

Table of Contents

1. [Introduction](#)

NoSQL

Introductie

De ontwikkeling van relationele databases betekende een geweldige verbetering in het data landschap. Gebaseerd op een simpel wiskundig model, had men een oplossing voor data anomalieën (inconsistente data). Maar door een exponentiële groei van e-commerce en sociale media wordt de focus in databases gelegd op scalability, low cost, flexible en highly available. Deze doelen zijn moeilijker te bereiken met relationele databases, mede door de hoge kosten die deze met zich meebrengen.

Scalability

Om op de gepaste tijdstippen een variërende workload te hebben. Bijvoorbeeld bij een spike in trafiek naar je website kan met extra servers online brengen om de load op te vangen. Bij gebruik van relationele databanken is een dergelijke scale moeilijk te beheren. NoSQL databases zijn zo gedesigned om servers toe te voegen of te verwijderen met een minimale interventie van de administrator.

Cost

De kost van database licenties is steeds een belangrijke factor bij de keuze van technologie. De meeste NoSQL systemen zijn open source.

Flexibility

Database designers moeten bij de start van het project alle tabellen en kolommen reeds geïmplementeerd hebben. NoSQL databases hebben geen vaste structuur, met andere woorden een programma kan dynamisch nieuwe attributen toevoegen zonder het schema aan te passen.

Availability

Je wil nooit dat je website niet beschikbaar is. No SQL databases hebben het voordeel om op verschillende low-cost servers te draaien. Wanneer 1 server faalt, de andere kunnen makkelijk de workload opvangen.

Er zijn 4 types NoSQL databases:

- Key-value
- Document
- Column
- Graph

We behandelen enkel document databases

Document databases Deze bewaren records als documenten. Documenten zijn semi-gestructureerde entiteiten gemodelleerd in JSON of XML formaat (JSON: Javascript Object Notation) (XML: Extensible Markup Language). Bijvoorbeeld: { "firstName": "Tom", "lastName": "Peeters", "beroep": "Lector" } en van de belangrijkste karakteristieken van een document database is dat het geen vast schema (zoals relationele databases) nodig heeft vooraleer je data kan toevoegen. Het toevoegen van een document maakt ook het onderliggende schema! Omdat het geen vast schema heeft hebben ontwikkelaars meer flexibiliteit. Als een gebruiker gedefinieerd als bovenstaand JSON formaat wijzigt in:

```
{ "firstName": "Tom", "LastName": "Peeters", "beroep": "Lector", "woonplaats": "Westerlo", "email":
"tom.peeters@ap.be" }
```

Dan is dit geen enkel probleem, en wordt de tweede record (eigenlijk document genoemd vanaf nu) in de collectie met extra attributen woonplaats en email gepusht. Document databases voorzien ook een API of query taal om de documenten uit de databank te halen.

Bijvoorbeeld:

Stap 1: maak een collectie aan

```
db.createCollection("employees");
```

Stap 2: Voeg data toe:

```
db.employees.insert( { "naam": "peeters", "voornaam": "tom" } )
```

Eigenlijk moet je stap 1 niet uitvoeren, want vanaf dat je data toevoegt, zal de document oriented database automatisch een collectie toevoegen. Zoek operaties gebeuren bijvoorbeeld als volgt:

```
db.employees.find(); db.employees.find( { "voornaam": "tom" } );
```

Verschillen met een relationele database

- Ten eerste hebben we bij een NoSQL database geen vast schema meer nodig.
- Een volgend belangrijk verschil is dat een document  embedded  documenten alsook een lijst van warden (array) kan bevatten.

```
{ "firstName": "Tom", "LastName": "Peeters", "beroep": "Lector", "woonplaats": "Westerlo", "email":
"tom.peeters@ap.be" "VorigeBeroepen": [ { "positie": "developer", "werkgever": "Agfa-Gevaert" }, { "positie": "project
manager", "Werkgever": "Agfa Gevaert" } ] }
```

Het embedden van documenten of lijsten zorgt ervoor dat we geen joins moeten uitvoeren zoals bij relational databases.

- Document databases hebben dus ook de mogelijkheid om data te query-en en te filteren zoals relationele databanken

Wat is een document?

Voor de gemakkelijheid beginnen we met een HTML document. Deze bevatten content en formatting commando's. HTML documenten gebruiken voor gedefinieerde tags om de content te formatteren. Documents in een document database hebben deze voor gedefinieerde beperkingen niet. Ontwikkelaars zijn vrij om de structuur te kiezen. Bijvoorbeeld een customer record in JSON notatie: { "customer_id": 123, "naam": "Peeters", "adres": { "straat": "Ellermanstraat 33", "plaats": "Antwerpen" }, "first_order": "1/9/2015", "last_order": "7/9/2015" }

Structuur van een JSON object

- Data wordt georganiseerd als key-value pairs
- Documenten bevatten name-value pairs, gescheiden door een komma

- Een document start met { en eindigt met }
- Namen zijn strings: bijvoorbeeld: customer_id, adres, ..
- Waarden kunnen numbers, strings, Booleans, arrays, objecten, NULL zijn.
- De waarden van een array worden opgelijst beginnende met [en eindigen met]
- De waarden van een object zijn opnieuw key-value pairs beginnend met { en eindigen met }

Dus een document is een set key-value pairs. De keys worden als strings voorgesteld, terwijl de values basic types kunnen bevatten, of structuren kunnen zijn (arrays, objecten).

Documenten bevatten zowel de structuur als de data. JSON en XML zijn 2 formaten die hiervoor vaak gebruikt worden. Meerdere documenten worden in een collectie gestopt. Collecties zijn dus een lijst van documenten. Document database designers moeten denken aan het zo snel mogelijk kunnen toevoegen, verwijderen, updaten en zoeken naar documenten. Belangrijk te weten is dat documenten binnen een collectie niet pers❖ dezelfde structuur moeten aannemen.



Tips in Collectie design

Collecties bevatten een lijst van documenten. Omdat collecties geen vaste structuur van documenten vereisen kunnen met andere woorden verschillende document types in een collectie zitten. Bijvoorbeeld klant info en server log data kunnen in dezelfde collectie terecht komen. Dit is niet aan te raden! Algemeen beschouwen we dat collecties documenten bewaren over hetzelfde ❖onderwerp❖. Vermijden van abstracte entiteits types! Bijvoorbeeld je wil logging data bewaren over de clicks op je webpagina, alsook algemene server logs. Deze 2 entiteiten hebben een id en timestamp gemeen. Ga deze dan niet als 1 collectie modelleren omwille van het weinige gemeenschappelijke, want als je uiteindelijk de weblog en server log er apart wil uithalen, zul je een extra attribuut ❖type❖ moeten bewaren om de data eruit te filteren. En filteren is dikwijls trager dan het werken met meerdere collecties. Let wel, bovenstaande zijn tips!

Een voorbeeld: we willen bijhouden welke producten klanten hebben besteld. We gaan 1 document aanmaken dat boeken, cd❖s en kleine keukenapparaten kan bevatten. Voorlopig zijn er dus 3 type producten, maar op termijn kan dit uitbreiden. Alle producten hebben:


- Produkt naam
- Beschrijving
- SKU (stock keeping unit)
- Afmetingen
- Gewicht
- Gemiddelde klantenscore
- Prijs Elk type product heeft zijn specifieke attributen: Boek:
- Auteurs naam
- Publisher
- Jaar publicatie
- Aantal pagina❖s CD❖s:
- Artiest naam
- Producer naam
- Aantal tracks
- Tijdsduur Kleine keuken apparaten:
- Kleur
- Spanning (voltage)

Hoe beslissen dat we onze data in 1 of meerdere collecties gaan opslaan? Je vraagt je eerst af wat de noden van de klant zijn. Je klant wil volgende zaken te weten komen:

- Het gemiddeld aantal producten die door elke klant worden gekocht
- Top 20 van de populairste producten
- Gemiddelde prijs verkochte goederen
- Hoeveel producten per type zijn de laatste 30 dagen gekocht? Alle queries gebruiken data van alle product types, behalve de laatste. Dit kan al een indicatie zijn om  n collectie te gebruiken. Een andere indicatie tot gebruik van 1 enkele collectie is de mogelijkheid dat de klant zal groeien en andere product types introduceert.

Operaties op Document databases

De basis operaties zijn net zoals bij relationele databases:

- Insert
- Delete
- Update
- Select Wij gaan MongoDB gebruikt als document database (is voorlopig de meest gebruikte). De taal voor de manipulatie is die voor MongoDB  er is geen uniforme taal.

Toevoegen van documenten in een collectie

`db.employees.insert({"naam":"peeters","voornaam":"tom"})` Je kan ook in bulk data toevoegen: [en] zorgen dat je een array van documenten toevoegd.

Delete documenten

`db.employees.remove()`

Met de remove methode verwijder je documenten. Bovenstaand commando verwijdert alle documenten in de collectie employees, maar de collectie blijft nog steeds bestaan.

Verwijderen van een geselecteerd document:

`db.employees.remove({"naam":"peeters"});`

Deze query verwijdert alle documenten met naam "peeters".

Updaten van documenten

De update method heeft 2 parameters nodig:

- Document query
- Keys en values te updaten:

`db.employees.update({"id":"10"},{$set: {"voornaam":"arno"}});`

Selecteren van data

De find methode wordt gebruikt om documenten van een collectie te selecteren.

```
db.employees.find();
```

```
db.employees.find({"naam":"peeters"})
```

Deze 2 find methodes geven alle keys en values terug in de documenten.

Je kan ook een extra parameter meegeven om te specificeren welke keys je wil laten zien, samen met een 1 om aan te duiden dat deze moet getoond worden.

```
db.employees.find({"naam":"peeters"}, {"voornaam":1});
```

```
db.employees.find({"naam":"peeters"}, {"voornaam":1,"id":1});
```

Je merkt dat MongoDB by default ook een unique identifier weergeeft, zelfs als deze niet gevraagd is.

Complexe queries door conditionele logica

Bijvoorbeeld: `db.employees.find({"id":{"gte":10}})`

De logica dat je kan gebruiken:

- `$lt` : kleiner dan
- `$lte` : kleiner of gelijk aan
- `$gt` : groter dan
- `$gte` : groter of gelijk aan
- `$in` : selecteert documenten waar de waarde van een veld gelijk is aan die gespecificeerd in de array.
Bijvoorbeeld: `db.employees.find({"id":{"$in": [10,11]}})`
- `$or` : voorbeeld: `db.inventory.find({ $or: [{ quantity: { $lt: 20 } }, { price: 10 }] })` : deze query selecteert alle documenten in de inventory collection waarbij de quantity kleiner is dan 20 of de prijs gelijk aan 10
- `$not` : `db.inventory.find({ price: { $not: { $gt: 1.99 } } })` : deze query selecteert alle documenten waarbij de prijs kleiner of gelijk is aan 1.99

MongoDB commando's

Om een database aan te maken gebruik je het use commando, bijvoorbeeld `use news`. Om te kijken welke databanken beschikbaar zijn : `show dbs`. De goede lezer merkt dat onze news database niet beschikbaar is, dit komt omdat deze nog geen collecties bevat. Om een collectie aan te maken:

```
db.createCollection("EenCollection")
```

Basis terminologie

Documenten en collecties zijn de basis data structuren van een document database. Er zit wat analogie in met rijen en kolommen van relationele databases.

Volgende termen zijn belangrijk in ons document model:

- Document
- Collection
- Embedded Document
- Schemaless

- polymorphic scheme

Document

Een document is een set van key-value paren. Een key is een unieke identifier, gebruikt om een waarde op te zoeken.

Key zijn normaal altijd strings, terwijl values verschillende data types kunnen zijn: string, number, array, object.

Arrays zijn handig als je verschillende instanties van een value wil hebben. (In tegenstelling tot relationele databanken, heb je hier geen 2 tabellen, en dus geen join nodig).

```
{ "naam": "peeters", "CursusIds": [123, 456, 589]
}
```

CursusIds is een lijst van cursus Ids. Omdat CursusIds slechts enkel nummers zijn is een array hier aangewezen, maar indien je meer info over een cursus wil bewaren kan je ook met embedded documenten werken.

Bijvoorbeeld:

```
{ "naam": "peeters", "Cursus": [{ "id": 123, "naam": "databases" }, { "id": 456, "naam": "web frameworks" }, ] }
```

Collections

Een collection is een groep van documenten. De documenten binnen een collectie zijn meestal gerelateerd aan elkaar.

Embedded Documents

Een voordeel van document databases is de mogelijkheid om gerelateerde data op een meer flexibelere manier te bewaren dan bij relationele databases.

Indien je een werknemer en een project wil bewaren, zal je in een relationele databank waarschijnlijk 2 tabellen gebruiken, terwijl men in een NoSQL database met embedded documents kan werken. Op die manier kan je "joining" vermijden. Joining verwijst naar een proces waarbij data van de ene tabel gelinkt is met een vreemde sleutel naar een andere tabel.

Een join van 2 grote tabellen kan tijdsintensief zijn, terwijl het gebruik van embedded documenten bij elkaar horende data in 1 document kan bewaard worden. Dus embedded documents zijn documenten binnen in een document.

Schemaless

Voor document databases heb je geen analyst nodig die bijvoorbeeld eerst een ER-schema opstelt om de structuur op te zetten. Een dergelijke structuur noemen we ook een schema. In een relationeel model duidt je dan aan wat de tabellen, kolommen, primaire en vreemde sleutels zijn.

Document databases hebben dit niet nodig, en vandaar zeggen we dat ze schemaless zijn.

Schemaless betekent :

- meer flexibiliteit
- meer verantwoordelijkheid

Meer flexibiliteit

Ontwikkelaars kunnen op elk moment nieuwe key-value paren toevoegen, zelfs na de collectie is aangemaakt. Eens de collectie aangemaakt kan je er variërende documenten naar toe sturen.

Meer verantwoordelijkheid

er bestaan dus geen regels op de database structuur, en dus moet een programmeur een verantwoordelijke rol opnemen als hij iets met de data doet.

Polymorphic Schema

Een document database is polymorfisch omdat de documenten verschillende vormen kan aannemen.

Bijvoorbeeld: { { "a":1, "b":2, "c":3, }, { "b":1, "d":4, "g":5, }, { "a":8, "d":2, "f":3, "g":3, } }

Data modelleren

Als je voor een document database een design moet maken, vraag je je eerst af welke queries te er op wil loslaten. Terwijl bij relationele databases je gaat afvragen welke entiteiten en de relatie ten opzichte van elkaar allemaal nodig zijn.

Nadat je bij je relationeel model de entiteiten hebt bepaald, ga je aan normalisatie (en eventueel denormalisatie) doen.

Normalizeren is het organiseren van je data in tabellen om potentiële data anomalieën tegen te gaan (inconsistente data). Bij normalisatie vermindert de redundante (overtollige) data in de database. Om te normalizeren gebruiken we enkele "normalisatie" regels (1e, 2e, 3e,...).

Wanneer we bij document databases meerdere collecties gebruiken zeggen we dat deze genormaliseerd is. Genormaliseerde documenten betekent dat we een referentie leggen naar andere documenten.

Bijvoorbeeld: Server en ServerLog Data:

Log Data: { { "ID": 123, "Event Data": "bla bla", "ServerId": 123 } { "ID": 456, "Event Data": "bla bla", "ServerId": 222 } }

Server Data: { { "Id": 123, "info": "test" }, { "Id": 456, "info": "test" } }

}

Normalisatie help dus om data anomalieën te vermijden, maar kan performantie problemen veroorzaken! Als je data in meerdere tabellen moet opzoeken (joining). Met andere woorden het design van een database is altijd een trade-off tussen een hoge genormaliseerde database zonder redundante data en een gededenormaliseerde database. Dus denormalisatie doet een undo van de normalisatie regels.

Waarom aan denormalizatie doen?

Denormalizatie veroorzaakt data anomalie♦n, heeft meer geheugenruimte nodig (dubbele data bewaren), maar betekent een veel betere prestatie.

Document Database Design

Document database designers bewaren gerelateerde data in hetzelfde document. (Er zal dikwijls een trade-off gemaakt moeten worden tussen prestatie en anomalie♦n.) Document database designers trachten steeds data anomalie♦n te vermijden, maar zijn bereid om verantwoordelijkheid te tonen ten opzichte van scalability en flexibility. Bijvoorbeeld: indien er redundante kopies van klantadressen in de database worden bewaard, kan de programmeur een update methode maken die alle kopies van adressen update. Op die manier kunnen anomalie♦n vermeden worden, maar betekenen dus een grotere verantwoordelijkheid voor development.

Modelleer 1-N relatie

Er zijn 3 gedachtegangen om dit te verwezenlijken.

In mongoDB kan je een een array (van subdocumenten) embedden in een parent document. Maar als je een schema designed moet je aan het volgende denken:

- Wat is de kardinaliteit van de relatie? Maar veel genuanceerder: is het een 1 op weinig, of 1 op veel .

Een 1 op weinig relatie

Dit kan bijvoorbeeld de adressen van een persoon zijn. Een juiste use case om dit in het document persoon te embedden:

```
db.person.findOne() { name: 'Kate Monster', ssn: '123-456-7890', addresses : [ { street: '123 Sesame St', city: 'Anytown', cc: 'USA' }, { street: '123 Avenue Q', city: 'New York', cc: 'USA' } ] }
```

De voordelen hiervan is dat je geen aparte query moet maken om de details (adressen) te kennen. Het nadeel is dat je de adressen niet als standalone kan raadplegen. Dit nadeel speelt parten bij bijvoorbeeld: elke persoon heeft een aantal taken toegewezen gekregen. Embedding de taken binnen een persoon, zal de query ♦toon alle taken voor morgen♦ veel moeilijker maken

Een op veel relatie

Bijvoorbeeld : elk product bestaat uit onderdelen (niet meer dan enkele duizenden). Dit is een juiste use case voor ♦referencing♦. De Object Ids van de onderdelen stop je in een array bij het product document. En dus heeft elk onderdeel zijn apart document:

```
db.parts.findOne() { _id : ObjectId('AAAA'), partno : '123-aff-456', name : '#4 grommet', qty: 94, cost: 0.94, price: 3.99 }
```

Elk product heeft ook zijn eigen document die een array bevat van objectIds referenties

```
db.products.findOne() { name : 'left-handed smoke shifter', manufacturer : 'Acme Corp', catalog_number: 1234, parts : [ // array of references to Part documents ObjectId('AAAA'), // reference to the #4 grommet above ObjectId('F17C'), // reference to a different Part ObjectId('D2AA'), // etc ] }
```

Om alle onderdelen van een product eruit te halen doe je: // Fetch the Product document identified by this catalog number

```
product = db.products.findOne({catalog_number: 1234}); // Fetch all the Parts that are linked to this Product
product_parts = db.parts.find({_id: { $in : product.parts } }).toArray();
```

Het voordeel in dit design is dat elk onderdeel een stand-alone document is, dus makkelijk in te zoeken en up te daten. Een trade off is dat je een tweede query nodig hebt om de onderdelen van een bepaald product eruit te halen.

Veel-op-veel relatie

Veel op veel relaties gebruiken 2 collecties. Een voor elke entiteit, en elke collectie onderhoudt zijn lijst van bijvoorbeeld studenten en cursussen

```
Courses: { { "CourseId": "132", "title": "databases", "Enrolled Students": [ 1, 2, 3, 4, 5 ]
},
{ "CourseId": "456", "title": "web technology", "Enrolled Students": [ 1, 3, 4, 5, 6 ]
}, { "CourseId": "789", "title": "Web frameworks", "Enrolled Students": [ 1, 2, 3, 4, 5 ]
} }
Students: { { "StudentId": 1, "name": "peeters" "Courses": [ 132, 456, 789 ] },
{ "StudentId": 2, "name": "vandeperre" "Courses": [132,789 ] },
}
```

Hier beperken we duplicate data door te refereren naar identifiers. Maar we moeten er wel voor zorgen dat beide entiteiten steeds correct geupdate worden.