

Databanken 3

NoSQL Databases

Academiejaar 2018 - 2019

Jan Van Overveldt

Tom De Reys



In dit tweede hoofdstuk over NoSql gaan we dieper in op enkele implementaties van NoSql databases. We geven een overzicht van de meest voorkomende vormen en duiken dieper in de meest populaire NoSql databank, nl. MongoD

Inhoud

1 Key - Value Stores



2 Column Family Store

3 Graph databases

4 Document Stores

5 Search Engines

6 MongoDB

In dit hoofdstuk gaan we een 5-tal vormen van NoSql databases bespreken. Er zijn nog meer soorten, maar dit zijn de 5 meest gangbare:

- Key-Value Stores
- Column Family Stores
- Graph Databases
- Document Databases
- Search Engines (variant van document databases)

Tot slot gaan we meer in detail bekijken hoe MongoDB - de meest gebruikte NoSql databank en zelf een Document Store - werkt en enkele concepten over MongoDB uitwerken

Roundup....

NoSQL is gewoon een *buzzword* om aan te geven dat de wereld niet stilstaat en veranderende omstandigheden ervoor zorgen dat Relationale Databases niet meer het rijk voor zich alleen hebben.

(maar nog steeds wel veruit dominant zijn)

Echter vandaag de dag zijn er al heel wat andere types databases komen om te hoek kijken...



De NoSQL technieken bestaan eigenlijk al redelijk lang, maar van 2010 zien we de NoSQL databanken ook echt doorbreken. De redenen hiervoor hebben we eigenlijk in het eerste hoofdstuk al aangehaald. Gezien de massale groei aan informatie die we tot onzer beschikking hebben in combinatie met de diversiteit in formaten, volstaan de huidige generatie relationele databases niet langer. NoSql databases daarentegen voldoen meer aan de huidige noden en daarom ziet het er naar uit dat ze deze keer echt door gaan breken zijn.

Voor de duidelijkheid: NoSQL staat niet voor "geen SQL", maar voor Not Only SQL databanken.

NoSQL



Onder de noemer [No-SQL databases](#) vallen een heleboel verschillende database varianten, die enkel het feit gemeenschappelijk hebben dat ze:

- **Niet-relationeel** zijn van structuur
 - = niet of minder gebaseerd op het concept van tabellen, rijen, kolommen en joins
- Vaak **schemaless** zijn
 - Database legt geen constraints op aan structuur en de relaties van de data
(in tegenstelling tot relationele DB's, waar data exact moet voldoen aan het database schema)
- Meestal eenvoudig te **clusteren** zijn
 - Eenvoudig om performantie en beschikbaarheid te verhogen

NoSql databases is eigenlijk de verzamelnaam van een héél aantal soorten databases die anders gestructureerd zijn dan de klassieke relationele databases. Wij bespreken er in deze presentatie de meest gangbare vormen en zoemen tot slot dieper in op MongoDb.

Wat kenmerkt deze No-Sql (= niet-relationele) databanken nu?

- Hun structuur is verschillend van de klassieke concepten zoals tabellen, rijen, ... Meest voorkomende concepten zijn Documents, Classes, Key-Value pairs, ...
- Daarnaast werken zij vaak niet volgens een vast schema, de data hoeft dus niet te voldoen aan een opgelegde structuur, constraints, Men is vrijer om de data te modeleren en persisteren.
- In tegenstelling tot de RDBMS zijn zulke systemen al van begin af aan ontworpen om te werken met clustering en ondersteunen ze meestal zowel replicatie als sharding om de performantie en availability te verbeteren.

NoSql - voorbeelden



Zeer **grote variëteit** aan (vnl. open source) producten.
De markt is nog niet uitgekristalliseerd...

Er zijn nog redelijk veel spelers op de markt wat betreft NoSql databases momenteel. Bovenstaande slide toont de meest gekende NoSql DBMS, maar er zijn nog redelijk veel spelers op de markt. Er zijn wel trends, zoals MongoDb die de meeste populaire NoSql is en blijft. Daarnaast zijn ook Cassandra en Redis populair en zijn specifieke NoSql databanken voor search engines zoals ElasticSearch sterk aan het opkomen momenteel.

Bron: <http://db-engines.com/en/ranking>

Meest voorkomende types NoSQL databanken

Key-value stores

Database bestaat uit key-value maps waarin gelijk welke entiteit (value) kan gekoppeld zijn aan een key.

Column family stores

Value is een groep van 'kolommen' (= key-value pairs).

Men spreekt ook wel van **Big table**

Graph databases

Bedoeld voor probleemdomen waar entiteiten zeer veel relaties hebben naar andere entiteiten (bv. sociale relaties, geografische data, ...)

Document stores

Value is een document met een zekere structuur (bv. JSON, XML,...)

Search Engines

Databases die voornamelijk gericht zijn op het aansturen van search engines. Meestal is dit een variant van een document store maar kan ook op basis van Graphs zijn

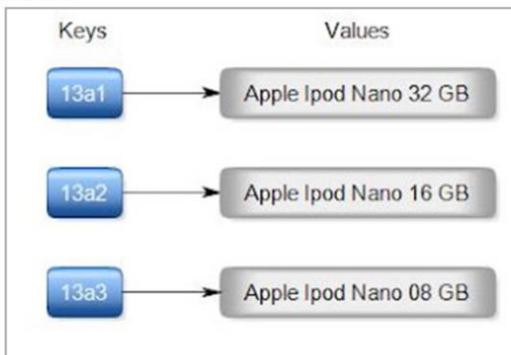
We kunnen NoSQL databases opdelen in 4 verschillende types, afhankelijk van wat voor soort van datamodel ze gebruiken. Elk van deze types leggen we uit in de volgende slides, met telkens de belangrijkste databases op de markt van dat type database.

Eerst zullen we deze vier types kort theoretisch overlopen waarbij we op een aantal technologieën dieper ingaan.

Er zijn ook meer en meer "search engine" databases in opkomst, maar die zijn meestal gebaseerd op één van deze 4 types. De meest gekende is Elasticsearch. Elasticsearch bijvoorbeeld is gebaseerd op het concept van een document store.

Key-Value stores

Key-value stores



- Zeer eenvoudig datamodel
 - "keyspaces": koppel een **value** aan een unieke **key**.
 - Value is een binaire blok data waar om het even wat kan ingestopt worden (schema-less)
- De meeste key-value stores hebben extreem goede voorzieningen op vlak van **clustering** (availability en performance)

Ten eerste heb je de Key-Value stores. Hierbij ga je entiteiten koppelen aan een bepaalde key zodanig dat je deze snel kan opzoeken op basis van deze key. Eén entry is een store noemen ze ook wel eens een "keyspace".

Eigenlijk je dit type NoSQL database vergelijken met een Dictionary (uit .NET) of Map (uit Java) waarin je ook key-value pairs kan bijhouden. Er is echter één groot verschil, namelijk dat bij een Key-Value store de value niet hetzelfde hoeft te zijn voor elke key. Je kan dus gelijk welke entiteit stockeren in zulke een key-value store, zolang de key maar uniek is. De ene entry kan de value een Order zijn, de volgende een Customer, ...

Dit type NoSQL database blinkt uit in eenvoud en werkt gigantisch snel als je enkel op basis van keys iets moet opzoeken. Opzoeken binnen de values is daarentegen minder eenvoudig of zelfs onmogelijk.

Key-value stores



EXAMPLE

```
IRiakClient client = RiakFactory.pbcClient();  
  
Bucket myBucket = client.fetchBucket("test").execute();  
  
int val1 = 1;  
myBucket.store("one", val1).execute();  
  
Integer fetched1  
= myBucket.fetch("one", Integer.class).execute();
```

Andere



...

Bovenstaande code geeft weer hoe "eenvoudig" het is om zulk een key-value store aan te spreken. In deze case gaan we vanuit Java een "Riak" Key-Value store databank aanspreken.

Eerst moeten we een Client object aanspreken om een connectie te kunnen gaan leggen met een Riak-databank. Vervolgens spreken we uit die Riak databank een bepaalde Key-Value store aan, wat bij Riak een 'Bucket' noemt. In die 'Bucket' kunnen we dus onze key-value pairs gaan bewaren.

Je kan verschillende Buckets definiëren, in dit voorbeeld hebben we onze Bucket 'test' genoemd.

De volgende lijnen code gaan dat in de geselecteerde Bucket het getal 1 wegschrijven als Value met de key "one" als bijhorende key.

Met de laatste lijn code gaan we dan uit onze Bucket de waarde ophalen die we net weggeschreven hebben. Om die waarde op te halen, hebben we opnieuw de Key nodig waaraan deze waarde gekoppeld is.

Meer informatie over Riak kan je vinden op <http://basho.com/products/riak-kv/>

Key-value stores

Wanneer gebruiken?

- Je values hebben géén structuur
(bv. *binary blob image data*)



OF

- Je values hebben structuur maar die moeten niet doorzocht of geaggregeerd kunnen worden
EN
Je values moeten niet naar elkaar verwijzen
(bv. *profile settings, winkelkarretje, session informatie, ...*)
- Wordt ook vaak als in-memory databank gebruikt

Key-Value stores zijn zeker niet voor alle soorten toepassingen geschikt, maar zijn wel de beste keuze als aan één van de volgende voorwaarde voldaan wordt:

- De values die je gaat stockeren hebben geen (of een beperkte) structuur. Typisch voorbeeld hiervan zijn BLOB (images, geserialiseerde files, ...)
- De values hebben wel een structuur, maar je moet daar niet op zoeken en/of aggregatie-operaties op uitvoeren. Je kan immers alleen maar zoeken op key EN er is géén verwijzing van de ene value naar de andere value. Dus elke value staat geïsoleerd van de andere.

Een typisch voorbeeld zou kunnen zijn dat je een Key-Value store gaat gebruiken om je profile settings in te bewaren. Als key neem je dan bijvoorbeeld de gebruikersnaam en als value de bijhorende profiel settings. Zulke settings zijn uniek per gebruiker en ga je dus alleen maar ook ophalen op basis van de gebruikersnaam. Profielsettings zijn meestal ook niet gelinkt met andere entiteiten en staan dus ook los van alle anderen.

Ook winkelwagentjes bij online shops zijn een goed voorbeeld. Als je inlogt bij bv. Amazon krijg je zonder vertraging de inhoud van je basket beschikbaar die je in een eerdere sessie hebt aangemaakt. Amazon moet er miljoenen van opslaan maar moet die wel snel beschikbaar kunnen stellen bij het inloggen. Durability (ACID) is minder belangrijk. Als je inlogt en blijkt dat je winkelkarretje van de vorige keer niet meteen beschikbaar is, ga je daar ook geen nachten van wakker liggen, waarschijnlijk heb je het zelfs niet opgemerkt. De volgende keer als je inlogt is het daar immers terug.

Key-value stores worden in (web) applicaties ook gebruikt om tijdelijk gegevens te gaan bewaren over http-calls heen. In een web applicatie wordt elke http-call apart afgehandeld

en zijn de gegevens berekend en ev. bijgehouden in het geheugen van de web-server bij een volgende http-call al weer weg. Stel dat men toch gegevens wil bijhouden over calls heen, kan me ervoor opteren om die gegevens in een in-memory key-value store te bewaren. De SessionState in web applicaties werkt eigenlijk zo, al is die sessionstate gekoppeld aan één specifieke webserver en belast die het geheugen van de web server. Om dat te vermijden, of toe te staan dat elke http-request via een load balancer op een andere web server afgehandeld kan worden, kan je je gegevens tijdelijk gaan stockeren in een in-memory-databank.

Het grote voordeel van een in-memory databank is dat deze bijzonder snel zijn aangezien de meest vertragende factor, namelijk disk I/O, hier vermeden wordt.

Column Family Store

Column family store

row key	columns ...			
	name	email	address	state
jbellis	jonathan	jb@ds.com	123 main	TX
	daria	dh@ds.com	45 2 nd St.	CA
egilmore	name	email		
	eric	eg@ds.com		

Zoals key - value maar de **values** in een key-space bestaan opnieuw uit een **lijst van key - value paren**

Een column family lijkt op een table waarbij echter niet elk rij dezelfde kolommen hoeft te hebben

Het volgende type van NoSql databases is een "Column family store". Deze toont v  l gelijkenissen met een Key-Value store. Ook hier wordt er gebruik gemaakt van een key, maar voor de bijhorende value wordt deze keer meer dan alleen maar een entiteit bijgehouden.

Elke value is op zich ook weer een lijst van key-value pairs. Key zijn de 'kolom-namen', value zijn dan de bijhorende 'values' van de key.

Je bent in het algemeen vrij te kiezen welke kolommen je voor één bepaalde rij opslaat. Per rij mogen die dus verschillen want er wordt immers geen schema opgelegd.

Je kan dit vergelijken met een tabel, waarvan de primary key een 'row key' bevat, en dat elke rij zijn eigen kolommen en corresponderende waarden heeft. Als je in zulke stores een 'entiteit' uit de store gaat ophalen op basis van de (row) key, ga je dus terug een key-value pair lijst krijgen. Het voorbeeld in de slides zal er in de database waarschijnlijk ongeveer zo uitzien

Key	Column Key	Column Value
jbellis	name	jonathan
jbellis	email	jb@ds.com
lbellis	adress	123 main
lbellis	state	TX
egilmore	name	eric
egilmore	email	eg@ds.com

dhutch name daria

....

Column family store

EXAMPLE



Masterless architecture

Lineair scalability

Combinatie van **sharding** en **replication**



Andere



De meest populaire Column Family Store is Cassandra.

Typisch aan Cassandra is dat deze ontworpen is met als bedoeling dat deze high-availability ondersteunt in een combinatie van Sharding en Replication. Atypisch voor replicatie, is dat er hier géén master-database(s) voor nodig zijn, maar elke node als 'master' zal optreden. De scalability van Cassandra is ook krachtig en verloopt linear. Dubbel zoveel nodes opzetten betekent ook dubbele snelheid, wat vrij uniek is voor een DBMS. Meestal komt er bij upscaling in verhouding minder performantie bij omdat extra nodes inbouwen voor wel wat overhead zorgt (zoals replicatie opzetten...), maar bij Cassandra is dat niet het geval.

Eén van de bekendste applicaties die Cassandra gebruikt, is NetFlix. Zij gebruiken deze DMBS als hun belangrijkste data storage engine.

NetFlix gebruikt trouwens ook nog een ander type NoSql database voor hun suggesties aan te kunnen bieden, namelijk Neo4J. Neo4J is een graph database die we verderop in de cursus ook nog gaan behandelen.

Column family store

Performance



Inserts/Updates

Operational reads on key

Operational reads on other columns

Joins

Analytical reads

Column family stores zijn ontzettend goed in het snel wegschrijven van grote datahoeveelheden en net zoals bij een key value store is het uitlezen van gegevens gekoppeld aan een bepaalde sleutel heel performant.

Veel van de column family stores laten het echter niet toe om te filteren op andere velden dan de indexen (vb. Hbase). Ook maakt het joinen van tabellen meestal geen onderdeel uit van de functionaliteit van dergelijke databases (vb Hbase, Cassandra, DynamoDB).

Deze beperkingen kan je wel omzeilen door gebruik te maken van bepaalde trucjes. Om toch te kunnen zoeken op een niet indexveld kan je ervoor kiezen eerst snel een nieuwe tabel aan te maken die het veld waarop gezocht moet worden mee in de index te plaatsen. Het lijkt misschien absurd maar het kan toch een goede optie zijn.

Het joinen van tabellen kan je dan weer overlaten aan een tool die de data uit de databank leest. Denk hierbij aan tools zoals Pentaho. Beter nog zijn de BigData applicaties die ook via clustering werken en hierdoor de data van de column family store ook zeer snel verwerkt krijgen.

Al dit bovenstaande heeft dus ook tot gevolg dat dergelijke systemen meestal niet gemaakt zijn voor analytische queries. Dergelijke databanken zijn dan meestal ook meer aan de operationele zijde gepositioneerd.

Die slechte ondersteuning voor analytische queries is dan ook één van de belangrijkste reden van het ontstaan van Apache Kudu. Door een verzameling aan slimme technieken slaagt deze database er in zowel op vlak van insert/update verwerking en analytische queries schitterende cijfers neer te leggen. Al moet wel gezegd worden dat de technologie van deze beide functionaliteiten niet de snelste is omwille van tradeoffs die ze moesten maken om beide functionaliteiten goed beschikbaar te krijgen.

Column family store

Wanneer gebruiken?

Je hebt een probleem domein dat zich leent tot relationele* opslag (*veel gelijkwaardige entiteiten met onderlinge relaties*) maar je hebt:

- Veel sparse columns
= kolommen die maar voor enkele rijen een waarde hebben
- Nood aan een minder strak schema
- out-of-the box scalability en availability
- Geen analytische queries
- Je wil gigantisch snel massa's operationele data wegschrijven en ophalen.



*parent-child relaties waar een document database vaak meer geschikt is (zie verder)

Ook column family store databases hebben hun specifiek nut. Ze zijn zeker niet altijd de beste keuze, maar er zijn zeker grote voordelen aan verbonden. Bovendien toont deze store ook gelijkenissen met deze van een relationeel model.

Als de meeste van je entiteiten een gelijkaardige structuur hebben, of je hebt entiteiten met vél sparse columns (=columns die vaak leeg zijn) is dit type store goed geschikt. Je hebt (meestal) een minder strak schema dan het klassieke relationeel model, maar je kan (meestal) toch zoeken op zowel key maar ook op de columns.

Dit type store is ook zeer eenvoudig scalable met de nodige voorziening van availability, net zoals bij key-value stores.

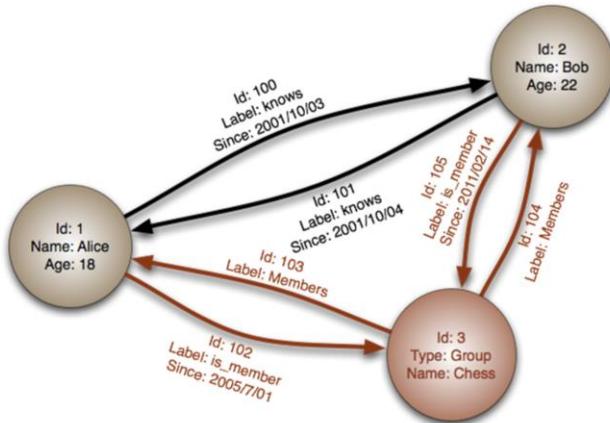
Als je entiteiten onderling relaties hebben, kan je deze ook wel stockeren, maar zeker bij parent-child relaties kijk je beter naar document databases (zie verder).

Tenslotte is het belangrijk te weten dat dit type databanken vaak niet goed geschikt is voor analytische queries, maar dit heeft ook veel te maken met hoe de technologische implementatie precies in elkaar zit.

Voor writes en gerichte reads is dit databank echter steeds zeer geschikt.

Graph databases

Graph database



Het datamodel van een database bestaat uit 'geconnecteerde' nodes. Kent expliciet het concept 'relatie'. **Relaties kunnen eigenschappen en een richting hebben.**

Dit type DBMS legt meer nadruk op de relaties tussen entiteiten want in bepaalde situaties zijn de relaties tussen entiteiten minstens even belangrijk dan de entiteiten zelf. Een graph database legt de focus op die relaties en maakt dan ook gebruik van het concept "relatie". Tussen nodes kunnen relaties in beide richtingen gelegd worden en deze relaties krijgen op zich ook eigenschappen.

Als je data gaat querieën in een graph database, ga je als het ware de relaties af en filter je daarop. De eigenschappen van een node ga je enkel maar tonen in je resultaten, maar gebruik je normaal niet om op te filteren.

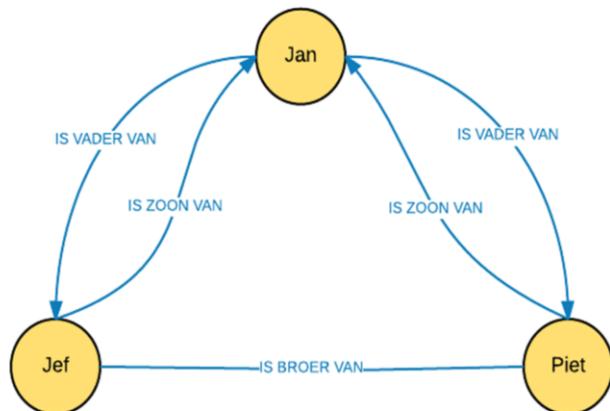
Je kan de vergelijking maken met rollenames en navigability van de associaties in een domein model. De label van de relatie is de rolename van de associatie, je hebt de navigability van de associatie en dan kan je nog voor een specifieke 'instantie' van die associatie extra gegevens bijhouden (zoals het attribuut Since in ons voorbeeld).

Het belang van dit type databases wordt alsmaar groter. Veel sociale media gebruiken een Graph database achterliggend om de relaties tussen personen te modeleren. Op basis van al die relaties, worden er dan ook suggesties gedaan voor relaties te leggen met andere mensen die mogelijk interessant zijn voor jou. LinkedIn & Facebook zijn daar twee bekende voorbeelden van.

Graph database - Modeleren

- Node

- Feiten
- Eigenlijke inhoud of waarde die je wilt verkrijgen



- Relatie (Edge)

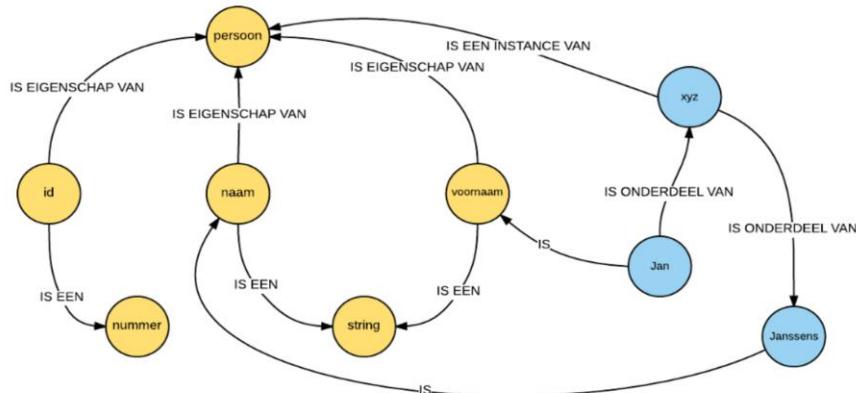
- Link tussen nodes
- Wordt benoemd
- Kan attributen bevatten
- Kunnen uni- of bi-directioneel zijn
- Gebruikt voor filters in queries

Je gaat je modeleren opbouwen volgens nodes en associaties. Op zich lijkt dit niet spectaculair anders dan de klassieke werkwijze maar dat is het wel. Je gaat in dit geval ook héél hard focussen op de associaties tussen je nodes en werkt hier ook opnieuw vanuit de "wat moet ik gaan bevrager/berekenen" met deze gegevens ipv wat is naar disk space en redundantie de beste structuur?

Nodes kan je vergelijken met de feiten of de waarde(n) die je als resultaat van queries wil gaan verkrijgen. Je gaat daar zo weinig mogelijk gegevens in bewaren. Daarnaast ga je tussen de verschillende nodes linken (associaties) leggen en daar ga je wel gegevens over bijhouden. Eerst en vooral een naam, maar je kan ook extra attributen toevoegen. Als je zulk een graph database ontwerpt, moet je eigenlijk altijd in je achterhoofd houden dat je associaties tussen je nodes ervoor zorgt dat je nodes kan filteren op bepaalde condities.

Graph database - Modeleren - Triples

- Subject - Predicate - Object
 - Bv.: Jan is een voornaam
 - Subject en Object zijn je nodes en bevatten je data
 - Predicate is de relatie daartussen en beschrijft dus de band tussen een Subject en een Object



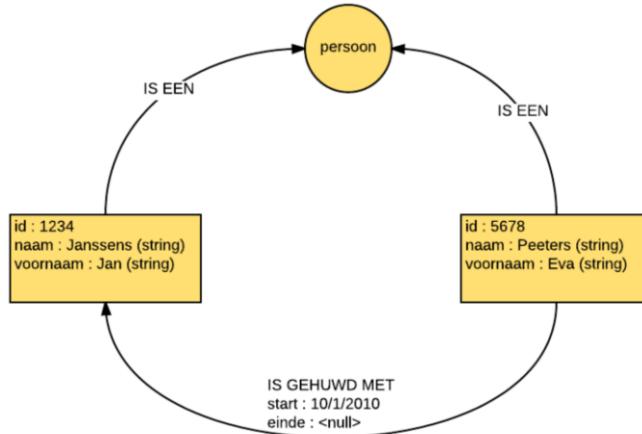
Er zijn twee vaak gebruikte datastructuren bij Graph Databases. Een eerste vorm zijn de Triples.

Bij deze structuur gaan we eerst de opbouw van het datamodel (subject) in nodes opdelen (zie gele nodes), en daarnaast gaan we de eigenlijke data (objects) in nodes bijhouden (zie blauwe nodes). Elke data (object) gaan we dan linken met zijn bijkomende 'subject' zodanig dat we weten wat de betekenis van deze data is. Zowel de nodes alsook de relaties zijn atomair, bevatten dus maar één gegeven.

Je krijgt door het toepassen van deze modeleringstechniek een redelijk gestructureerd model, maar wel een enorm groot aantal nodes. Ook het opstellen van deze structuur vraagt veel werk en inzicht, waardoor er ook een enorme hoge instapdrempel is. Daarom wordt deze techniek enkel maar gebruikt als je daaruit zeker je voordeel kan halen, zoals bij semantic web en linguïstische contexten om bv. zinnen te ontleden. Ook bij (academische) research projecten vind je deze modelering vaak terug. Meer commerciële (mainstream) applicaties zullen een andere modeleringstechniek gebruiken.

Graph database - Modeleren - Property Graphs

- Nodes blijven nodes
- Relaties gaan ook attributen bevatten
 - Query's (where) over nodes, gevraagde info in de attributen



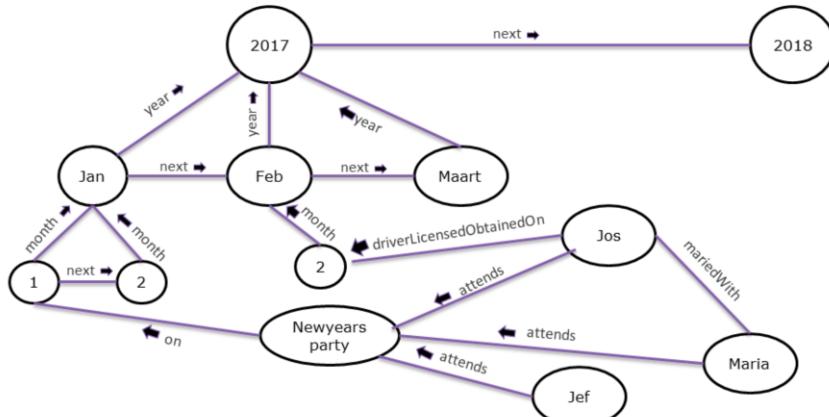
De tweede vorm van modeleren zijn Property Graphs.

In deze vorm gaan we in nodes en/of relaties attributen geven i.p.v. per attribuut een afzonderlijke node. Zo recuceer je het aantal nodes dat je nodig hebt enorm en wordt je datamodel overzichtelijker. Je blijft wel relaties leggen tussen je nodes maar aan die kan je attributen toewijzen i.p.v. enkel een naam.

Hou bij deze vorm wel rekening dat je data waarop je gaat filteren in je queries, zoveel mogelijk in de relaties bughoudt. Graph databases zijn geoptimaliseerd in het zoeken van de relaties tussen de nodes en kan daardoor veel tijd winnen. Bij queries ga je vanuit een node de relaties die daaruit vertrekken bekijken en daarop filteren, zonder dat je de node naar waar deze relatie verwijst gaat uitlezen. Als je te veel gaat filteren op de attribuut-waarden van een node, ga je dit grote voordeel niet kunnen gebruiken.

Graph database - Modeleren - Datum

- Datums worden vaak per onderdeel in nodes gemodelleerd
 - Elke jaartal = Node
 - Jan/Feb/Maart/April/.... = allemaal nodes
 - Elke dag van een maand => allemaal nodes....



Een - correct en courant gebruikt voorbeeld - is hoe dat datums worden gemodelleerd. Een date of date time datatype bestaat zelfs (nog) niet in sommige Graph Databases. In plaats daarvan modelleer je elk onderdeel van de datum (bv. dag - maand - jaar) in een aparte node en leg je relaties onderling zodanig dat je weet welke dag-node gekoppeld is aan welke maand-node, een maand-node is dan weer gekoppeld aan een jaar-node.

Als je dan op specifieke dagen, maanden en of jaren moet gaan zoeken, kan dit via deze node-structuur veel efficiënter dan dat je een datetime als attribuut binnen een node gaat bijhouden. De database engine gaat immers op basis van de relaties direct de juiste node hebben i.p.v. anders elke node apart te gaan moeten benaderen en daar het date-time attribute te gaan uitlezen en vergelijken.

Bijvoorbeeld: zoek alle personen die hun rijbewijs gehaald hebben op 2 februari 2017. Eerst ga je naar juiste datum-node en van daaruit zoek je in alle binnenkomende relaties naar deze die als naam "driverLicensedObtainedOn" hebben. Zo krijg je alle personen terug die je zoekt. Met een ander type databank zou je alle personen moeten gaan doorzoeken om dan te bekijken of de datum 'driverLicenseObtainedOn' overeenkomt met de opgegeven datum. Als je dan een héél aantal personen hebt, moet deze controle ook héél vaak uitgevoerd worden en gaat dit het hele proces enorm vertragen.

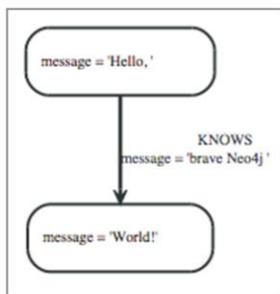
Een ander voorbeeld kan zijn dat je alle events van vanaf 1 januari 2017 wil zoeken. In plaats van alle events één voor één af te gaan, ga je hier naar de dagnode '1' die hangt van de maandnode 'Januari' die op zicht hangt achter het jaar '2017', ga je van die dagnode alle events die erachter hangen ophalen en als je die hebt, ga je via de relatie 'next' naar de volgende dagnode.

Op deze slide kan je een voorbeeld zien van een eenvoudig "data model" zoals men het zou kunnen modeleren in een graph database.

Op de website van Neo4J vind je trouwens zéér interessante artikels en e-books over hoe je kan modeleren, connecteren en queries uitvoeren met graph databases.

Graph database

EXAMPLE



```
firstNode = graphDb.createNode();
firstNode.setProperty( "message", "Hello, " );
secondNode = graphDb.createNode();
secondNode.setProperty( "message", "World!" );

relationship = firstNode.createRelationshipTo( secondNode, RelTypes.KNOWS );
relationship.setProperty( "message", "brave Neo4j " );
```

De belangrijkste speler uit de Graph databases is Neo4J.

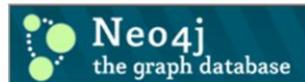
De naam "Neo" uit "Neo4J" komt voor vanuit de film 'The Matrix'. Als je verder met Neo4J werkt zal je merken dat er ook andere benamingen uit deze film afkomstig zijn, zoals bv. de querytaal "Cypher".

Neo4J is - net zoals de meeste NoSql databanken - schemaless maar je kan wel - voornamelijk om performantiereedenen - schema-informatie toevoegen aan je nodes en relaties.

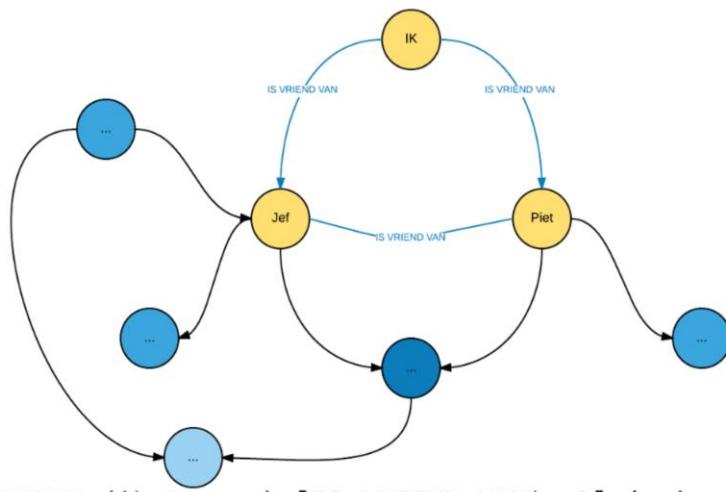
Het voorbeeld uit de slide bevat twee nodes (messages "Hello" en "World") die verbonden worden met een relatie van het type "KNOWS". Op zulke relaties kan je ook een "message" property instellen waarin je meer informatie over die relatie in kan bijhouden.

Neo4J heeft ook ondersteuning voor sharding.

Graph database



EXAMPLE



```
MATCH (ik:Person)-[IS VRIEND VAN*..3]-(vrienden:Person)
WHERE ik.name = 'Vermeulen' AND ik.voornaam = 'Jos'
RETURN ik, vrienden
```

Bovenstaande query is geschreven in Cypher (<https://neo4j.com/docs/cypher-refcard/current>), de query-taal om Neo4J databanken te doorzoeken.

Deze query zal de vrienden van mijn vrienden 2 niveaus diep teruggeven, en dit vrij efficiënt en dus ook performant. Ook al hebben we bv. 2 miljard gebruikers, we hoeven die absoluut niet allemaal te doorzoeken. We starten vanuit mijn eigen node ('Ik') en gaan dan de relaties van het type 'Is Vriend Van' af. Of er nu 10, 100, ... of 2 miljard nodes zijn, dat verandert niets aan deze werkwijze. Je hebt dus weinig last van het groeiende volume van een databank voor je zoekoperaties, mits natuurlijk je je modeleren correct hebt doorgevoerd.

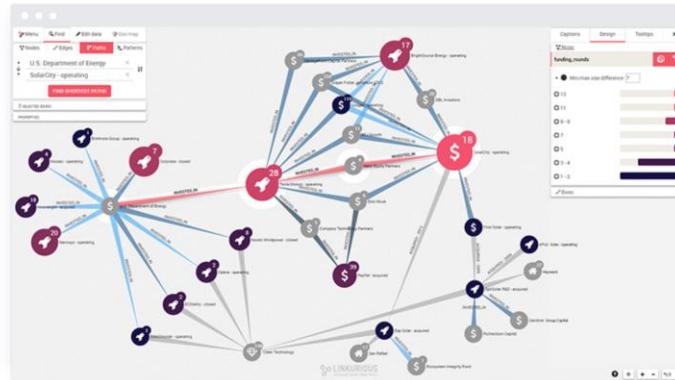
In de praktijk wordt Neo4J onder andere gebruikt door NetFlix. Zij gebruiken deze graph database om gebruikers suggesties te kunnen aanbieden voor series en/of films waarin zij mogelijk geïnteresseerd zijn op basis van hetgeen de gebruikers daarvoor al gezien hebben. Ook tijdens het onderzoek van een groep internationale journalisten naar grootschalige belastingontduiking werd Neo4J gebruikt om die structuren te ontleden. Bij het grote publiek werden deze resultaten bekend gemaakt onder de naam "Panama Papers" (<http://www.zdnet.com/article/panama-papers/>).

Op de website van Neo4J vind je trouwens subsets van datasets waarmee je aan de slag kan. Onder andere een deel van de data gebruikt om de Panama Papers bloot te leggen, kan je daar downloaden.

Hierop bestaan dan tools om de graph database te visualiseren



FIND HIDDEN INSIGHTS IN YOUR GRAPH DATA



24

Voor de panama papers hadden ze 11,5 miljoen bestanden, deze data importeerden ze in een Neo4J graph database. Dan gebruikten ze linkurious om een neo4J database te visualiseren, en hierop werkten ze dan met 400 journalisten die in deze data konden gaan zoeken. Zo konden ze interessante links zien tussen de echte personen en de offshore bedrijven.

Graph database



Wanneer gebruiken?

- Je hebt entiteiten met véél relaties
- Je wil real-time inzicht krijgen in de data
- De relaties zijn het belangrijkste om bij te houden en daarop te gaan zoeken
- Je wilt de verbanden tussen gegevens uitzoeken en analyseren, niet zo zeer de gegevens zelf
- Voorbeelden:



Social Computing



Business Intelligence



Identity Management



Master Data Management



Real time Recommendations



Fraud Detection

Bij Graph databases draait het allemaal rond relaties en verbanden tussen gegevens bijhouden en/of zoeken. Je kan bijna zeggen dat de data van een node op zich van ondergeschikt belang is. Dus als je sterk moet focussen op de relaties (verbanden) tussen entiteiten, is een graph database veruit het krachtigste.

Enkele voorbeelden daarvan zijn natuurlijk de sociale netwerk sites zoals Facebook en LinkedIn. Facebook heeft trouwens hun eigen Graph database "Giraph" ontworpen, die je ook kan vinden via Apache (<http://giraph.apache.org/>). Voor al hun onderzoeken naar relaties tussen personen maken zij gebruik van "Giraph".

Probeer je eens in te denken hoe je met een klassieke genormaliseerde datastructuur de volgende query zou moeten schrijven voor bijvoorbeeld Facebook: zoek tussen 2 miljard gebruikers potentiële vrienden van jezelf door te zoeken wie er niet in je vriendenlijst zit maar wel door minstens 2 andere vrienden (twee niveau's diep, dus ook van vrienden van vrienden) gekend zijn.... Wetende dat er redelijk wat many-to-many tabellen in je datamodel zitten,

Ook in de document stores zou dit niet gaan, daar deze niet geoptimaliseerd zijn voor relaties tussen documenten te gaan bijhouden, laat staan doorzoeken.

Graph databases vind je in diverse domeinen terug. Het meest voor het hand liggende zijn de sociale netwerksites, maar ook meer en meer in Business Intelligence (BI) om analyses op verbanden tussen data beter en sneller te kunnen achterhalen. Identity Management is een minder voor de hand liggend domein waarin Graph Databases voorkomen, maar onderschat deze tak niet. Er zijn hoe langer hoe meer toepassingen waarbij de rechten van

een persoon dynamisch aangepast moeten worden naargelang bepaalde functionaliteiten (bv. klanten die voor een bepaald bedrag kopen, mogen sneller andere deals zien dan klanten onder die drempel, bv. Vente-Exclusive), ... Graph databases helpen dan om zulk een dynamisch model veel eenvoudiger op te stellen.

Master Data Management is het domein waarin ondernemingen (referentie)data van diverse systemen verzamelen, beheren (aanvullen, aanpassen, ...) en vervolgens dan terug gaan sturen naar alle andere systemen. Zo tracht men van bepaalde gegevens (zoals bv. klantgegevens) één waarheid te hebben binnen een onderneming. Als je diverse systemen met elkaar moet gaan integreren, krijg je vaak verschillende waarheden voor hetzelfde object (bv. geboortedatum van één persoon is anders bij facturatie als bij marketing). Ook daar kunnen Graph databases helpen om te achterhalen welk systeem het bij het rechte eind heeft.

Real Time Recommendations gaat nog een stap verder. Als je op zoek gaat naar een bepaald product, krijg je als klant vaak een lijst van ofwel vergelijkbare producten, ofwel producten die te maken hebben met het product dat je koopt. Bv. je zoekt een laptop op bv. bol.com en je krijgt nog een lijstje met andere vergelijkbare laptops, of andere accessoires (extra geheugen, ...) die mogelijk interessant kunnen zijn.

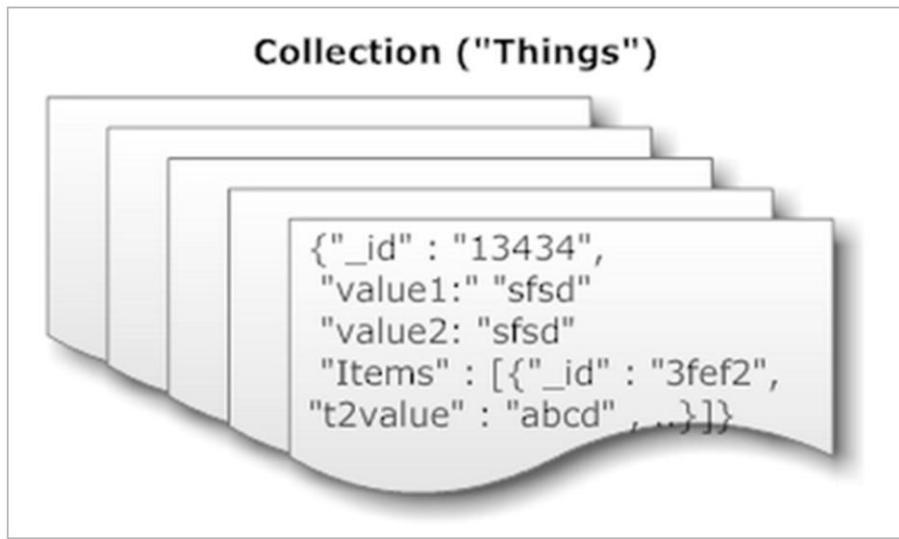
Na Social Computing is waarschijnlijk Fraud Detection het meest gekende gebruik van Graph Databases. Denk daarbij aan de Panama papers affaire die enkele jaren terug heel veel belastingontduiking wereldwijd aan het licht bracht. Vaak door data die meestal in verschillende departementen verspreid zit die gaan koppelen met elkaar (zie ook Master Data Management), ga je gegevens kunnen koppelen en die verder gaan bestuderen. Zo zijn er ook al enkele BTW-carrousels aan het licht gekomen (= bedrijven blijven doorfactureren via diverse bedrijven naar elkaar zonder diensten/goederen te leveren).

Je kan bij Neo4J trouwens nog een uitgebreide lijst van use-cases bekijken om te zien waar je deze kan gebruiken: <http://neo4j.com/use-cases/>

Document Store

Document store

Data bijhouden in documenten en documenten bundelen in collecties



Document Stores zijn op dit moment de meest gebruikte vorm van NoSql database. Hier spreken we niet meer van key-value pairs of nodes, maar van 'Collections' waarin 'Documenten' bewaard. Een document is dan weer een set van gegevens die logisch bij elkaar horen. Die set van gegevens wordt dan in een hiërarchische structuur (JSON, XML, ...) bewaard. Dit type van NoSql databank is één van de meest gebruikte alternatieven van RDBMS om de data van applicaties in te gaan beheren.

Je gaat dus als het ware je data groeperen in documenten, die je dan op zich gaat groeperen in Collecties. Elk document binnen een collectie mag ook zijn eigen structuur hebben, er is dus géén vast schema waaraan een document moet voldoen.

Een aantal document stores bieden die functionaliteit wel aan, maar het is zeker geen vereiste - en eigenlijk ook geen best-practice - om dit toe te passen. Herinner je maar hetgeen we in het hoofdstuk Big Data besproken hebben.

Document store

Een document database bestaat uit collecties van **documenten**. Een document is een **hiërarchische structuur** (XML, JSON,...) waarin gegevens zitten die 'logisch samen horen'

Een document is meestal niet volledig genormaliseerd en kan lijsten van child documents of andere geaggregeerde gegevens bevatten.

Men spreekt van aggregates en aggregate orientation

In een document database kan je verschillende collecties definiëren. Typisch ga je eenzelfde soort 'entiteiten' verzamelen in één collectie, maar dat is op zich geen verplichting.

Een collectie is eigenlijk een verzameling documenten. Zulk een document hoeft niet een "platgeslagen" set van gegevens te bevatten, maar kan op zich ook child documents of andere geaggregeerde gegevens te bevatten. Zo kan bijvoorbeeld een document rond 'Order' een lijst van 'Orderlines' te bevatten en bevat een order document ook een attribuut 'TotalSalesPrice' dat alle verkoopprijzen van de bijhorende 'Orderlines' optelt.

Dit concept om in één 'document' meer dan één entiteit te bewaren, noemt men ook wel aggregates en 'aggregate orientation'.

Door deze techniek toe te passen in het modeleren van je documenten, kan je héél efficiënt je data gaan ophalen en bewaren. Maar het vergt wel een ander denkpatroon om zulk een model op te stellen dan dat je klassiek gewoon bent. De focus ligt bij dit type databanken niet op de normalisatie-technieken en dus het vermijden van redundante data, maar meer op zo efficiënt mogelijk benaderen van de data.

Aggregate orientation

Intermezzo

English - detected ▾



Dutch ▾



to aggregate Edit

aggregeren

verb

samenvoegen

combine, join, conjoin, joint, unite, aggregate

opeenhopen

accumulate, aggregate, conglomerate, agglomerate, amass, heap up

verzamelen

collect, gather, cumulate, compile, congregate, aggregate

bijeenbrengen

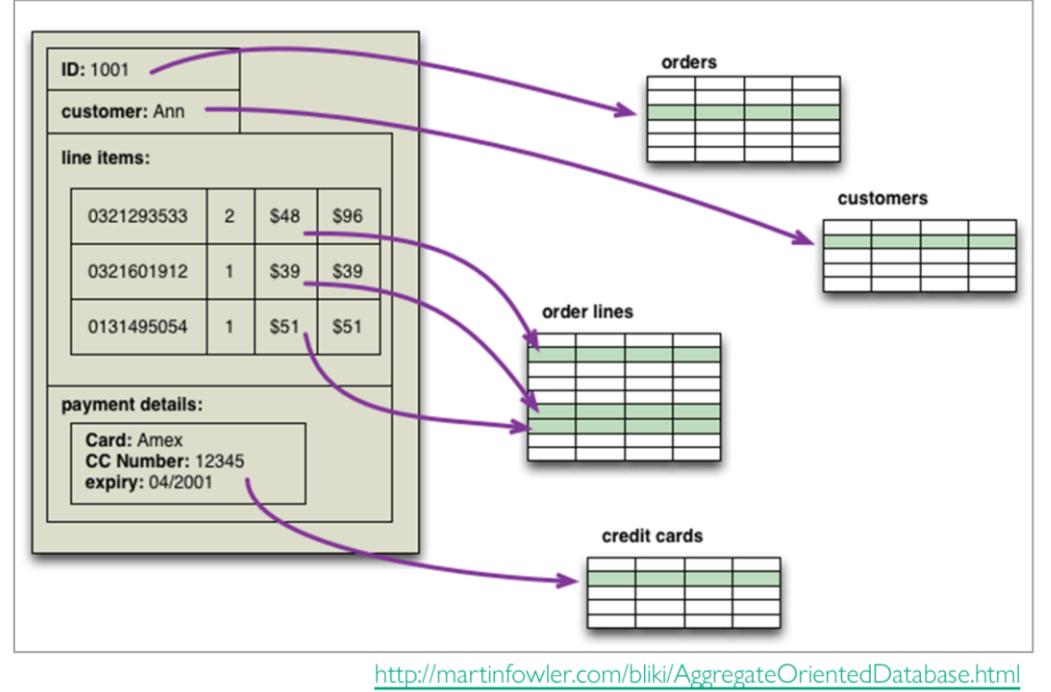
gather, collect, accumulate, aggregate, muster, mass

Aggregate is een term die in de (relationele) databank-wereld ook al gebruikt wordt. Denk maar aan de aggregation functions die vaak in combinaties met een group-by gebruikt worden.

'To Aggregate' wil eigenlijk zeggen 'samenvoegen' van gegevens. In SQL gaat dit over het samenvoegen van gegevens via functies zoals AVG, SUM, MIN, ... op basis van een bepaalde set van velden die je via GROUP-BY definieert.

Aggregate Orientation in de NoSql-wereld gaat niet over hoe je de gegevens opvraagt, maar hoe je ze structureert. In plaats van alle entiteiten op te splitsen in verschillende tabellen en daartussen relaties te leggen, ga je ze net samen in één document verzamelen. Op die manier kan je gegevens die vaak samen opgevraagd worden, sneller opzoeken en ophalen.

Aggregate orientation



Een klassiek voorbeeld van hoe je de gegevens kan aggregeren in één document, is een "Order" scherm. Daarin vind je altijd verschillende entiteiten tussen die opgehaald moeten worden uit diverse tabellen. De klantgegevens komen uit de 'Customer' tabel, ordergegevens uit 'Orders', 'Orderlines' uit weer een andere tabel en dan heb je nog de data over het product per orderline, betalingsgegevens van klanten, ...

Om dus deze set van samenhangende data op te halen, moet je dus verschillende tabellen gaan query'en. Als je daar eens over nadenkt, is dat eigenlijk vanuit een applicatie-standpunt 'complex' om telkens die gegevens te gaan samenrapen en verwerken.

Met de techniek 'Aggregate Orientation' ga je samenhangende data die je samen gaat ophalen, groeperen in één document. Je streeft ernaar om data die nauw aan elkaar samenhangt te groeperen, ook al bestaat de kans dat je daardoor hier en daar redundante data krijgt. In bovenstaand voorbeeld zou je één document 'Order' kunnen maken met daarin verschillende subdocumenten voor de gegevens van een klant, orderlijnen, product, ...

Aggregate orientation

```
{  
  _id: <ObjectId1>,  
  username: "123xyz",  
  contact: {  
    phone: "123-456-7890",  
    email: "xyz@example.com"  
  },  
  access: {  
    level: 5,  
    group: "dev"  
  }  
}
```



Embedded sub-document
Embedded sub-document

Data model with embedded fields that contain all related information.

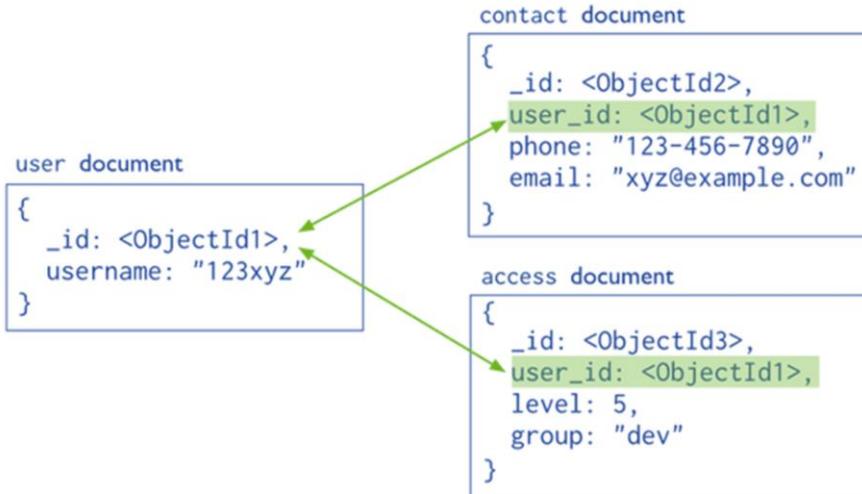
<http://docs.mongodb.org/manual/core/data-model-design/>

Hierboven staat een voorbeeld van een document met bijhorende subdocumenten zoals deze opgeslagen wordt door MongoDB.

MongoDB houdt de data bij in JSON-formaat. Dit document toont de gegevens van een gebruiker. Je hebt `_id`, een surrogate key wat een unieke identifier is van je document. Als attribuut is er een `username` en zijn er twee sub-documenten. Het eerste subdocument noemt 'contact' bevat de gerelateerde contactgegevens zoals telefoonnummer en e-mail adres.

Daarnaast heb je ook het subdocument 'access', waarin zijn rollen bijgehouden worden.

Aggregate orientation



Data model using references to link documents. Both the **contact** document and the **access** document contain a reference to the **user** document.

Je kan ook verwijzingen leggen tussen entiteiten die in andere documenten bewaard worden, vergelijkbaar met foreign keys. Je gaat dan de object_id van het document naar waar je wil verwijzen toevoegen aan je document, net zoals bij een foreign key dus.

Anders dan bij foreign keys, heb je bij Document Store standaard géén "referentiele integriteit". Bij sommige Document Stores kan je dit wel op één of andere manier nabootsen, maar dat wordt sterk afgeraden. Vooral omdat je net een Document Store gebruikt voor de aggrate orientation zodanig dat je 'relaties' zoveel mogelijk tracht te voorkomen.

Aggregate orientation

- Andere denkwijze dan klassieke relationele modelering
 - Vertrek vanuit de noden van de applicatie: wat zijn de typische insert/update/query patronen?
(maw. welke data ga je tegelijkertijd behandelen)
 - Aan de hand daarvan neem je deze data samen in één document structuur
- Verwijzingen (FK's, ...) tussen documenten kan nog, maar minder "gangbaar"
(bv. *klantId opnemen in Order document ipv volledige klantinfo opnemen in Order document*)
- Echte parent-child relaties zoals Order-OrderLines worden sowieso wel altijd samengenomen in één document

Werken volgens de principes van aggregate orientation vergt toch wel wat oefening. Je bent gewoon om alles per entiteit/record te aanschouwen en één record te aanschouwen als één entiteit.

Nu ga je eigenlijk verschillende entiteiten samenbrengen in één document. Om die oefening te doen, moet je dus niet meer vertrekken vanuit de normalisatie-principes, maar vanuit de noden van de applicatie die deze gegevens gaat beheren. Je gaat bekijken wat de typische CRUD-patronen zijn die uitgevoerd moeten worden vanuit de client om zodoende structuur van de documenten daarnaar op te bouwen.

Zoals in de vorige slide al aangehaald, betekent dit niet dat er géén FK's meer kunnen zijn tussen de documenten. Je kan perfect de object id van document B opnemen in document A. Dat ga je typisch doen als er een bepaalde entiteit vaak geaggregaat moeten worden en daardoor te vaak redundant bijgehouden moet worden. In het voorbeeld van enkele slides terug zou je bijvoorbeeld kunnen kiezen om de klant-gegevens niet meer op te nemen in de document-structuur van 'Order', maar wel een verwijzing van het object_id van de klant toe te voegen.

Echte parent-child relaties worden wel zo goed als altijd samengenomen in één document. Je moet al een goede reden hebben om dat NIET te doen.

Het is de balans zoeken tussen het vermijden van 'relaties' tussen de documenten en welke gegevens je al dan niet redundant mag bijhouden. Als je veel parent-child relaties nodig hebt, is eigenlijk een document store databank een zéér goede keuze omdat deze parent-child relaties vaak samen opgehaald en/of gepersisteerd worden.

Aggregate orientation



Voordelen

- Performance winst als je data die vaak samen wordt weggeschreven en opgevraagd samen in één document zit (minder 'joins')
- Een document is vaak een goede kandidaat om in zijn geheel te repliceren tussen nodes (availability, sharding)
- Sterk verbonden entiteiten uit het OO-domein van de applicatie kunnen vaak makkelijk gemapped worden op één document

Indien je een goede opbouw van je documenten hebt, biedt een Document Store met Aggregate Orientation vele voordelen. Doordat de data die vaak samen wordt opgehaald en/of weggeschreven samen bewaard wordt in één document, hoef je ook maar dat ene document op te vragen en/of aan te passen. Je moet dus niet op zoek naar alle gerelateerde data zoals in de klassieke werkwijze, want je hebt ineens alle benodigde data vast. Ook de vertaalslag van de document store naar de applicatie toe wordt dan sterk vereenvoudigd, wat veel tijdswinst kan opleveren.

Ook naar replicatie toe zijn de documenten vaak handig. Een document wordt altijd in zijn geheel gerepliceerd, je kan bepaalde condities instellen i.v.m. sharding van een collectie tussen de nodes, ... Net zoals alle NoSql databases is zulk een storage vanaf begin af ontworpen voor clustering en dus eenvoudig in te stellen om high-availability noden op te vangen.

Aggregate orientation



Nadelen

- Je hebt de nadelen van denormalisatie (multiple updates voor dezelfde data, ...)
 - Bv. indien de klant details gegevens zowel mee opgenomen hebt in elk order alsook en die klant gegevens wijzigen, moet je alle orders met die klant in aanpassen. Maar dan weer niet als je net die historische gegevens wil bewaren (zie SCD bij DWH)
- (Te?) Sterke koppeling tussen applicatie domein model en databank structuur
 - Echter meeste document databases zijn schemaless. Wijzigingen in het domein model kunnen vrij gemakkelijk opgeven worden en hoeven niet altijd te leiden tot aanpassingen in de achterliggende database

Dat denormalisatie voordelen heeft, vooral naar queries toe, hebben we al uitvoerig besproken. Maar uiteraard mogen we de nadelen van denormalisatie niet vergeten. Éénzelfde set van gegevens kan in verschillende documenten en vaak dan ook in verschillende collecties bewaard worden, bv. de naam van een klant. Je mag niet vergeten om als de klantnaam wijzigt, alle documenten (in mogelijk meerder collecties) te gaan updaten. Maar pas op, misschien wil je dat niet altijd, denk maar aan het slowly changing dimension type dat je eventueel gebruikt. Zo kan het zijn dat - als je SCD type 2 gebruikt - je net niet de andere klantgegevens wil aanpassen....

Het voordeel dat de structuur van de documenten nauw samenhangt met de applicatie, heeft ook zijn nadelen. 'Loosly coupling' is net één van de belangrijkste principes bij applicatie-ontwikkeling om verschillende lagen zo onafhankelijk mogelijk van elkaar te laten bestaan (denk maar aan unit testing). Zo minimaliseer je de kans dat als de implementatie wijzigt in één laag, een andere laag ook aangepast moet worden.

Dat principe zet je eigenlijk hier 'op de helling' omdat de koppeling tussen het domein en de structuur van de databank zo sterk is. Wijzigingen in het domein kunnen een grote impact hebben op de opbouw van je documenten.

Om dit zoveel mogelijk tegen te gaan, zijn de meeste document databases schemaless. Dit houdt in dat de documenten uit een collection qua structuur niet hoeven te voldoen aan allemaal dezelfde structuur. Zo kunnen wijzigingen in het domeinmodel makkelijker opgevangen worden en niet per se leiden tot aanpassingen van bestaande documenten.

Document store - MongoDB

EXAMPLE



- Werkt met JSON documenten die je stockeert in (key-value) collections
- Schemaless, maar optie tot checking wel mogelijk mbv Mongoose
- Out-of-the box voorziening voor replicatie (replica sets met automated master election) en sharding
- Aggregation pipeline en map-reduce support
- Veel gebruik in (javascript based) web projecten

Andere document stores



CouchDB



...

Als voorbeeld voor een Document Store gebruiken we MongoDB (<https://www.mongodb.com/>).

Als document store gebruikt MongoDB het concept van collecties waarin je documenten gaan in groeperen. De collecties ga je als een key-value benaderen zodanig dat elke collectie zijn eigen (unieke) naam heeft, maar binnen een collectie een reeks documenten aanwezig zijn waarvan het formaat niet op voorhand vastgelegd is.

Daarnaast is MongoDB in principe schemaless wat wil zeggen dat de documenten niet moeten voldoen aan een vooraf gedefinieerde structuur. Het is echter wel mogelijk om bepaalde checks in te stellen zodanig dat je controles kan uitvoeren of de documenten aan een bepaalde structuur voldoen.

Er is ook out-of-the-box ondersteuning voor replication en sharding. Die is ook vrij eenvoudig op te zetten, wat je ook gaat doen in de tutorial en voor het examenproject. Om MongoDB te querien gebruik je géén SQL-statements maar een soort van javascript functies. Daarmee kan je Aggregation pipelines en map-reduce functies uitvoeren op je collecties. Deze laatste behandelen we niet in deze cursus maar aggregation pipelines komt wel aan bod..

Andere bekende document stores zijn "CouchDB" (<http://couchdb.apache.org/>) en "RavenDB" (<https://ravendb.net/>).

Verderop in dit hoofdstuk gaan we MongoDb meer in detail bekijken. Er is ook een tutorial beschikbaar waarmee je MongoDb kan leren kennen. Ook voor het examenproject zal je MongoDb moeten gebruiken.

Document store

Wanneer gebruiken?

- Je hebt veel entiteiten in je domein die zich in een parent-child relatie tot elkaar verhouden en dus samen kunnen behandeld worden
- Niet alle instanties van een entiteit hebben dezelfde eigenschappen
- Voorbeelden:
 - Product information, time series data (resultaten van diverse sensoren met daarbij horende het moment van de meting),



Document Store databases worden voornamelijk gebruikt als je entiteiten veel parent-child relaties hebben die je samen kan groeperen in één document. Daardoor kan je veel sneller zulke relaties inlezen dan wanneer deze in aparte documenten (of tabellen in een RDBMS) zit. Je hebt immers naar I/O toe maar één actie (per document) die uitgevoerd moet worden.

Bovendien is er geen strikt schema dat opgelegd is, dus hoeven niet alle documenten uit een collectie dezelfde eigenschappen te bevatten. Zo kan de impact bij nieuwe releases van de client beperkt blijven (oude data kan geldig blijven), kan je overervingsstructuren in éénzelfde collectie bijhouden, ...

Binnen de Document Store databases die er op de markt beschikbaar zijn, is MongoDB veruit de meest populairste. CouchDB en Couchbase vervolledigen de top 3, maar volgen al op ruime afstand. RavenDB is vooral gekend binnen de .NET wereld, maar wordt nog niet zo courant gebruikt.

Search Engines

Search engines – Full text search

- Datastructuur om snel informatie op te zoeken.
- Formaat vergelijkbaar met document store
- Extra: Full Text Search



Een belangrijke categorie in het noSQL verhaal zijn de search engine databanken. Als je er ooit over de werking van een search engine hebt nagedacht zal je ook wel begrepen hebben dat daar geen gewone relationele databank achter zit.

Eigenlijk kunnen we in deze niet echt spreken van een afzonderlijke categorie. Search engine database maken meestal gebruik van andere noSQL technieken zoals graph databases en document stores. Bij de populaire search engine database Elastic Search is de data gesstructureerd in een document store. Je maakt dus documenten aan (veelal in json formaat) waarin je heel snel en efficiënt kan zoeken. Dit doen ze door zeer gerichte indexering toe te passen dat dit mogelijk maakt.

Search engines – Full text search

- Elk “full text searchable field” krijgt een inverted index
- Inverted index:
 - Een veld wordt opgesplitst in tokens (+- woord)
 - Op de tokens worden normalization rules toegepast
 - ✓ Stemming: vb. Meervoud naar enkelvoud (foxes -> fox), werkwoord naar infinitief... (jumped -> jump)
 - ✓ Lowercase: Quick -> quick
 - ✓ Synonyms: leap -> jump
 - ✓ ...
 - Voor elke genormaliseerde token wordt bijgehouden in welk document het voorkomt

The diagram illustrates the process of creating an inverted index. It shows two input documents on the left and a resulting inverted index table on the right. A large purple arrow points from the input to the output.

Term	Doc_1	Doc_2
Quick		x
The	x	
brown	x	x
dog	x	
dogs		x
fox	x	
foxes		x
in		x
jumped	x	
lazy	x	x
leap		x
over	x	x
quick	x	
summer		x
the	x	

Term	Doc_1	Doc_2
brown	x	x
dog	x	x
fox	x	x
in		x
jump	x	x
lazy	x	x
over	x	x
quick	x	x
summer		x
the	x	x

Om de werking van “search engine databases” uit te leggen maak ik gebruik van de populairste speler op de markt “ElasticSearch”. Net zoals vele andere spelers maakt deze gebruik van Apache License. De manier van werken is daarom meestal vergelijkbaar.

Standaard wordt elk veld in de opgeslagen documenten geïndexeerd. Op string velden wordt hiervoor standaard een “full text” index geplaatst. Deze maakt het mogelijk om documenten op te zoeken die “in bepaalde mate” overeenstemmen met je zoekopdracht. De mate van overeenstemming worden vastgelegd in een soort van score die aan het gevonden document wordt toegewezen.

Het indexing mechanisme maakt gebruik van een “inverted index”. In de afbeeldingen onderaan zie je hoe die tot stand komt.

Hier zijn twee documenten geïndexeerd waarin het veld content volgende zinnen bevat

1. The quick brown fox jumped over the lazy dog
2. Quick brown foxes leap over lazy dogs in summer

Een inverted index gaat eerst de string in tokens opsplitsen. Meestal wil dit zeggen dat er gesplitst wordt op spaties om woorden te bekomen.

In een volgende stap worden die woorden genormaliseerd. Op de woorden worden een aantal transformaties gedaan die er voor zorgen dat afgeleide woorden allemaal als hetzelfde beschouwd worden. Belangrijke technieken zijn hier stemming en synonyms (die taalafhankelijk zijn) en lowercase.

Search engines – Full text search

- Het splitsen in tokens en het normalizeren van tokens gebeurt door een Analyser
- Analyser is volledig configurerbaar
- Een full text search levert een resultaat met een matching score die aangeeft hoe relevant het document is volgens de zoekopdracht.
- De analyser wordt uitgevoerd tijdens het indexeren en op de zoekquery

```
{ "query" : { "match" : { "about" : "rock climbing" } } }
```

```
{
  ...
  "hits": [
    {
      "total": 2,
      "max_score": 0.16273327,
      "hits": [
        {
          ...
          "_score": 0.16273327,
          "_source": {
            "first_name": "John",
            "last_name": "Smith",
            "age": 25,
            "about": "I love to go rock climbing",
            "interests": [ "sports", "music" ]
          }
        },
        {
          ...
          "_score": 0.016878016,
          "_source": {
            "first_name": "Jane",
            "last_name": "Smith",
            "age": 32,
            "about": "I like to collect rock albums",
            "interests": [ "music" ]
          }
        }
      ]
    }
}
```

Hoe dat je dat tokenizen en normaliseren juist doet bepaal je zelf (maar je kan ook gebruik maken van default settings). Dit doe je door de zogenaamde Analyzer te configureren. Je beslist zelf welke regels er moeten toegepast worden en je kan dit vrij aanpassen naargelang het veld.

Als je een zoekterm lanceert zal de analyzer die zoekterm verwerken en met die set van tokens de inverted index raadplegen. Op basis van de correctheid van de match wordt er dan een score aan de documenten toegekend die op één of andere manier overeenstemming hebben met de zoekterm. Hoe hoger de score, hoe relevanter het resultaat.

Search engines store

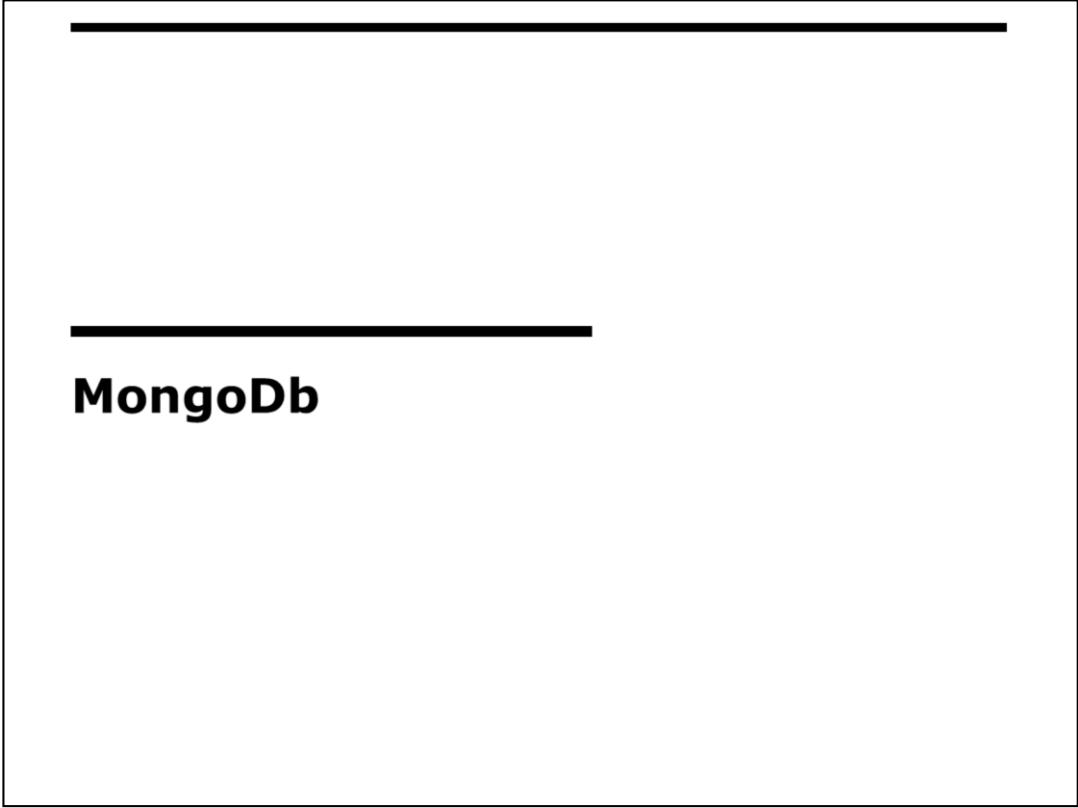
Wanneer gebruiken?

- Je wil gegevens eenvoudig doorzoekbaar maken
- Je wil kunnen zoeken op relevantie niveau i.p.v. exacte waarden.
- Andere typische document store use cases



Het spreekt voor zich dat search engines gebruikt worden om gegevens eenvoudig doorzoekbaar te maken. Je hebt dus een document store ter beschikking met ook dezelfde eigenschappen met dat verschil dat je ook zeer efficiënt kan zoeken.

Eigenlijk zijn de mogelijkheden oneindig. Je kan er bijvoorbeeld voor kiezen om een selectie van de data van Human Resources beschikbaar te maken om personen te kunnen zoeken (zelfs al kan je de naam niet exact spellen). Je kan ook alle documenten in een bedrijf indexeren zodat je snel informatie kan doorzoeken. Zo kan de juridische dienst snel vinden in welke documenten een mogelijk ongeldige clausule werd opgenomen.



MongoDb

Introductie

- Populairste Document Store
- Databases
 - => Collections
 - => Documents
- BSON
 - binaire JSON
 - Compacter dan andere formaten
 - Géén (definitieve) validatiestandaard
 - Géén standaard querytaal, ook JSONPath (nog?) niet

```
{  
  "firstName": "John",  
  "lastName": "Smith",  
  "age": 25,  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": 10021  
  },  
  "phoneNumbers": [  
    {  
      "type": "home",  
      "number": "212 555-1234"  
    },  
    {  
      "type": "fax",  
      "number": "646 555-4567"  
    }  
  ]  
}
```

MongoDB is een van de meest populaire NoSQL databanken en valt onder de categorie van de **document stores**. Zie de slides rond NoSQL voor de noodzakelijke omkadering, concepten en toepassingen.

De belangrijkste structuren binnen MongoDB zijn:

Database

Net zoals in een klassieke databank kan een MongoDB server meerdere databanken bevatten.

Collection

Een databank bestaat uit een of meerdere *collections*. Een collection is een lijst van documenten.

Document

Een Document in MongoDB is een hiërarchische structuur die intern wordt opgeslagen als BSON, een binaire vorm van JSON.

JSON wordt vooral gebruikt als transport formaat bij REST en Ajax calls. Het is compacter (minder ‘verbose’) dan XML (omdat niet wordt gewerkt met <tags>) en dus beter geschikt voor transport over verbindingen met een lagere bandbreedte zoals 3G netwerken.

Een nadeel van JSON is dat er nog geen definitieve validatiestandaard voor is vastgelegd (json-schema.org). Dit maakt het minder geschikt voor B2B integraties waar goede afspraken cruciaal zijn.

Ook een standaard query-taal bestaat er nog niet. JSONPath begint wel hoe langer hoe meer als officieuze standaard query-taal aanvaard te worden, maar omdat dit het initiatief is van één persoon en bedrijven vaak hun eigen 'versie' daarvan bouwen, kan het zijn dat sommige queries in de ene omgeving wel werken maar in een andere dan weer niet.

Desondanks deze nadelen wint JSON hoe langer hoe meer aan populariteit. Meer en meer websites ondersteunen JSON en ook programmeertalen en hun framework bieden geleidelijk aan meer ondersteuning. Denk maar aan ASP.NET MVC waar je JSON objecten kan gebruiken om te communiceren van en naar je controllers.

Introductie

- Schemaless
 - Standaard geen controle op structuur document
- Documenten in één collectie hoeven niet dezelfde structuur te hebben
- Hiërarchische structuur (Aggregation Orientation)
 - Modelleren volgens gebruik i.p.v. normalisatie-technieken
- Transacties
 - Atomicity per document
 - Niet ondersteund
 - Wel via complex 2-phase commit, maar wordt afgeraden

45

Geen schema's

MongoDB is **schema-less** wat betekent dat er geen controle is op de structuur van een document. Je kan documenten met een verschillende structuur dan ook in 1 collectie samen zetten. Dit kan handig zijn om bv. twee verschillende versies van een document te ondersteunen bij database migraties, of documenten samen te zetten die wel in beperkte mate van structuur kunnen verschillen (bv. product info).

Het is echter niet aangewezen om totaal verschillende entiteiten in 1 collectie samen te zetten.

Hiërarchische structuur

Documenten kunnen via een hiërarchische structuur opgebouwd worden. Dit wil zeggen dat je bijvoorbeeld parent-child relaties in één document bewaren waardoor je efficiënter bij elkaar horende gegevens kan beheren.

Gezien de schema-less eigenschap van NoSql databases, worden die documenten vaak in een zo vrij mogelijk formaat bewaard. In concreto zal dit meestal Json of XML zijn.

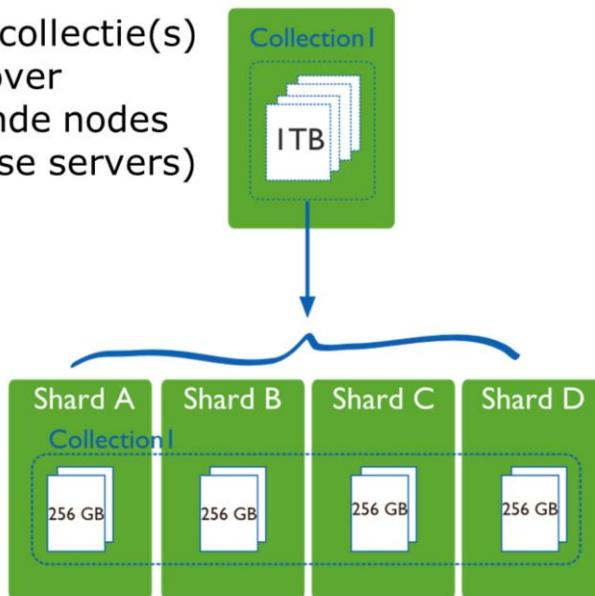
Transacties

MongoDB ondersteunt geen transacties, maar garandeert wel Atomicity voor insert/update/delete operaties op 1 document. Om toch meerdere documenten in 1 'transactie' te bewerken (bv. omdat er referenties tussen documenten liggen), wordt een (omslachtig) 2-phase commit protocol ondersteund.

Sharding



MongoDB collectie(s)
verdelen over
verschillende nodes
(= database servers)



<http://docs.mongodb.org/manual/sharding/>

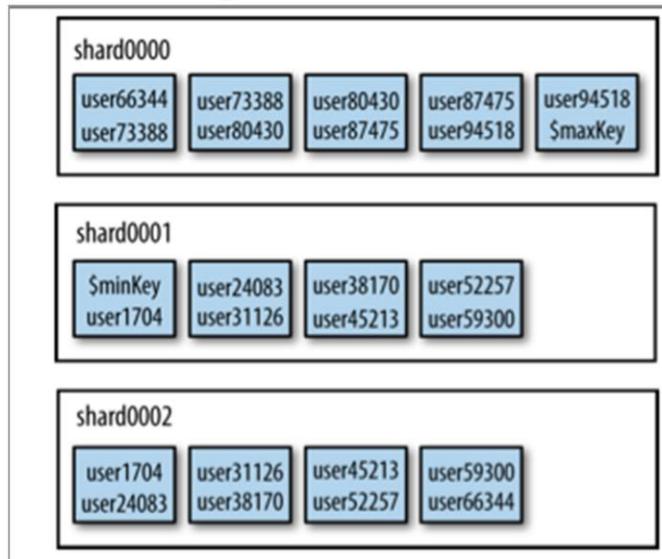
MongoDB kent het concept van collections oftewel een verzameling van bij elkaar horende documenten. Die documenten moeten niet allemaal dezelfde structuur hebben maar in realiteit zal dat wel vaak het geval zijn.

Sharding kan toegepast worden op het niveau van collections. Dat wil zeggen dat collecties kunnen verdeeld worden over verschillende nodes.

Sharding



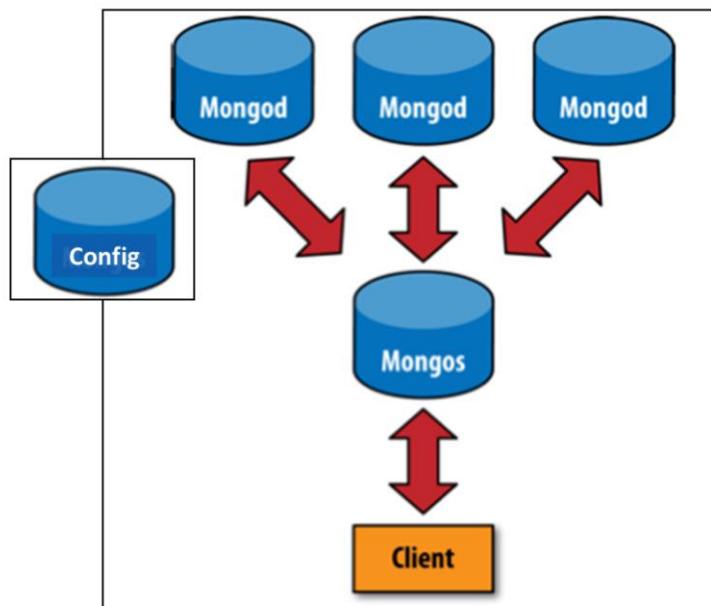
Voorbeeld: sharding van een collection 'users'



Op deze slide zie je hoe een 'users' collection verdeeld kan worden over verschillende nodes.

Op basis van een 'Shard Key' worden de verschillende documenten die in een collectie bewaard worden verdeeld over de verschillende shards. Die verdeling gebeurt automatisch, daar moet je als client eigenlijk weinig voor doen. Over die 'Shard Key' hebben we het later nog.

Sharding



Om sharding (en ook replicatie) toe te kunnen passen gebruikt MongoDB een aantal componenten, waarvan je hierboven een overzicht ziet.

- "Mongod" instanties draaien op elke node waar een shard moet worden opgeslagen.
Typisch ga je de 'Mongod' verspreiden over verschillende servers.
- Een "Mongos" instantie staat in voor het routen van reads, inserts en updates. Deze zal door de client aangesproken worden waardoor je een 'single-point-of-access' krijgt
- De "Config" instantie houdt bij waar welke gegevens exact zijn opgeslagen zodanig dat de Mongos weet waar hij welke documenten kan ophalen.

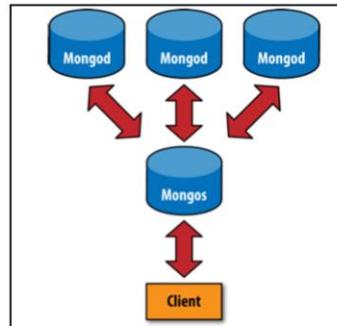
De clients gaan ook géén SQL kunnen sturen richting Mongos. Er zijn voor de meest gangbare programmeertalen API's beschikbaar waarmee je MongoDB via de Mongos kan aanspreken (zie <https://docs.mongodb.org/ecosystem/drivers/>).

Sharding

Op elke shard node draait een MongoDB server (**mongod**)

Applicaties praten niet rechtstreeks met deze servers maar met een 'router' (**mongos**)

Bij insert van een document stuurt mongos het document naar 1 van de shards

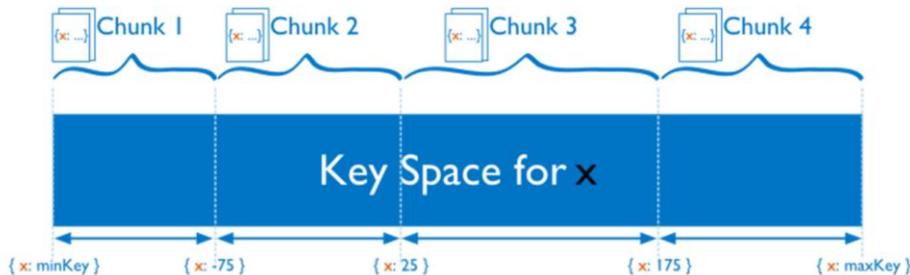


Bij het opvragen van data stuurt mongos de query door naar de shards (query routing), deze sturen de gevonden documenten terug naar mongos die ze aflevert aan de applicatie

De mongos router is *stateless* (geen data dir) en wordt meestal op de app server gedraaid

In tegenstelling tot de klassieke (R)DBMS, ga je (een deel van) MongoDB installeren op de application server. Klassiek doe je dit nooit om de belasting van je application server géén impact te laten hebben op deze van je DB-server, maar bij MongoDB kan dit wel. De component Mongos is eigenlijk een router die de binnenkomende requests vanuit de applicatie ontvangt. Op basis van de configuratie, weet hij naar welke Mongod (of meerdere Mongod's) hij de request moet doorsturen. Eens de request doorgestuurd is, wacht hij op de resultaten (documenten) daarvan. Indien deze van verschillende Mongod's komen, zal hij deze bundelen en in één pakket terugsturen naar de client.

Shard Key



Op basis van de shard key worden de documenten in een collection opgedeeld in chunks. Chunks worden toegewezen aan shards.

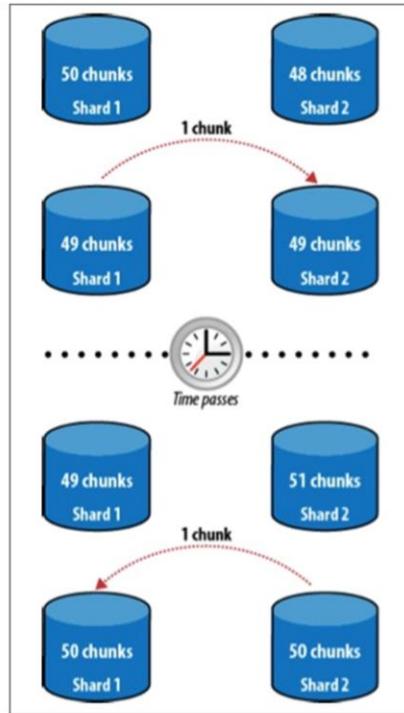
Een **config server** houdt bij op welke shard de chunks zich bevinden en welke range van de shard key in elke chunk zit

Chunk 1	—————>	shard 1
Chunk 2	—————>	shard 1
Chunk 3	—————>	shard 2

De manier waarop data verdeeld wordt over de shards is vrij belangrijk. Je moet daarom goed nadenken over een shardkey. Die sleutel (die meestal een functionele waarde heeft voor het document) gaat bepalen in welke shard een document terecht moet komen. De shardkey deelt een collectie namelijk op in chunks, en elke chunk wordt aan een shard toegewezen.

Eén shard kan voor één collectie meerdere chunks opslagen, dus je kan meer chunks hebben dan er shards (= Mongod) zijn.

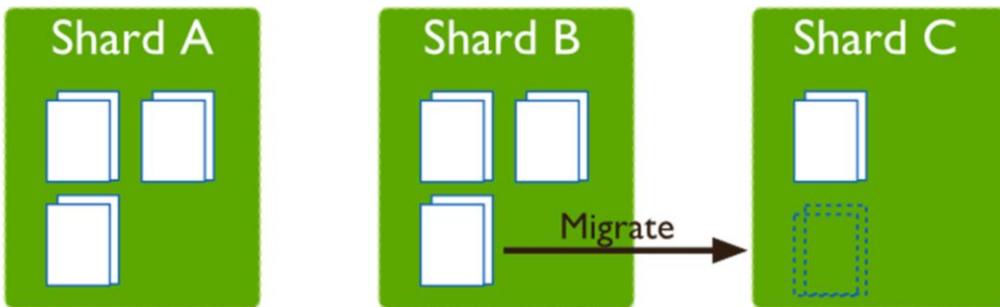
Chunk migration



MongoDB tracht de chunks gelijkmatig te verdelen over de verschillende shards. Als er een onevenwicht ontstaat zal een chunk worden verhuist van de node met een overgewicht naar een node met een ondergewicht. Dit gebeurd ook automatisch, als 'DBA' of ontwikkelaar hoef je daar niets voor te doen.

Op die manier verdeelt MongoDB de data en dus ook de load gelijkmatig over de verschillende shards.

Chunk migration



Wanneer een shard teveel chunk's bevat worden chunk's door de 'Balancer' verplaatst naar een andere shard.

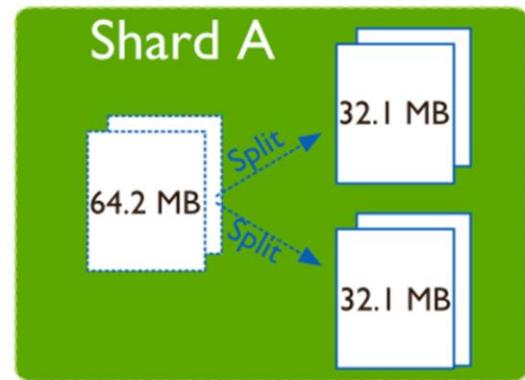
Een **config server** houdt bij op welke shard de chunks zich bevinden en welke range van de shard key in elke chunk zit

De config server van MongoDb speelt in héél dit verhaal een belangrijke rol. Deze gaat namelijk bijhouden hoe de chunks verdeelt zijn over de Shards. Bovendien gaat die ook bijhouden welke range van de shared key in welke chunk zit, zodanig dat hij over hij binnenkomende request direct kan achterhalen waar (welke shard) de gevraagde documenten zijn.

Chunk splitting



Wanneer een chunk te groot wordt, dan wordt deze door de 'Balancer' gesplitst, zodat hij eventueel kan verhuisd worden naar een andere shard



Default chunk size is 64 MB.

Chunks kunnen ook te groot worden. Dit doet zich voor als binnen een bepaalde range van de shardkey te veel documenten opgeslagen moeten worden. In dat geval gaat de balancer die shunk opsplitsen en dus ook de ranges aanpassen. De kleinere chunks kunnen nadien eventueel naar andere shards (en dus nodes) worden verhuisst.

Shard key



<http://docs.mongodb.org/manual/tutorial/choose-a-shard-key/>

De keuze van de shard key is belangrijk voor een optimaal gebruik van sharding

1. High cardinality

Een shard key met **veel verschillende mogelijke waarden**, laat mongoDB toe om documenten evenwichtig te verdelen (bij insert en via *chunk splitting*) over de shards

Documents met dezelfde shard key zitten altijd in dezelfde chunk.

Een shard key met een te lage cardinality kan er dus voor zorgen dat chunks niet gesplitst kunnen worden om ze te verdelen over shards.

(bv. *in een customer collection is 'gender (M/V)' geen goede shard key*)

Bij de keuze van de shard key, moet je rekening houden met drie kenmerken:

- High Cardinality: zorg voor optimale spreiding van de shards door veel variatie in de waarden van de shard key te voorzien
- Write Scaling: insert operaties verspreiden over de shards
- Query Isolation: documenten die samen gequeried worden, kunnen best bewaard worden in dezelfde shard

High Cardinality wil dus zeggen dat de shard key *véél* verschillende mogelijke waarden moet bevatten. Zo kan de verdeling van de documenten zo evenwichtig mogelijk gebeuren. Stel je kiest als shardkey 'geslacht'. Die shardkey heeft slechts 2 verschillende mogelijkheden. Er kunnen ten hoogste twee chunks worden aangemaakt die nooit verder kunnen worden opgesplitst. Alle documenten gaan dus opgeslagen worden op exact twee nodes.

Bij een heel grote databank zou zelf leeftijd een slechte key zijn. In dat geval zouden er maar iets van een 110 chunks kunnen bestaan wat nog steeds relatief weinig is.

Shard key



2. Write scaling

De verdeling van de shard key moet voldoende random zijn zodat bulk write (insert/update) operaties door mongos goed verdeeld kunnen worden over de shards

(bv. een timestamp is meestal geen goede shard key, want dan komen opeenvolgende inserts op dezelfde shard terecht)

3. Query isolation

Indien de shard key onderdeel is van de where clause van vaak voorkomende queries, kunnen deze performant worden uitgevoerd.

Voor dergelijke queries kan mongos immers via de shard key bepalen op welke shard(s) de data zich bevindt. In het ideale scenario zit alle op te vragen data op 1 shard.

(bv. 'city' indien vaak alle documenten voor een bepaalde stad worden opgevraagd)

Write scaling houdt in dat opeenvolgende inserts best niet in dezelfde shard gebeuren. Op die manier kunnen insert en update operaties zo optimaal mogelijk verdeeld worden en dus de performantie verhogen omdat de processing time over de verschillende nodes verspreidt worden.

Met Query isolation ga je dan weer proberen de shard key zodanig te kiezen, dat data die vaak samen opgehaald moet worden, ook in dezelfde shard zit. Als je meest voorkomende queries steeds maar één shard moet aanspreken, zal dit altijd sneller gaan dan dat hij meerdere shards moet aanspreken.

Je kan eigenlijk wel een vergelijking maken met indexes in een relationele databank. Daar ga je indexes gebruiken op velden die vaak gebruikte in WHERE-clausule, terwijl je bij MongoDb je shard key 'intelligent' kan gebruiken om zoekoperaties te versnellen.

High cardinality - Write Scaling - Query Isolation

Soms conflicterende belangen. Een goede shard key voor write scaling kan een slechte shard key zijn voor isolation.

bv. *random (hashed) shard key geeft ideale write scaling maar slechte query isolation*

Alternatieve manieren om shard key te bepalen

- Composite shard key
(bv. city + username als city een te lage kardinaliteit heeft)
- Bereken een ideale shard key in de client applicatie en voeg die toe in elk document

De drie besproken kenmerken kunnen helaas ook elkaar 'tegenwerken'. Er zijn soms conflicterende belangen tussen het optimaliseren voor data weg te schrijven (write scaling) en op te halen (query isolation). Ook bij de klassieke relationele databanken is dat zo, denk maar aan indexes. Met een index kan je zoekoperaties drastisch versnellen, maar insert/update/delete operaties gigantisch vertragen...

Hetzelfde geldt bij MongoDb voor de keuze van de Shard Key...

Je kan ook een shared key uit meerdere velden samenstellen om de cardinaliteit, write scaling en/of query isolation te vergroten. Men spreekt dan over een 'composite shard key'. Of als je zelf voor de type documenten een ideale shard key kan berekenen in de applicatie, kan je uiteraard die gebruiken.

Het type 'Shard key' dat we tot hiertoe besproken hebben, is de range shard keys. Hierbij werden de shards opgedeeld in ranges: vb. 1900 -1920, 1920-1940...). Er zijn echter daarnaast ook hash keys, waarbij er voor elk document een hash code berekent wordt. Die hashcode zorgt er dan voor dat de documenten gelijkmatig verspreid worden over de shards. Er zal dus geen onevenwicht ontstaan in grootte van de shards.

Een nadeel is hier echter dat het uitvoeren van een query op een bepaalde range (gebruikers met een geboortedatum tussen 10-02-1990 en 10-02-2010) er voor gaat zorgen dat bijna alle nodes moeten aangesproken worden. Goed voor write scaling en high cardinality, slecht voor Query isolation.

Je merkt dat het kiezen van een shardkey een moeilijke evenwichtsoefening is met altijd een trade-off tussen write scaling en query isolation.

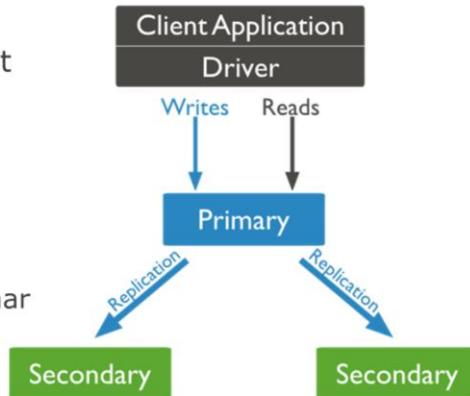
Replica sets



= Replicatie mechanisme voor het bekomen van high availability

Shared-nothing master-slave architectuur

- Alle writes gaan naar primary (master)
- Asynchrone replicatie via oplog naar secondary's (slaves)



<http://docs.mongodb.org/manual/replication/>

MongoDB werkt voor replicatie met zogenaamde replica set. Een replicaset is een groep van mongod instanties die dezelfde dataset (shard) beheren. Voor elke replica set is er een primary node die als master werkt. Deze krijgt alle writes binnen. Deze writes komen op een log terecht die gebruikt wordt om de gegevens naar de secondary nodes te repliceren, vergelijkbaar dus met hoe MySql het doet

Bij MongoDb gebeurt heel dit proces wel asynchroon wat wil zeggen dat de consistentie op de secondary nodes niet gegarandeerd is. Reads kunnen eventueel afgehandeld worden door secondary nodes maar deze gegevens kunnen nog niet volledig up to date zijn. Maar zoals al enkele keren aangehaald in dit hoofdstuk, moet je je afvragen of dit wel een groot probleem is. De praktijk leert ons dat dit niet onoverkomelijk is, zeker als je met omvangrijke hoeveelheden data werkt en/of gegevens die continu wijzigen.

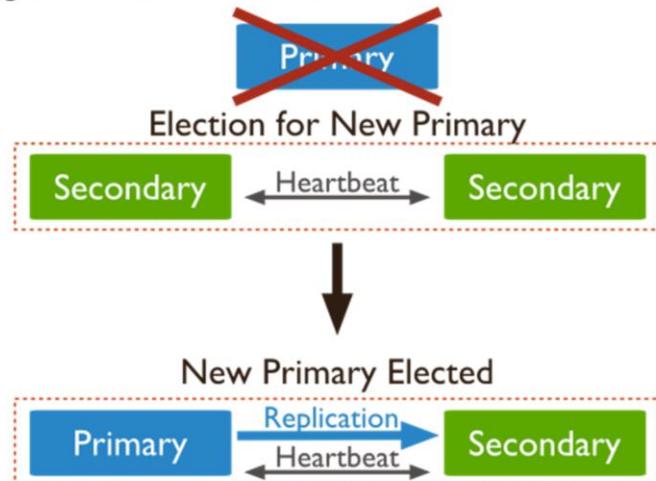
Een replica set moet minstens één node bevatten die dan als primary fungert, maar om de high-availability te garanderen is het aangeraden minimum aantal nodes per replica set 3. Binnen één Replica Set kunnen er maximaal 50 nodes zijn.

Replica sets - elections



Alle nodes in een replica set pingen elkaar continue (heartbeat) om te kijken of er een node down is

Wanneer de primary uitvalt wordt een van de secondary's tot primary gekozen d.m.v. een *election*



Als je een MongoDB met replica sets opzet, zal er bij een failure van de primary een automatische fail-over gebeuren. Het detecteren van een uitval van de primary node doet hij ook automatisch. Alle nodes in een replica sets pingen continue naar elkaar om te zien of de andere node nog actief is.

Indien de primary niet meer reageert wordt deze als inactief beschouwd en moet er een andere node als primary gaan fungeren. Om te bepalen welke van de secondary nodes er wordt gepromoveerd wordt er een 'verkiezing' gehouden. Deze election gebeurt door een soort van voting-systeem die door de nodes in de replicaset wordt uitgevoerd. Het resultaat daarvan bepaald dus wie de primary mag zijn.

Replica sets - Arbiters en elections

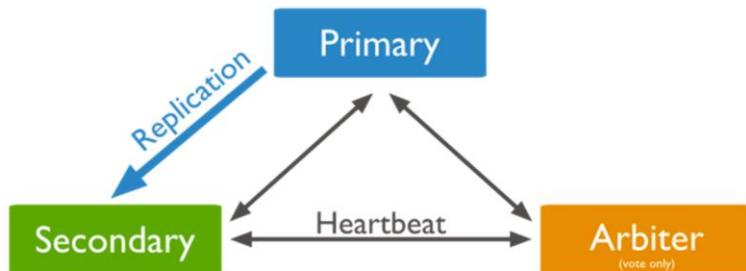


Arbiters kunnen toegevoegd worden aan een replica set om elections vlotter te laten verlopen.

Arbiters hebben geen kopie van de data.

Elections verlopen volgens een zeer strikte set regels.

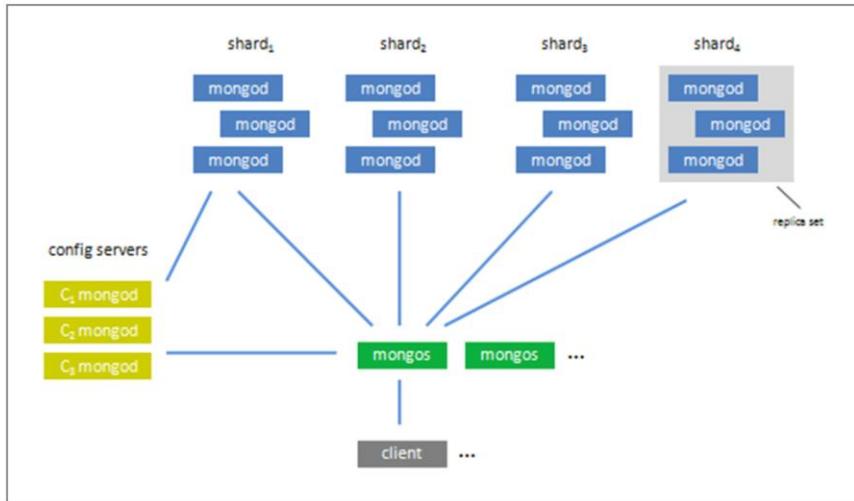
We gaan hier niet dieper op in.



Het algoritme dat gebruikt wordt tijdens die elections is niet eenvoudig, volgt strenge regels (performantie, response times, ...) en verloopt ook automatisch. Omdat je typisch wil dat in geval van een uitval van de primary node je zo snel mogelijk een nieuwe primary beschikbaar wil hebben staan, kan je een 'Arbiter' toevoegen aan een 'replica set' die het proces van de elections versnelt. Het belangrijkste verschil is dat dan niet de replica sets onderling gaan beslissen wie de nieuwe primary wordt, maar de arbiter zal dat bepalen.

De details van zulke elections valt out-of-scope van deze cursus.

Sharding & Replication



Elke shard, alsook de config server opzetten als een replica set zorgt voor een performante en high-available database

Naast de shards (data) zelf kunnen ook config servers als replica set opgezet worden en dus bijgevolg ook de mongo's. Op deze manier wordt de availability gegarandeerd. De architectuur hiervan uitdenken is niet zo eenvoudig, maar hoe het hele systeem technisch uit te rollen daarentegen is niet zo heel moeilijk.

Als je je MongoDB via deze architectuur opzet, heb je zowel naar performance alsook high-availability een zéér sterke oplossing.

Vanaf MongoDB versie 3.4 is het trouwens verplicht om voor per shards een replica set te voorzien. Het is mogelijk om enkel een primary node in je replica set te voorzien (dus maar één node in een replica set) maar dan heb je natuurlijk niet echt een high-availability oplossing voor je shard.

Ook voor de config server ben je verplicht te werken met een replica set sinds MongoDB 3.4. Ook hier kan je een replica set van één node definiëren maar uiteraard gebruik je er beter meerdere.

MongoDb

Tutorial => Zie Canvas

Maak deze eerst vooraleer je aan het NoSql-deel van het examenproject begint

Meer uitleg over hoe MongoDB op te zetten en te gebruiken, vind je in de tutorial op Canvas. Ook het opzetten van sharding en replica sets wordt daarin toegelicht.

Aangezien MongoDB toch een heel ander manier van werken is dan de klassieke RDBMS, moet je eerst deze tutorial maken vooraleer je aan de slag kan met het examenproject.

Véél succes!