastructuur uit op een even aantal, dan kan je er een zogenaamde Arbiter bij plaatsen: dat is een lege member, die dus nooit primary kan worden maar die wel een stem heeft bij de bepaling wie er Primary wordt.
We gaan een replica set bouwen van 3 members.

Sluit alle command boxes. We beginnen met een schone lei.

Op een command box en tik in:

```
mkdir \data\z1
mkdir \data\z2
mkdir \data\z3
mongod --smallfiles --oplogSize 50 --port 27001 --dbpath \data\z1 --replSet z
```

Open een tweede command box en tik in

```
mongod --smallfiles --oplogSize 50 --port 27002 --dbpath \data\z2 --replSet z
```

Open een derde command box en tik in:

```
mongod --smallfiles --oplogSize 50 --port 27003 --dbpath \data\z3 --replSet z
```

We hebben nu 3 losse members, maar afgezien van dat ze ieder voor zich weten dat ze member zijn van replica set 'z' zijn ze nog niet met elkaar in contact.

Check dit door in een vierde command box een shell te openen naar member port 27003 en de replica set status op te vragen:

```
mongo --port 27003
rs.status()
```

Je ziet dat er gevraagd wordt de set te initieren met rs.initiate(…)
Dat gaat zo:

```
rs.initiate(
        { _id:'z',
          members:[
            { _id:1, host:'localhost:27001' },
            { _id:2, host:'localhost:27002', "arbiterOnly" : true },
            { _id:3, host:'localhost:27003' }
          ]
        }
)
```

De prompt verandert ook meteen, om aan te geven dat we met een replica set werken. Nu geeft hij de set name aan en de status van het member waar je mee bent verbonden. De status is nog even onbekend dus staat er
Z:OTHER>
Vraag de status opnieuw op en bestudeer de output:

```
rs.status()
```

We zien onder meer dat er 3 members zijn, op poorten 27001, 27002 en 27003.
We zien ook de bijbehorende member id's (1 tm 3) en de status van ieder member:
Id1 is secundary
Id2 is arbiter
Id3 is primary
We zien aan de prompt dat we op de Primary zitten.
Is dat niet het geval en is Id1 op port 27001 de Primary?
Doe dan het volgende:
Tik in:

```
exit
mongo –port 27001
rs.stepDown()
```

Hiermee vertel je het member om geen primary meer te zijn. De meerderheid van stemmen kan nu alleen nog maar member Id3 kiezen als nieuwe primary, immers, member Id2 is Arbiter en kan per definitie niet primary worden.

Laten we eens verbinden aan Id1:

```
exit
mongo --port 27001
```

Aan de prompt zien we nu dat we op een Secundary member zitten.

We blijven op de Secundary en proberen wat data in te voegen:

```
db.foo.insert( { _id : 1 }, { writeConcern : { w : 2 } } )
```

NB: Ik introduceer hier meteen writeConcern: de parameter w:2 geeft aan dat ik de prompt pas terugkrijg als er naar minimaal 2 members is geschreven. Uitleg en meer parameters vind je op https://docs.mongodb.com/manual/reference/write-concern/

Het inserten is niet gelukt en de error zegt duidelijk waarom:

```
WriteResult({ "writeError" : { "code" : 10107, "errmsg" : "not master" } })
```

Master, primary, what's in a name… In elk geval mag je blijkbaar niet inserten vanaf een secundary member.
We connecten weer naar de Primary, id3, en proberen het opnieuw:

```
exit
mongo --port 27003
db.foo.insert( { _id : 1 }, { writeConcern : { w : 2 } } )
```

Gelukt!
Laten we nog 2 rijen toevoegen:

```
db.foo.insert( { _id : 2 }, { writeConcern : { w : 2 } } )
db.foo.insert( { _id : 3 }, { writeConcern : { w : 2 } } )
```

Secundary member lost. Now what?

Eens kijken wat we nou aan die replica sets hebben…
We connecten aan de Secundary, vragen de data op en brengen hem daarna down.

```
exit
mongo –port 27001
db.foo.find()
```

Dat lukt ook al niet! Maar het moet toch mogelijk zijn te queryen op een secundary? Dat kan, zie het volgende:

`rs.slaveOk()` -> Dat wil zeggen dat je toestemming geeft te lezen vanaf een secundary member
`db.foo.find()`

We gaan de replica set breken.
Ga naar de eerste command box, waarin je de server op port 27001 hebt gestart (dit is gelukkig te zien in de titelbalk van die box) en stop die met ^C.
Ga dan weer naar de command box waarin je shell nog draait. Tik in

```
exit
mongo –port 27003
db.foo.find()
rs.status()
```

Bestudeer de output, met name die omtrent Id1. Je ziet dat hij not reachable/healthy is.

Kunnen we nog data toevoegen vanaf de primary? Tik in:

```
db.foo.insert( { _id : 4 } )
```

Let op: geen writeConcern gebruiken, want er is maar 1 member waarnaar geschreven kan worden.

We voegen er nog 2 toe:

```
db.foo.insert( { _id : 5 } )
db.foo.insert( { _id : 6 } )
```

Nu starten we het eerste member weer. Ga naar de betreffende command box en tik in:

```
mongod --smallfiles --oplogSize 50 --port 27001 --dbpath \data\z1 --replSet z
```

Lees de output: daarin zie je dat hij reconnect en recovered.

Laten we connecten aan de herstelde secundary en kijken of alle documents er weer zijn:

```
exit
mongo –port 27001
rs.slaveOk()
db.foo.find()
```

Ja, alles is er weer. Het member heeft zichzelf gesynchroniseerd met de primary. Zonder concrete aanleiding zal hij ook niet zomaar weer primary worden.

All members lost. Now what?

We beginnen even met een schone lei.
Connect aan de Primary en drop de collection:

```
exit
mongo –port 27003
db.foo.drop()
```

We maken weer een collection foo aan met wat data:

```
db.foo.insert( { _id : 1 }, { writeConcern : { w : 2 } } )
db.foo.insert( { _id : 2 }, { writeConcern : { w : 2 } } )
db.foo.insert( { _id : 3 }, { writeConcern : { w : 2 } } )
```

Breng de secondary server down door naar command box 1 te gaan en ^C te geven.

Ga naar de shell en check de status:

```
rs.status()
```

We voegen wat data toe:

```
db.foo.insert( { _id : 4 } )
db.foo.insert( { _id : 5 } )
db.foo.insert( { _id : 6 } )
```

Let op: deze data staat nu dus alleen op member id3, want de secundary is down.

Nu stoppen we de Primary. Ga naar de betreffende command box (de derde als je alles precies gevolgd hebt) en stop hem met ^C.

Nu draait alleen de Arbiter nog. Daarop kan je de status van de replica set testen:

```
exit
mongo –port 27002
rs.status()
```

We brengen nu de Secundary weer op, op port 27001. Dat is je eerste command box. Tik daarin:

```
mongod --smallfiles --oplogSize 50 --port 27001 --dbpath \data\z1 --replSet z
```

Bekijk de output. Van de ooit 3 members zijn er nu 2 up. Die besluiten met meerderheid van stemmen dat port 27003 er niet is en dat port 27001 dan de Primary wordt.

Ga naar de command box met de shell (de vierde box als het goed is) en tik in:

```
exit
mongo --port 27001
rs.status()
```

Is de data die we hadden toegevoegd aanwezig?

```
db.foo.find()
```

Nee dus. De 27001 was down op het moment dat de data werd toegevoegd. Voordat hij weer kon synchroniseren met 27003 ging 27003 down.

Kunnen we nieuwe data toevoegen? Tik in:

```
db.foo.insert( { _id : "last" } )
```

Dat gaat goed.
We brengen de voormalige Primary (27003) weer in de lucht. Ga naar de betreffende command box en tik in:

```
mongod --smallfiles --oplogSize 50 --port 27003 --dbpath \data\z3 --replSet z
```

Kijk naar de output: er staat dat er een rollback plaats vindt. Daar komen we zo op terug.
Nu gaan we onderzoeken welke data er op de (nu secundary) 27003 aanwezig is.
Ga naar de shell en tik in:

```
exit
mongo --port 27003
rs.slaveOk()
db.foo.find()
```

We zien alleen de laatst toegevoegde data. Logisch, maar niet fijn. Is nog wat met die rollback te doen?

Het antwoord is 'Ja'. Op het moment dat de 27003 weer opgebracht werd, zag hij dat hij data had die niet gerepliceerd is, maar ook dat de database alweer een stuk verder in tijd is.
Met de data die hij nog had, namelijk rij 4 tm 6, doet hij een rollback. Dat wil zeggen, hij slaat de data op in een externe file en vergeet ze vervolgens.

Aan ons de uitdaging de data boven water te halen en alsnog toe te voegen aan de collection 'foo'.

Ga naar de vierde command box (die waar de shell draait) en tik in:

```
exit
cd \data\z3\rollback
dir
```

Er staat een .bson file. Dit is de dump van de rollback data.
Om deze data te bekijken gebruiken we bsondump:

```
bsondump test.foo.2017-03-06T13-25-51.0.bson
```
File name varieert natuurlijk!

Deze bson file is hetzelfde formaat als een mongodump oplevert, een utility dat voor backups gebruikt wordt. De tegenhanger van mongodump is mongorestore. Dat gebruiken we nu om de ‘verloren’ data toe te voegen aan de ‘foo’ collection van de database ‘test’.
Tik in:

```
mongorestore --port 27001 --db test --collection foo test.foo.2017-03-06T13-25-51.0.bson
```

Nu controleren of de data inderdaad terug is gezet:

```
mongo --port 27001
db.foo.find()
```

Backup en recovery

Mongo backups worden gemaakt met het commando mongodump. Daarbij kan je zelf kiezen welk replica member je neemt. Aangezien het in principe CPU power kost, kan het aan te raden zijn hier een apart member voor te reserveren.
We gebruiken member Id3 in dit geval.
De database moet wel up zijn.
Tik in een command box (niet in de mongo shell) het commando:

```
mongodump --port 27003
dir
```

Je ziet dat hij een dump directory aanmaakt onder de je current directory, in mijn geval onder C:\data\z3\rollback, en daar de nodige files in zet.

Open nu weer een mongoshell naar het primary member. We gaan de collection 'foo' droppen.
Tik in:

```
db.foo.drop()
db.foo.find()
```

Als het goed is, komen er geen rijen terug.

Verlaat de mongo shell en restore de collection:

```
exit
mongorestore --port 27001 --db test c:\data\z3\rollback\dump\test
mongo --port 27001
db.foo.find()
```

Voilà, alle 7 rijen zijn terug.

Maar hoe zit het met consistentie?
We maken een collection van 1000.001 rows en terwijl die data gegenereerd wordt maken we een mongodump.
We hebben daarvoor een nieuwe lege mongod nodig. De database heet groot, de collection heet veel.

Stop alle mongod's met ^C. Sluit een paar command boxes, je hebt er uiteindelijk 3 nodig.
Maak een directory aan voor de nieuwe db 'groot' en start een mongod die die directory gebruikt:

```
mkdir \data\groot
mongod --smallfiles --oplogSize 50 --dbpath \data\groot
```

Ga naar de tweede command box, ga naar de directory waarin groot.js staat en typ daar in:

```
mongo --shell localhost/groot groot.js
```

Ga naar de derde command box en tik alvast in, **zonder op Enter te drukken**:

```
mongodump --port 27017
```

Ga nu terug naar de box met de mongo shell en tik in:

```
homework.init()
```

Er worden nu 1.000.001 rijen gegenereerd.
Laten we een dump maken terwijl dit aan de gang is.
Ga naar de box waarin je commando-zonder-enter al klaar staat en druk op Enter.

Het moge duidelijk zijn: hij dumpt wat er op dat moment in de collection staat en dat is het dan.

Je kan zelfs ongestraft nog een keer dezelfde dump maken en die overschrijft dan de vorige.

Dit is de reden dat het belangrijk is om replica sets te gebruiken: de kans dat alle members van een replica set de geest geven is bijzonder klein.

Nog een wetenswaardigheid: gooi de zojuist aangemaakte collection (veel) weg mbv het drop commando.

Neem daarna een kijkje in de directory c:\data\groot: alle datafiles en indexfiles staan er gewoon nog. Je zult ze zelf met de hand moeten verwijderen als ze niet meer nodig zijn.

Sharding

Sharding is enigszins vergelijkbaar met partitioning. Shard betekent letterlijk splinter.
Een collection wordt opgedeeld in meerdere stukken, op basis van een shard key.
De verschillende shards draaien op ieder op een eigen mongod.
Een eindgebruiker queried dan ook niet de mongod, maar een mongos. Dit is een server die weet welke data waar staat en acteert als een soort load balancer.

Verder draait er nog 1 (leuk voor thuis) of 3 (productie) of meerdere mongod config servers. Die hebben maar 1 doel: onthouden waar welke data staat.
De mongoserver gebruikt dit om data op te halen.

We gaan een eenvoudig cluster bouwen met 2 shards, 1 config en 1 mongos. Geen replica sets.

Eerst maken we een standalone database.

Maak de directory \data\db leeg.

Start een mongod:

```
mongod
```

Ga naar de directory waar je week6.js hebt opgeslagen en tik in:

```
mongo --shell localhost/week6 week6.js
homework.init()
```

Er worden 1.000.001 documents aangemaakt, dat duurt een tijdje. De voorgang is te volgen in het scherm.

Als het klaar is, check de status dan met

```
db.trades.stats()
```

Nu hebben we een single mongod en die gaan we transformeren in een sharded cluster met 1 member.
Stop de mongod en herstart hem met de –shardsvr optie:

```
mongod --shardsvr
```

Let op: hij start nu weer in de \data\db directory. Wellicht is het handiger om van te voren een directory te gebruiken waarin de shardnaam voor komt, mbv de –dbpath optie.
Voor nu laten we het zo.
Zie ook dat hij default op port 27018 start. Straks zullen we zien dat de mongos server de bekende default port 27017 gebruikt, als we niet zelf een port opgeven.

Nu starten we de config server. Deze gebruikt by default de directory \data\configdb, dus moeten we eerst even aanmaken.

```
mkdir \data\configdb
```

```
mongod –configsvr
```

Als laatste moet de mongos gestart worden. Die moet ook weten aan welke configdb hij moet verbinden dus dat geven we als optie mee:

```
mongos --configdb localhost:27019
```

Verbinden met de database (of de config server) loopt voortaan via deze mongos.
Ga weer naar de directory waarin week6.js staat en tik in:

```
mongo --shell localhost/week6 week6.js
```

De prompt laat al zien dat we aan de mongos verbonden zijn.

We gaan de eerste (en enige..) shard toevoegen aan de configuratie:

```
sh.addShard("localhost:27018")
```

Als dit gelukt is ({ "shardAdded" : "shard0000", "ok" : 1 }) dan vertellen we de config dat de database week6 (waar we mee werken) geshard mag worden:

```
sh.enableSharding("week6")
```

Nu heeft de database demogelijkheid te sharden. Sharden gebeurt per collection. Kijk welke collections we hebben en kijk ook of de data er nog steeds in zit die we er voor het hele shard gebeuren in hebben gestopt. De collection heet trades:

```
db.trades.find().pretty()
db.trades.count()
db.trades.stats()
```

We zien een aantal belangrijke zaken.

"sharded" : false,
"primary" : "shard0000",
"ns" : "week6.trades",
"count" : 1000001,
"size" : 496000240, …..

De collection is niet sharded. Dat klopt, we hebben nog geen shard key opgegeven. Verder is te zien dat er veel documents in zitten en dat hij behoorlijk groot is. Als hij erg klein was, zou er niet geshard worden, ook al is hij ervoor enabled. Sharding gebeurt op basis van de grootte van de database namelijk.

We gaan sharden op de compound shard key ticker + time. Om dat te kunnen doen, moet er een index op die key bestaan. Die maken we dus eerst.

```
db.trades.createIndex( { ticker : 1, time : 1 } )
sh.shardCollection("week6.trades",{ticker:1,time:1},false)
```

De laatste parameter, false, geeft aan dat de index niet unique is.

In de config database kunnen we de gevolgen onderzoeken:

```
use config
db.chunks.find({}, {min:1,max:1,shard:1,_id:0,ns:1}).pretty()
```

Zoals te zien zitten alle chunks nog in "shard0000".

We gaan een shard toevoegen en kijken of de chunks zich netjes verdelen over de shards. Dit gebeurt puur op basis van het aantal chunks per shard.

Iedere shard heeft zijn eigen storage location. De eeste gebruikt al \data\db. Voor de tweede maken we een eigen directory aan. Start een nieuwe command box en tik in:

```
mongod --shardsvr –dbpath \data\shard2 –port 27020
```

Nu moet de config server deze extra shard opnemen in het cluster:

Ga naar de directory waarin week6.js staat en tik in:

```
mongo --shell localhost/week6 week6.js
use config
sh.addShard("localhost:27020")
```

De chunks gaan zich nu verspreiden over de shards. Dit kan je volgen / controleren met

```
use config
db.chunks.find( { ns:"week6.trades" }, {min:1,max:1,shard:1,_id:0} ).sort({min:1})
db.chunks.aggregate( [
 { $match : { ns : "week6.trades" } } ,
 { $group : { _id : "$shard", n : { $sum : 1 } } }
] )
```

Als het goed is, heb je nu 15 chunks die netjes verdeeld zijn over de beide shards.