

Databanken 3

De opkomst van NoSQL

Academiejaar 2018 - 2019

Jan Van Overveldt

Tom De Reys



Er is meer in het IT-leven dan relationele databanken, en dat willen we in de laatste hoofdstuk van Databanken 3 bekijken.

We gaan dieper in op één van de populairste "alternatieve" vormen, namelijk NoSql databanken.

Inhoud



1 de alleenheerschappij van de relationele databanken

2 wat is Big Data?

3 het verhaal van performance & availability

NoSql is meer dan alleen maar een ander type van databank. Om deze juist te kunnen plaatsen en weten waarom en hoe je deze kan gebruiken, moeten we eerst een aantal algemene concepten bespreken. Pas als je die kent, kan je eigenlijk op de juiste manier omgaan met NoSql databanken.

Ten eerste gaan we het even hebben over de relationele databanken, meer bepaald waarom we naar alternatieven op zoek gaan en ons niet bij deze oude en vertrouwde vorm blijven. Concepten zoals ACID gaan we bespreken en in vraag stellen.

Dan gaan we een introductie over Big Data geven. Niet zo zeer technisch (daarvoor bestaat er een keuzevak), maar vooral over hoe en waarom dat nu ontstaan is. En waarom de klassieke relationele databanken geen goed idee zijn om te gebruiken bij Big Data.

Wat we zeker ook niet uit het oog mogen verliezen, is de problematiek rond performance en high-availability. We overlopen in dit hoofdstuk de verschillende mogelijkheden om deze te verbeteren.

Als laatste hoofdstuk geven we meer uitleg over NoSql databases op zich en bekijken we 4 verschillende soorten en bespreken hun belangrijkste kenmerken. Van twee van deze vormen gaan we in de volgende les meer concreet mee aan de slag.



Als eerste gaan we even inzoomen over het monopoly dat relationele databanken hebben in het huidige IT-landschap. Desondanks dat er verschillende grote spelers op de markt zijn, blijven deze type databanken veruit het meeste vertegenwoordigd. Hoe dit komt en waarom we misschien naar alternatieven moeten kijken, bespreken we in dit hoofdstuk.

Bij de start van een IT project....veel vragen

Platform (Java, .NET,...)?

Application server?

Front-end framework?

Back-end framework?

Mobiele strategie (native, web,...)?

Methodologie (Scrum,...)?

Database vendor (Oracle, MySQL,...)?

Infrastructuur (Linux, Unix, Windows,...)

Cloud?

三



Bij de start van een IT project....2 constanten

Type programmeertaal? **Object oriented**

Type database? **Relationeel**

Bij de opstart van een nieuw IT-software ontwikkelingsproject moeten er véél vragen beantwoord worden. Uiteraard moet er een bepaalde business nood zijn waarvoor nieuwe software (een deel van) de oplossing is. Daarnaast zijn voldoende betrokkenheid & motivatie van alle partijen en niet onbelangrijk de nodige financiële middelen noodzakelijk.

Er zijn bij de opstart van een IT project ook een resem technische en organisatorische vragen die een antwoord nodig hebben vooraleer men kan starten met de ontwikkeling. Welke methodologie gaan we gebruiken is misschien wel één van de belangrijkste. Gaan we voor (echte) SCRUM, of proberen we het nog eens op de goede oude waterval-manier - die met een gemiddelde slaagkans van 25% zacht uitgedrukt een risico is?

Daarnaast moeten er ook de juiste technologieën en producten gekozen worden die gebruikt zullen worden tijdens de implementatie fase. Geen eenvoudige en vooral een uiterst belangrijke taak. Als tijdens de implementatie blijkt dat je een verkeerde keuze gemaakt hebt, is de impact - en bijhorende kosten - vaak zo hoog dat het niet mogelijk is om deze keuze aan te passen. Je gaat niet halverwege een project van 5 manmaanden beslissen om niet meer langer in Java maar met Ruby verder te ontwikkelen, omdat je immers dan hetgeen wat al reeds ontwikkelt is grotendeels in de vuilbak moet gooien...

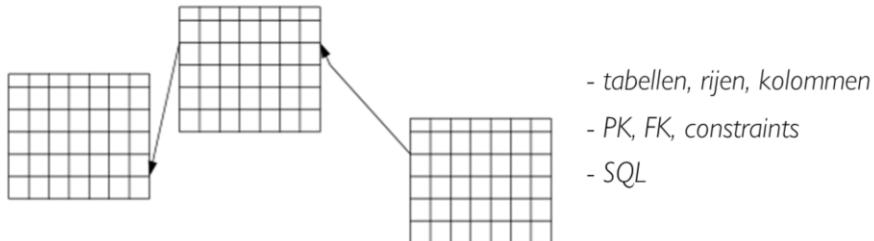
Die keuzes - waarvan voorbeelden vermeld staan in deze slide - moeten welfoordacht genomen worden op basis van de verschillende requirements (wat voor type applicatie wordt gevraagd, welke uptime wordt verwacht ...). Dit doen we onder andere op basis onderzoek van (nieuwe) technologieën, uitwerken van Proof-Of-Concepts (POC) en ervaringen met bepaalde producten.

Er zijn echter twee vragen die in zo goed als alle projecten steeds hetzelfde antwoord krijgen. Ten eerste, wat voor type programmeertaal hebben we nodig? De keuze valt zo goed als altijd op een object georiënteerde taal zoals Java of C#. Alternatieven (functionele talen, scripting talen zoals Phyton/Ruby/...) wordt zelden of eigenlijk bijna nooit overwogen, hoewel deze misschien beter geschikt kunnen zijn voor (een deel van) de implementatie.

Wat zelfs amper of nooit ter discussie staat, is welk type database er gebruikt moet worden. Zonder nadenken wordt er nog steeds vaak vanuit gegaan dat er een relationele databank moet gebruikt worden. Naar alternatieve type (No-SQL) databases denkt men niet meteen aan, desondanks dat deze in bepaalde case's een hele hoop voordelen hebben.

Relationele databanken

= de klassieke aanpak voor het omgaan met gegevens



Genormaliseerde relationele database voor **OLTP**

Eventueel bijkomend ETL naar een **gedenormaliseerd** relationeel data warehouse voor **OLAP**

Even opfrissen uit het Databanken 1...

De relationele databank is gebaseerd op tabellen, bestaande uit kolommen en bevat 0 of meerdere rijen. Een tabel heeft een vast aantal kolommen waarvan elke kolom de mogelijke waarde definieert (welk datatype, hoelang, verplicht of niet...). Die tabel kan 0 of meerdere rijen bevatten, één rij is een reeks van samenhorende waarden, namelijk voor elke kolom één waarde die voldoet aan de definitie van de kolom waartoe deze behoort.

Een tabel heeft normaal ook een Primary Key, wat wil zeggen dat de waarde van één (of meerdere) kolom(men) een rij uniek identificeert. Daarnaast kan je met behulp van constraints beperkingen opleggen op waarden van één (of meerdere) kolommen. Ook kan je verschillende tabellen naar elkaar laten verwijzen met foreign keys.

Alle relationele databanken kan je aanspreken met SQL statements (DML).

Tabellen in een OLTP worden ontworpen volgens de normalisatie-regels om redundantie van gegevens te vermijden.

Bij tabellen in een OLAP daarentegen wordt er gebruik gemaakt van denormalisatie-technieken en dus redundantie terug toegevoegd om hoofdzakelijk performantieredenen. Dat hebben we ook gedaan in het hoofdstuk over Data Warehousing.

Relationele databanken

The GOOD

- Veel types databases & tools beschikbaar
- Veel knowhow (developers, maintenance,...)
- Krachtige en flexibele standaard zoektaal (**SQL**)
- 'Makkelijk' te integreren in applicaties (*drivers, API's, ORM's*)
- **ACID**
 - Atomicity
 - Consistency
 - Isolation
 - Durability



Om misverstanden te vermijden: relationele databanken hebben zeker hun nut en een reeks voordelen! Anders zouden ze uiteraard ook niet zo massaal gebruikt blijven worden.

Doordat deze zoveel gebruikt worden, zijn er in de loop der jaren ook verschillende types (merken) van databases ontwikkeld door grote softwarebedrijven, met een resem van tools die je dan kan gebruiken om deze te beheren en/of aan te spreken. Er zijn enerzijds tools die door de ontwikkelaars van de databanken zelf op de markt zijn, maar ook onafhankelijke bedrijven en zelfs open-source projecten hebben tools op de markt gebracht die één of meerdere DBMS kunnen aanspreken. Daarnaast zijn er nog talrijke open-source projecten die ook zulke tools aanbieden.

Er is niet alleen een ruime keuze in tools beschikbaar, maar ook in knowhow. Er zijn immers héél véél mensen die in aanraking komen met deze type databanken waardoor je op het internet véél informatie vindt. Niet alleen over de werking van een DBMS, maar ook om deze te kunnen configureren en/of programmeren.

Verder is ook SQL een zéér krachtige en wijdverspreide taal om zoek-, DML en DDL opdrachten uit te voeren. Iemand die de basis SQL syntax kent, kan deze op alle relationele databanken gebruiken.

Bovendien zijn er ook véél drivers, API's en frameworks (zoals ORM's) beschikbaar om vanuit software een relationele databank aan te spreken. Elke volwaardige programmeertaal biedt wel mogelijkheden om vanuit code een databank te kunnen aanspreken. Bovendien zijn er voor de meest gangbare programmeertalen frameworks beschikbaar die vanuit code een

databank aanspreken vrij gemakkelijk maakt.

Een laatste - niet onbelangrijk - voordeel is dat deze databanken allemaal het ACID principe ondersteunen: Atomair, Consistent, Geïsoleerd & Duurzaam (zie volgende slide)

Relationele databanken

Database transacties ACID

A TOMICITY

Alles lukt of alles faalt (**commit-rollback**)

C ONSISTENCY

Database blijft consistent voor/na transactie (bv. FK constraints)

I SOLATION

Isolation level bepaalt mate van beïnvloeding tussen parallelle transacties op elkaar (bv. serialisable, read uncommitted, dirty reads,...)

D URABILITY

Resultaat van transactie is permanent (m.a.w. opgeslagen op disk)

ACID = BETROUWBAARHEID

bv. 1000€ van een rekening boeken = garanties dat dit lukt



ACID is het kernwoord om te beschrijven dat een databank en meer specifiek transacties betrouwbaar moet zijn. Elk van die letters staat voor een specifieke eigenschap:

- Atomair
 - Alle acties uitgevoerd binnen een transactie moeten in hun geheel slagen of falen.
 - Gedeeltelijk slagen van een transactie kan NIET
- Consistent
 - Data in de databank moet consistent blijven na de transactie, net zoals ook voor de transactie is. Er moet dus voldaan blijven worden aan de aanwezige constraints (zoals bv. foreign keys, unique constraints, ...)
- Isolatie
 - De DML-acties binnen een transactie moeten geïsoleerd kunnen uitgevoerd worden, dus los van andere operaties. Een belangrijk aspect hiervan is de 'Isolation Level' van een transactie.
 - Isolation Level geeft aan wat er moet gebeuren met de records die gequeried en/of gewijzigd zijn tijdens een transactie. Moeten die records gelocked worden zolang de transactie loopt, wordt er toegestaan dat er wel lees-operaties mogen uitgevoerd worden? Zo ja, ziet de lees-operatie enkel de oude gecommitteerde data of ook zogenaamde dirty reads (=data die nog niet gecommit is en dus mogelijk nooit bewaard zal worden)...
- Duurzaam
 - Eens de transactie gecommit zijn de wijzigingen "permanent" bewaard

Relationele databanken

The BAD



- **Impedance mismatch** met OO domein
 - Nood aan complexe tools zoals Hibernate en Entity Framework
- Wijzigingen in het **schema** hebben veel impact
 - Conversiescripts, migrations,...
- Complexe horizontale **scalability** (zie verder)
 - De meeste producten (Oracle, MySQL,..) zijn initieel bedoeld voor gebruik op een single server model
 - Voorzieningen voor **clustering** pas later toegevoegd
 - Data model is minder geschikt voor replicatie (aggregates, zie verder)
 - Licentiekost per node kan hoog zijn (bv. Oracle)

Naast de belangrijke voordelen van relationele databanken, zijn er ook een aantal nadelen waar je zeker rekening mee moet houden.

De methodiek en structuren die je bouwt in code volgens de OO principes, matchen niet met hoe een relationele databank gestructureerd moet worden. Enkele voorbeelden zijn:

- Het concept van primaire sleutels (iets dat je record uniek identificeert) bestaat niet in OO, maar op zich kan dat wel een meerwaarde geven om altijd een unieke identifier te hebben.
- Foreign keys is ook typisch iets voor een relationele databanken maar wat eigenlijk niet bestaat volgens de OO principes. In OO termen spreken we over associaties en maken we in klasse A een extra veld aan van het type 'klasse B'. Dat veld kan ofwel een object bevatten, ofwel een lijst van die objecten naargelang het type associaties. Om zulke relaties te kunnen bewaren in een relationele databank, moeten we dus foreign key velden gaan toevoegen.
Als we dan verder kijken naar n-op-n relaties, moeten we in de databank zelfs een aparte tabel aanmaken.
- Overerving is ook iets dat niet bestaat op een relationele databank. Ook daar moeten we bepaalde technieken gebruiken om dit te implementeren

Er zijn nog voorbeelden, maar bovenstaande zijn de meest voorkomende.

Het is dus niet eenvoudig de vertaalslag te maken tussen de relationele databank en de bijhorende objecten. Daarom zijn er ORM-frameworks zoals Hibernate en Entity Framework ontstaan om het de gebruikers (ontwikkelaars) makkelijker te maken. Maar we weten al van de eerste lesweek dat dit ook niet zonder risico is.

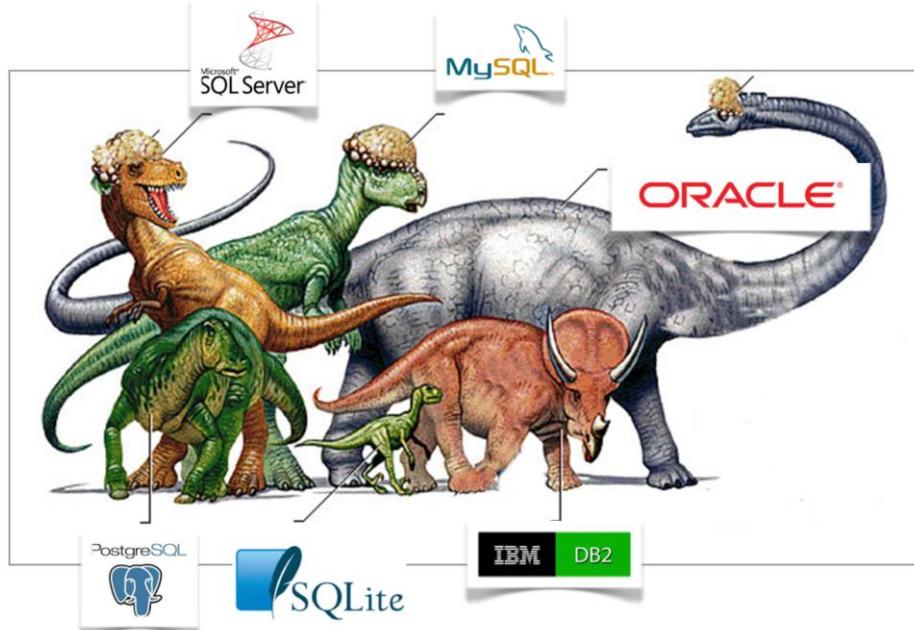
Een ander nadeel is dat wijzigingen aan het schema (kolom erbij, van naam veranderen, nieuwe constraints, ...) vaak v  l impact hebben op de reeds bestaande data. Vaak moeten er conversiescripts geschreven worden om de data te kunnen verrijken of opsplitsen en je zal merken dat er altijd wel een aantal "lastige" gevallen tussen zullen zitten. De complexiteit kan snel groot worden waardoor in een aantal gevallen er ook door de business zelf beslissingen moeten genomen worden i.v.m. de juistheid van de data. Ook als er datamigratie-scripts aanwezig zijn, of (ETL-)scripts om de data over te zetten naar een OLAP omgeving, moeten die allemaal aangepast worden.

Historisch gezien zijn relationele databanken ook ontworpen als "single-server" applicaties. Een databank draait op één machine waarnaar iedereen connecteert. Uiteraard is dat de laatste jaren helemaal niet meer het geval. Door de exponentiële groei aan data en de steeds groter worden vereisten rond high-availability, heeft men gezocht naar technieken om één applicatie tegelijkertijd te laten draaien op meerdere servers - en dan nog liefst zo transparant mogelijk zodanig dat de gebruikers van die applicatie niet hoeven weten op welke server(s) ze connecteren. Het beste voorbeeld hiervan is Cloud-computing, wat de laatste jaren echt doorgebroken is.

De meeste relationele DBMS hebben een voorziening voor clustering (beide servers nemen input aan) en replicatie (één server neemt input aan, andere server neemt een 'kopie' van de input aan), maar die zijn pas later aan het initiële ontwerp toegevoegd. Het gevolg is, is dat deze niet altijd even eenvoudig op te zetten zijn er daardoor er naargelang het DBMS een aantal restricties zijn. Zo worden oa. autonummering velden (zoals Identity op SQL Server) niet ondersteund bij clustering.

Bovendien zien de softwarehuizen die relationele DBMS bouwen dit vaak als een soort van 'cash-cow'. Zo heeft Oracle bijvoorbeeld een licentiemodel waarvoor je betaalt per node waarop je databank draait. Ook SQL Server heeft zulke licenties naast de per-processor formule. Al deze type licenties zijn z  r prijzig.

Relationele databanken - de grote spelers (db-engines.com)



De grootste spelers op de markt van Relationele Databanken Management Systemen (RDBMS) zijn in volgorde van populariteit zijn:

- Oracle
- MySQL (open-source)
- Microsoft SQL Server

Van de kleinere spelers zijn PostgreSQL, DB2 van IBM, & SQLite de belangrijkste namen, maar deze kom je sowieso een heel stuk minder tegen dan de grote drie spelers. Ook Microsoft Access staat nog steeds in de top 10, maar gezien de beperkte functionaliteiten van deze applicatie beschouwen echte IT-professionals dit niet als een professionele databank.

Het is nog zo dat de grote drie veruit het merendeel van de markt domineren, maar de laatste jaren komen hoe langer hoe meer de niet-relationele databanken opzetten. Zo is op dit moment (oktober 2018) MongoDB - een NoSql (document store) databank de 5^e populairste databank (<http://db-engines.com/en/ranking>) en staat deze vlak achter de grote 3 RDMS

Het aandeel van NoSql databanken groeit ook elk jaar en hun populariteit ook. Zo staan Cassandra, Redis en ElasticSearch ook nog in de top 10.

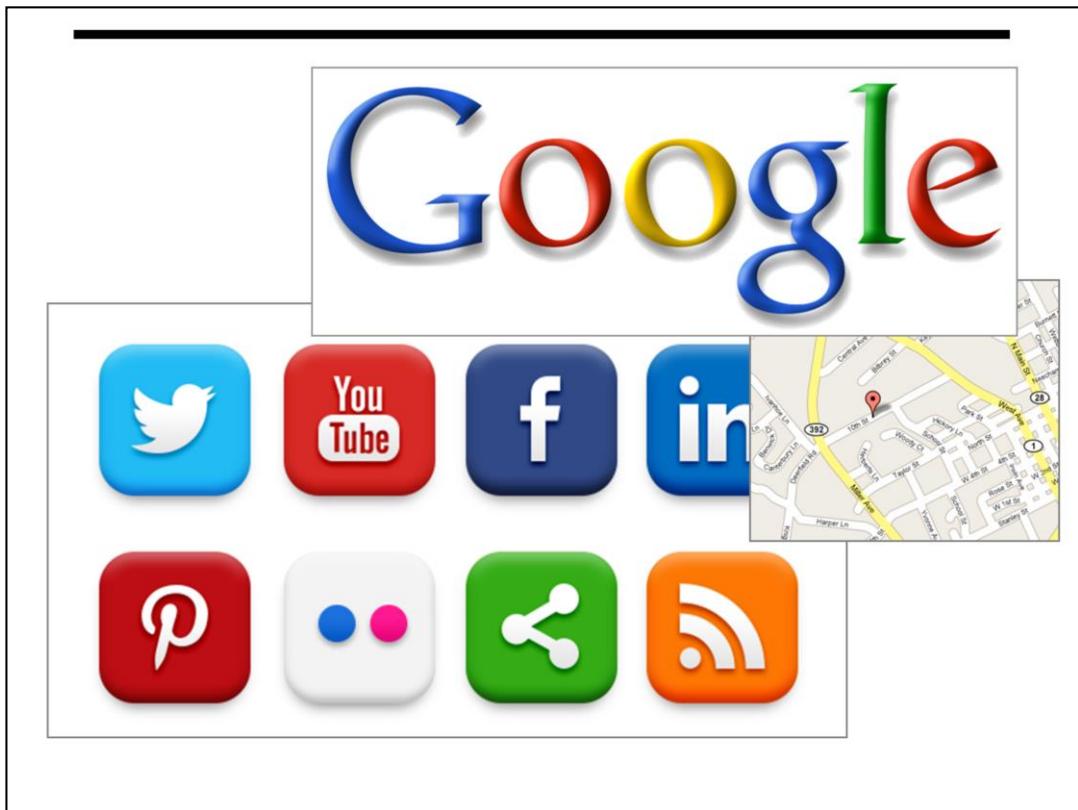


Men botst hoe langer hoe meer tegen de limieten van relationele databanken aan. Er is steeds meer en meer data beschikbaar waardoor de volumes aan data zéér groot wordt. Er zijn continu overal stromen ("streams") van data beschikbaar en die zijn niet altijd in een relationele vorm (bv. zoekopdrachten in een search-engine).

2

BIG DATA

Zoals we in de vorige slides ook aanhalen, lopen we tegen de limieten van relationele databanken. Data komt hoe langer hoe meer in zéér grote hoeveelheden en in verschillende vormen op ons af en daarom ontstond de nood om het verwerken van die data anders te bekijken. Klassieke relationele databanken volstaan niet meer om die data op te slaan en te verwerken. Om dit toch te kunnen realiseren zijn een heleboel technologieën ontstaan onder de noemer van "BigData".

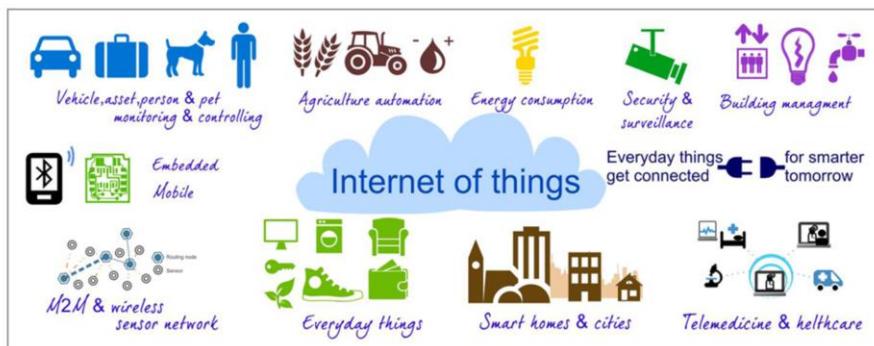


Google is het typevoorbeeld van een bedrijf dat over een gigantische hoeveelheid data beschikt in verschillende vormen.

Zij worden daar trouwens ook steenrijk van. Dankzij een goed uitgebouwde strategie rond "Ads" stroomt het geld daar binnen. Ze bepalen van elke gebruiker - naargelang hun zoekopdrachten, verplaatsingen, fotos... - in welk type producten de gebruikers mogelijk geïnteresseerd zijn, verkopen "reclameruimte" op pagina's van hun honderden miljoenen gebruikers van oa. YouTube en kunnen zo gerichte reclame-advertenties plaatsen. Uiteraard zijn ze niet de enige die dat doen. Je moet bijvoorbeeld op sites zoals bol.com of coolblue.be zoeken naar een bepaald product... Dagen nadien kom je nog reclame over die producten of gelijkaardige producten tegen op diverse andere sites zoals Facebook, krantensites,

Uiteraard is niet alleen Google de trotse bezitter van een enorme hoeveelheid data... Denk ook maar aan Twitter, LinkedIn, Facebook, ... allemaal hebben ze véél data in handen en willen ze die maar al te graag zo snel mogelijk kunnen verwerken en analyseren om zo je profiel te kunnen opstellen en gerichte 'informatie' te geven.

Internet Of Things (IoT)



Het is trouwens niet alleen ons gedrag op onze webbrowsers achter onze pc die als bron kan dienen. Tegenwoordig heeft bijna alles een WiFi of andere netwerkverbinding en dus een IP-adres. Van een pc, smartphone, game console, smartwatches, digitale televisie, ... tot zelfs je auto of je garagepoort toe. En alles wat een IP-adres heeft is een mogelijke bron van data. Die data komt ook vaak in diverse niet-relationele vormen zoals XML, JSON, CSV, ...

Dat fenomeen noemen ze ook wel eens het "Internet of Things". Internet is geëvolueerd en reikt véél verder dan enkel maar de servers, pc's en laptops zoals enkele jaren terug...

Bovendien zijn er ook hoe langer hoe meer sectoren die op deze kar springen, want het biedt natuurlijk ook enorme mogelijkheden. Het traceren van bagage via sensoren, aansturen van diverse installaties (sproei-installaties in de landbouw, airco, verwarming, ...) afhankelijk van het weer en zelfs de weersvoorspellingen, monitoren van energieconsumpties en bijsturen waar nodig, camerabewaking van je zaak steeds kunnen bekijken - ook al zit je aan de andere kant van de wereld-, ... Overall komen er hoe langer hoe meer toepassingen kijken dit gegevens van verschillende bronnen en in verschillende vormen snel moeten kunnen lezen, verwerken en bepaalde acties ondernemen.

Één van de sectoren die in de nabije toekomst ongetwijfeld groter en groter zal worden, is de gezondheidszorg. De combinatie van personeelstekort en de steeds "ouder" wordende bevolking zal verdere automatisering in die sector veroorzaken en dus ook de nood aan de bijhorende dataverwerking.

We hebben dus ons online gedrag waarmee we 'bewust' data ter beschikking stellen van bedrijven zoals Google, Facebook (Microsoft, ...), maar we doen dit ook vaak héél onbewust.

Wie connecteert er in een winkel wel eens niet met het WiFi netwerk van die winkel? Dat is vaak gratis, maar bedrijven zoals Ikea houden daar enorme hoeveelheden data van bij om je gedrag te analyseren, gerichter reclame te kunnen sturen, ...

In de marge hiervan: dit is ook de reden waarom er tegenwoordig zoveel producten en operating systemen "gratis" worden gemaakt voor de gebruikers. Denk maar aan Android, iOS en Windows 10. Deze houden een hele hoop gegevens over je bij, zoals je surfgedrag, posts op sociale media, locatiegegevens, ... Ze verwerken deze en gaan die dan verder verkopen

Big data



<http://readwrite.com/2011/05/17/how-big-is-a-yottabyte-infographic>

De eerste echte harde schijf (IBM, 1958) had een capaciteit van 5 Megabyte, wat in die tijd al gigantisch groot was. Die harde schijf nam dan ook de volle 1,5 vierkante meter in beslag...

Tegenwoordig heb je al SD-kaartjes ter grootte van 1,5 vierkante centimeter waarop je een terabyte of meer op kan bewaren. Ofte 10.000 keer kleiner in size, maar > 200.000 meer opslag. Al een geluk, want beeld je anders maar eens in hoe groot je laptop zou geweest zijn als je een harde schijf van 500 Gb zou hebben...

De hoeveelheid aan data - en de plaats nodig om deze te bewaren - heeft men eens in bovenstaande vergelijkingstabel opgeliist. Dit om een idee te geven wat voor een grote hoeveelheid data een YottaByte is...

Een goede reden waarom men probeert de fysieke opslagcapaciteiten steeds meer en meer compacter te maken...

Enkele weetjes (de vraag is hoe betrouwbaar dit is, maar wel leuk om te bekijken):
<http://www.live-counter.com/how-big-is-the-internet/>

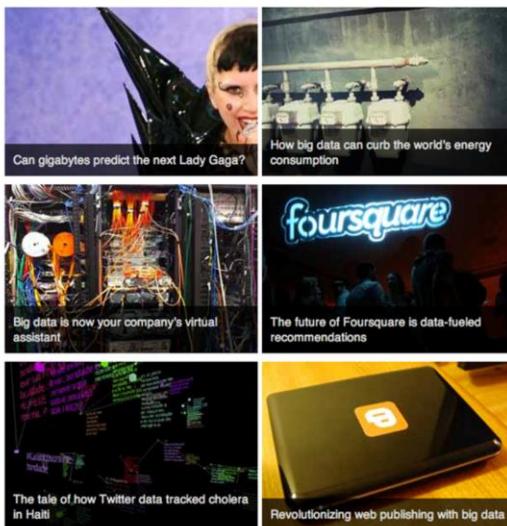
Cisco schatte dat de globale internettraffiek in 2019 2 zettabytes per jaar zou zijn.

Geschattte grootte van internet: 13,7 Zetabytes

Geschattte grootte van internet in 2030, 1 yottabyte

Big data

<http://gigaom.com/2012/03/11/10-ways-big-data-is-changing-everything/3/>
<https://www.targit.com/en/blog/2014/11/9-ways-big-data-analytics-changing-the-world>
<http://www.zdnet.com/article/shipping-to-data-the-case-of-amazon-prime-day/>



In deze slide en op de opgegeven link kan je enkele voorbeelden terugvinden waar BigData voor gebruikt kan worden of zelfs al wordt:

- Voorspellingen van 'luchtige' onderwerpen zoals 'welke factoren heeft Lady Gaga gemaakt tot wie ze is en wie gaat dan de volgende worden' tot 'hoe kunnen door het verzamelen van diverse gegevens de wereldwijde energieconsumptie verminderen?'
- Ondersteuning voor bedrijven door betere simulaties van diverse scenario's (marketing campagnes, product placements, ...) en gerichtere reclamecampagnes
- In kaart brengen van bepaalde 'rampen' of een uitbraak van een epidemie door het monitoren van sociale media (zoals bv. de Cholera uitbraak in Haïti na een grote aardbeving aldaar)
- Inbouwen van sensoren in de uitrusting van sporters om hun prestaties te kunnen opvolgen, blessure-risico's analyseren, veldbezettingen real-time bekijken, Wordt al op grote schaal in de Premier League gebruikt (<http://www.wired.co.uk/article/the-winning-formula>)
- Telenet maakt ook gebruik van verschillende databronnen om je gepersonaliseerde reclame te tonen op je Digibox/Digicorder. Op basis van je kijk- en surfgedrag (url's ontleden, zoekoperaties bijhouden & analyseren) krijg je in het pauze-menu van je toestel gepersonaliseerde reclame
- Big Data wordt ook hoe langer hoe meer gekoppeld met Machine Learning en Artificiële Intelligentie (AI) gebruikt. Er wordt aan de hand van diverse complexe algoritmes gezocht naar patronen in de verzamelde data om zo voorspellingen te kunnen maken en/of AI algoritmes te verbeteren
- Ook in de logistiek en transportsector - waaronder scheepvaart - wordt hoe langer hoe meer BigData gebruikt.

In dit artikel <https://www.smartdatacollective.com/how-amazon-shaped-big-data-landscape/> kan je een mooi voorbeeld terugvinden van de intrede van BigData en NoSql in de logistiek

Big Data

Big data: "Informatie is de nieuwe olie"

Big data, big deal?

De **hoeveelheid data** die wordt geproduceerd groeit **exponentieel** door de beschikbaarheid van informatie zoals GPS, RFID, beacons, sociale media, click streams,...

The internet of things...alles krijgt een IP adres en is een mogelijke bron van data

Deze data wordt **continue geproduceerd** in een **eindeloze stroom**

Het kan bijzonder interessant zijn voor een bedrijf om deze datasets in **REAL TIME** te genereren, analyseren en **combineren**, vaak in combinatie met Machine Learning algoritmes



We zijn vandaag omringt door een massa aan datastromen rondom ons heen. De uitdaging is dus hoe we deze zo snel mogelijk kunnen vast krijgen en verwerken zodanig we die data kunnen analyseren. En dan nog het liefst real-time. Een datawarehouse heeft net daar zijn beperkingen. Een real-time datawarehouse zou eigenlijk is eigenlijk naar performance toe niet haalbaar.

Big Data - 3 V's

Men spreekt van **de 3 V's**

Volume grote hoeveelheden data

Velocity die snel wijzigen, snel verwerkt moeten worden

Variety en afkomstig zijn van verschillende bronnen

Deze data is vaak **niet gestructureerd**

(bv. system logs, twitter berichten,...)

Relaxing ACID

Consistentie is vaak **minder cruciaal**

(bv. ik zie dat Obama 1.203.645 facebook friends heeft en iemand ziet op hetzelfde moment dat Obama er 1.203.698 heeft....**SO WHAT.....**)

Durability is vaak **minder cruciaal**

(bv. in memory database gebruiken voor data die enkel in real-time moet bekijken worden)

Een vaste definitie van BigData is er niet, maar waar iedereen het over eens is, is dat je over BigData mag spreken als deze voldoet aan de 3 V's:

- Volume: je moet grote hoeveelheden data verwerken
- Velocity: de te verwerken data wijzigt snel waardoor deze ook snel verwerkt moet kunnen worden
- Variety: de te verwerken data is afkomstig van verschillende bronnen en is bovendien vaak ook niet gestructureerd (logs, twitter berichten, zoekopdrachten, video's...).

Een ander kenmerk is dat het ACID- principe - waar relationele databanken op gebouwd zijn - minder cruciaal is. Dit is ook vaak het gevolg van net die enorme hoeveel data. De case vermeld in bovenstaande slide is daar een mooi voorbeeld van. Bekijk dus altijd kritisch: is het echt wel van levensbelang dat iedereen hetzelfde (aggregatie)resultaat of zelfs dezelfde records ziet op hetzelfde moment?

Duurzaamheid is ook al 'minder' belangrijk omwille van het hoge tempo waaraan de gegevens kunnen wijzigen. Ben je wel geïnteresseerd in de situatie zoals ze een uur geleden was? Of enkel en alleen maar in de real-time gegevens?

Deze voorwaarden om over Big Data te spreken, gaat dus zeker en vast niet alleen over analyse (BI) systemen. Big Data kan dus gerust ook gebruikt worden als een operationele systeem!

Big Data

Allright...

- Ongestructureerde data**
- Minder nood aan ACID**



Eeuuh...relationele databases...zijn goed in...

- Gestructureerde data (schema!)**
- Full proof ACID**

Samengevat:

De evolutie naar BigData die volop bezig is, gaat over ongestructureerde data die uit verschillende bronnen komen. Daarnaast is de noodzaak aan ACID juist aan het afnemen. Twee van de belangrijkste voordelen van een RDBMS zijn dus niet relevant voor veel van de use cases van deze tijd.

Big Data

Allright...

- **Grote volumes aan data**
- **Die snel wijzigen**



Eeuuh...relationele databases...zijn **niet** goed in...

- **Simple scalability**

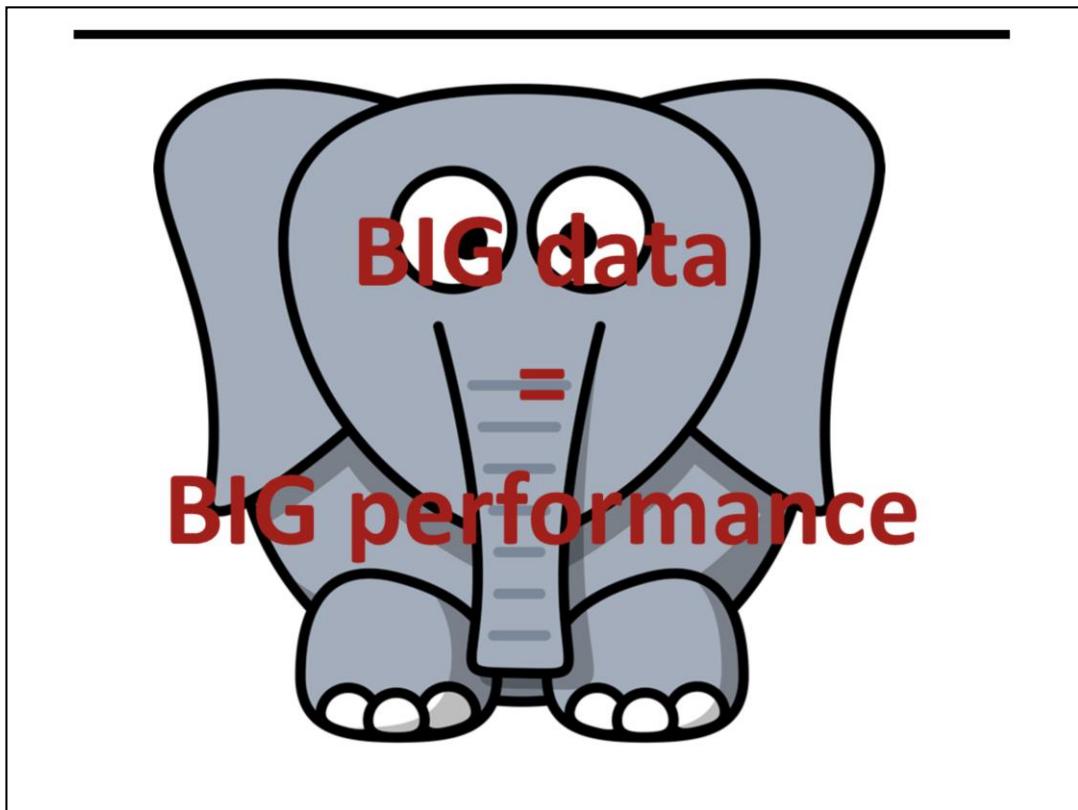
Conclusie...

BIG DATA vraagt vaak om een **andere database technologie** dan de klassieke RDMBS

Wat bij BigData net wel cruciaal is, zijn de grote volumes aan data die verwerkt moet worden en dan nog eens vrij snel liefst - omdat de data op zich snel kan wijzigen. Dat vereist dus een grote schaalbaarheid van je systeem zodanig dat je de gegevens en de verwerking daarvan via parallelisatie (load balancing, clustering, ...) snel kan uitvoeren.

En laat dat nu net een punt zijn waarin klassieke relationele DBMS systemen niet zo sterk in zijn.

Conclusie: omgaan met BigData heeft andere technologieën nodig dan de klassieke RDBMS. Dat is ook de reden waarom de niet-relationele DBMS systemen de laatste jaren aan een opmars bezig zijn en de evolutie daarvan niet zal stilvallen. Het concept van NoSql databanken bestaat al lang en heeft al enkele hypotheses gekend, maar was toen meer uit 'theoretische' noodzaak ontstaan dan praktisch. Daarom braken deze ook niet echt door. Echter is het IT-landschap nu zodanig veranderd waardoor er wel degelijk een vraag vanuit de markt is naar dit type databanken.

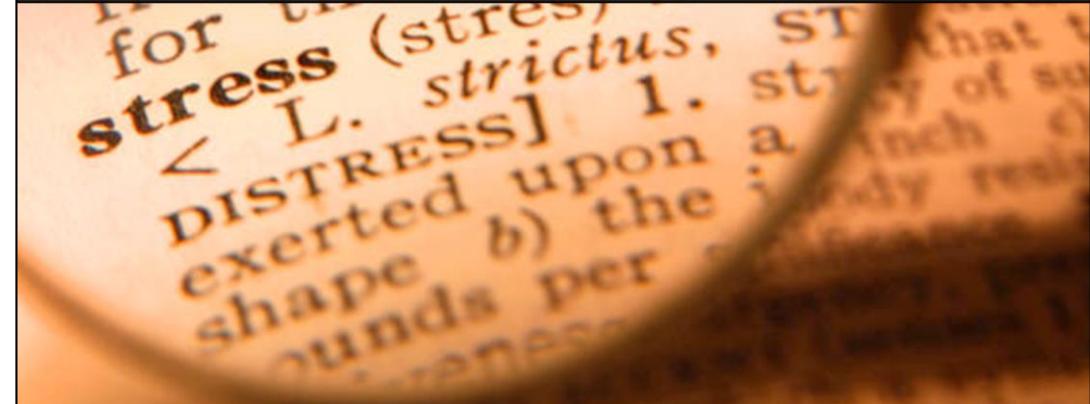


BIG Data vraagt dus ook een BIG performance! En om die big-performance te kunnen bieden, moeten we 'af' van het traditionele ACID en kijken naar DMBS die sneller werken en dus meer geschikt zijn hiervoor.

Let op: ACID wordt hier niet afgeschilderd als oud en onbruikbaar. In vél situaties MOET je een DBMS gebruiken dat full-proof ACID is, maar concepten zoals transacties die noodgedwongen locks op bepaalde records leggen zijn "nefast" naar performantie toe.

3

The story of **performance** and **availability**



In dit hoofdstuk bespreken we welke optimalisatie-technieken er gebruikt worden om de performantie van NoSql databanken zo hoog mogelijk te krijgen. Om een goed beeld daarover te krijgen, gaan we ook wat uitweiden over scaling en availability technieken zodanig dat we daar een goed inzicht in krijgen.

Performance

Hoe kan je een *database sneller maken?*
(meer data verwerken, snellere queries, meer gebruikers bedienen,...)



Met al hetgeen we al gezien en gelezen hebben de afgelopen jaren, weken en uren: hoe zouden we de performantie van de databank sneller kunnen maken?

Performance

Hoe kan je een database sneller maken?



Database tuning

indexes, materialised views, ...

Relaxing ACID

minder isolation (i.e. meer parallelisatie),

minder durability (i.e. minder flushes naar disk)

minder consistency checks (FK constraints, unique constraints weghalen)

Denormalisation

bv. Fact & Dimensions in een Data Warehouse

Scaling

vertical & horizontal **scalability**

Op vlak van database tuning kan je op verschillende manieren de performantie van je databank verhogen, tenminste als je deze technieken juist toepast. Enkele voorbeelden hiervan zijn:

- Het gebruik van indexen waarmee je select-queries kan versnellen. Indexen kunnen gebruikt worden indien je zoekt met een WHERE-clausule
- Materialised views is een andere manier om zoekoperaties te versnellen. Deze hebben we reeds besproken in het hoofdstuk rond datawarehouses

Daarnaast hebben we het ook reeds gehad over het concept achter "Relaxing ACID". Is consistentie echt zo belangrijk in je systeem? Zijn constraints (Foreign Keys, unique constraints, ...) en dergelijke wel altijd nodig? Is de duurzaamheid zo belangrijk of kan het kwaad dat het even duurt vooraleer de gegevens op disk bewaard worden (minder disk I/O)? En door isolation (locks) te vermijden, kunnen er meer operaties tegelijkertijd gebeuren.

Het toepassen van denormalisatie-technieken en dus redundantie toevoegen in je database, helpt vaak ook de performantie. Je moet dan minder Foreign Keys aanmaken wat als gevolg heeft dat er minder joins uitgevoerd moeten worden en ook de persistentie sneller maakt. Onder andere het stermodel waar je met feiten en dimensies werkt, leunt op deze principes.

Scaling kan de performantie ook positief beïnvloeden. Het gaat dan enerzijds over hardware scaling, maar ook over data over verschillende servers heen te verspreiden, zelfs data van binnen dezelfde tabel (partitioning), zodanig dat je disk I/O - wat nog altijd de meest

vertragende factor is - ook verdeeld wordt.

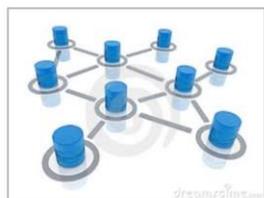
Performance

Een systeem dat **scalable** is kan makkelijk aangepast worden om een hogere **performantie** te bereiken

Vertical scaling (scale-up)
Server upgrade (memory, disk,...)



Horizontal scaling (scale-out)
Data verspreiden over een **cluster** van meerdere servers
(**nodes**)



Een scalable systeem is een systeem dat je eenvoudig kan aanpassen om hogere performantie te bekomen. Enerzijds spreekt met over 'vertical scaling' of 'scale-up', als men eigenlijk de hardware gaat upgraden. Dat kan gaan over meer memory, snellere disks, ... Op zich blijft de databank draaien op dezelfde machine, enkel onder de motorkap gaat men die hardware verbeteren.

Daarnaast heb je ook 'horizontal scaling', ofte 'scale-out'. In plaats van alle data te verzamelen op één dezelfde server, gaan we deze verspreiden over verschillende server heen. Men spreekt dan over een cluster, wat neerkomt op een aantal samenwerkende database servers. Elk van die servers is dan een node in die clusters.

Enerzijds kan je clustering gebruiken in de context van replicatie. Insert, updates & deletes van een bepaalde tabel gebeuren op node A, terwijl de lees-acties op node B lopen. Zo kan elke server op zich afzonderlijk de query parsen en uitvoeren.

Je kan ook kiezen om de verschillende tabellen te verdelen over verschillende nodes (sharding), of zelfs partities van tabellen verdelen.

Availability

Hoe kan je een database meer beschikbaar maken?

Super de lux hardware

(RAID disk,...)



Clustering

Data verspreiden over meerdere servers (**nodes**)



Er zijn twee veel gebruikte manier om databases meer beschikbaar te maken.

Je kan dit hardware-matig bewerkstelligen door zware hardware te voorzien met allemaal back-up systemen in. RAID is daar een mooi voorbeeld van. Als er één HDD uitvalt, bevat de andere(n) ook nog alle gegevens. Ook netwerkkaarten, RAM, ... worden allemaal hardware-matig 'dubbel' voorzien. En dan mogen we UPS nog niet vergeten voor als de stroom uitvalt.

Daarnaast ga je met clustering niet alleen de performantie verhogen, maar je kan er ook voor kiezen om alle data te 'dupliceren' (repliceren) over verschillende servers heen. Als er dan één bepaalde server/node uitvalt, zijn er nog anderen waarnaar geconnecteerd kan worden.

Availability

Availability %	Downtime per year	Downtime per month*	Downtime per week
90%	36.5 days	72 hours	16.8 hours
95%	18.25 days	36 hours	8.4 hours
98%	7.30 days	14.4 hours	3.36 hours
99%	3.65 days	7.20 hours	1.68 hours
99.5%	1.83 days	3.60 hours	50.4 minutes
99.8%	17.52 hours	86.23 minutes	20.16 minutes
99.9% ("three nines")	8.76 hours	43.2 minutes	10.1 minutes
99.95%	4.38 hours	21.56 minutes	5.04 minutes
99.99% ("four nines")	52.6 minutes	4.32 minutes	1.01 minutes
99.999% ("five nines")	5.26 minutes	25.9 seconds	6.05 seconds
99.9999% ("six nines")	31.5 seconds	2.59 seconds	0.605 seconds



(Bron: wikipedia)

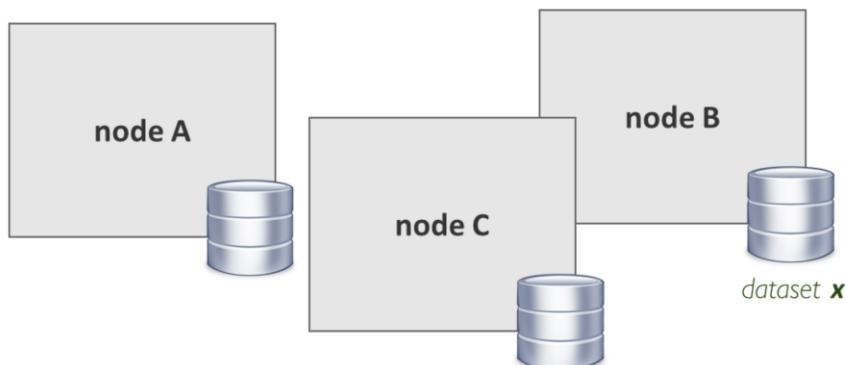
Meestal wordt de availability van applicaties uitgedrukt in %. Deze tabel geeft mee wat deze percentages eigenlijk betekenen in concreto. Op het eerste zicht lijkt 90% toch wel een behoorlijk percentage, maar als je het aantal uren downtime dan in tijdsduur uitdrukt wordt dat toch ineens wel veel...

Om zowel je hardware alsook software aan die vereisten te laten voldoen moet je vaak véél investeren. Mede daarom zijn Cloud-platvormen ontstaan om je te helpen - tegen een variabele vergoeding - aan die vereisten te voldoen. De vendors van Clouds (Amazon AWS, Microsoft Azure, IBM Bluemix, Oracle Cloud, ...) zorgen zelf dat het OS een optimale uptime heeft, jij moet 'enkel' je eigen software robuust genoeg maken. Maar ook daarvoor zijn er truckjes voor om die downtime zoveel mogelijk te beperken en zelfs te vermijden.

Een ander aspect dat hoe langer hoe belangrijker wordt, is de high-availability. Iedereen verwacht dat alle systemen continu beschikbaar zijn. Wanneer sites als bijvoorbeeld Facebook eens een half uur niet beschikbaar zijn, is dat wereldnieuws.

Als IT'er is dat echter een enorme uitdaging. Je moet er dus voor zorgen dat al je systemen blijven draaien ook al gaat er achterliggende hardware stuk. Zelfs bij updates van de software mag je systeem niet uitvallen, laat staan bij updates van je OS.

"The attack of the cluster"



node = hardware waarop een instantie van database server draait
(bv. Oracle, SQL Server, MongoDB,...)

= disk waarop de business data staat

dataset x = datastructuur op disk, gemanaged door database software
(bv. table in tablespace,...)

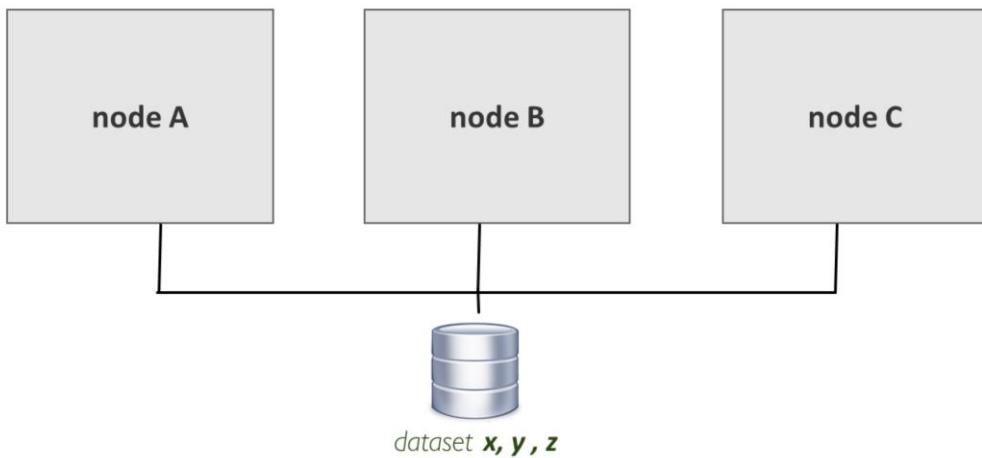
In de volgende slides gaan we iets dieper in rond het concept van clusters, vooral welke mogelijkheden dit biedt naar optimalisaties en high-availability toe. Allereerst geven we eerst een overzicht van gangbare terminologie als men spreekt over clusters en clustering.

- Node: een node is de hardware waarop de DBMS geïnstalleerd staat
- Disk: I/O (opslaglocatie), meestal HDD
- DataSet: structuur van de data zoals deze op de disk bewaard wordt
- Instance: (staat niet op de tekening). Je kan meerdere installaties van een DBMS op één dezelfde server installeren. Dat kan zowel meerdere installaties van verschillende DBMS zijn, maar ook meerdere installaties van dezelfde DBMS. Zulk een installatie noemt men ook wel eens een instance. Wordt minder gebruikt bij Clustering.

Clustering is de algemene term die gebruikt wordt als je verschillende servers - die op een aparte OS en eigenlijk best ook op aparte hardware - toch samen laten werken. In productieomgevingen wordt dit vaak toegepast voor databases, vooral in het kader van performantie optimalisaties en/of high-availability.

In de volgende slides bespreken we enkele mogelijke scenario's bij clustering. Weet wel dat niet alle DBMS al deze scenario's ondersteunen.

Shared storage

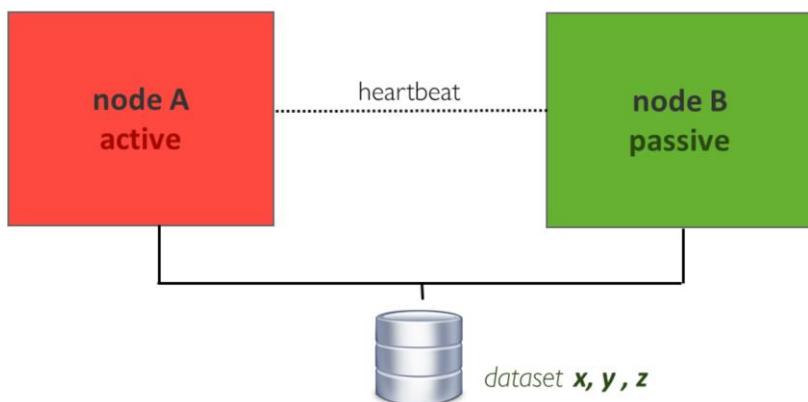


Elke node heeft **toegang tot dezelfde data** (netwerk drive,...). De data zelf wordt dus niet gedistribueerd maar gedeeld ('shared').

Shared storage wil zeggen dat er meerdere servers beschikbaar zijn om de processing uit te voeren maar dat alle datasets centraal op één plaats worden bewaard (op bijvoorbeeld een NAS, SAN).

Het verwerken van de queries wordt dus versneld, maar disk I/O blijft de bottleneck naar performantie toe. Ook als deze gesharede datastore uitvalt, kan geen enkele node meer verder.

Voorbeeld - Local fail-over cluster



Slechts 1 server is actief, de andere is (hot) standby.
Passieve wordt actief zodra de actieve uitvalt. Detectiemechanisme nodig (bv. heartbeat ping lijn tussen de servers of een lock op de shared data)

Availability

Als actieve uitvalt kan passieve overnemen. De shared disk blijft wel een **single point of failure...**

Een veel voorkomende toepassing van clustering is fail-over. Dit houdt in dat er één actieve node is en één (of meerdere) passieve nodes.

Alle wijzigingen en queries worden door de active node uitgevoerd.

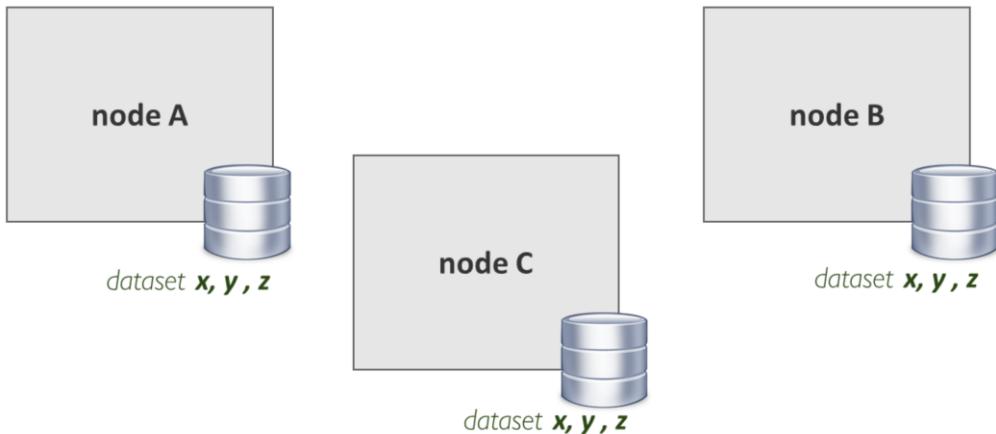
Beide nodes staan ook in connectie met elkaar: de passieve node polled regelmatig bij de active node of deze nog online staat.

Stel dat om één of andere reden de active node uitvalt (stroompanne, HW-panne, reboot, ...) dan neemt de passive node het (automatisch) over. De clients merken er niets van, ook de ontwikkelaars of DBA hoeven hier niets extra voor te doen.

In dit scenario wordt dus alle data shared bijgehouden, maar is er maar één node tegelijkertijd actief.

Hierbij zorgen we voor een betere beschikbaarheid omdat er twee servers klaar staan om het werk te doen. Aangezien de data echter op een centrale plaats staat is dit echter een single point of failure. Stel dat de netwerkconnectie naar de NAS verbroken is dan kan geen van de twee servers zijn taken uitvoeren.

Replicatie - shared-nothing ("load-balancing" cluster)



Elke node heeft een volledige **kopie** van de data.

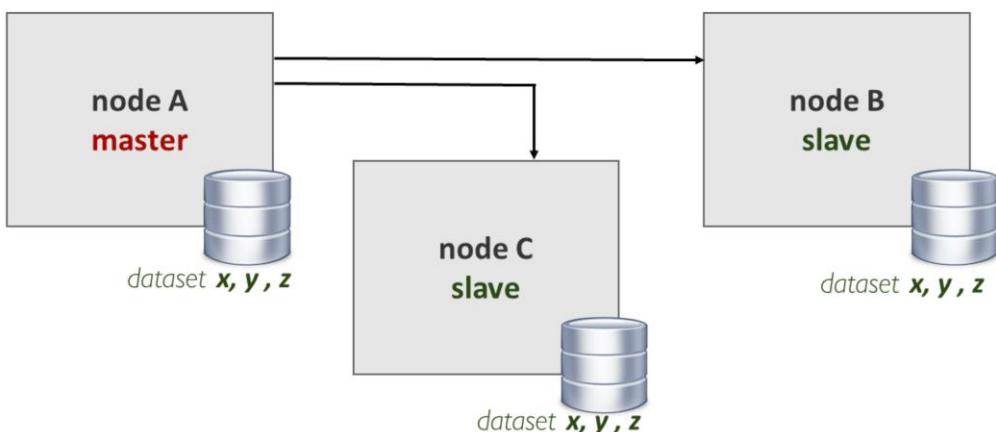
Nood aan een **replicatie** mechanisme tussen de systemen:
kopiëren van data van het ene systeem naar het andere.

Replicatie is een techniek die wel al gezien hebben in week 2 rond optimalisaties. Dit houdt in dat elke node een volledige kopie heeft van alle data. Er is dus een mechanisme nodig tussen de verschillende nodes om die data onderling te kopiëren.

Er zijn twee mechanisms die daarvoor gebruikt worden. Meestal gebeurd dat volgens de 'transaction logs', zijnde dat voor elke afgelopen transactie dat stukje log wordt doorgestuurd naar de andere nodes en de wijzigingen die beschreven zijn in de transactionlog daar ook uitgevoerd worden.

Een alternatief scenario is dat sommige DMBS-systemen je de keuze geven om ipv het resultaat (transaction log) je ook de queries op zich kan doorsturen tussen de verschillende nodes. Voordeel is dat er minder trafiek is tussen de servers, nadeel is dat die queries op zich op elke node terug geparsed moeten worden. Niet elke DMBS ondersteunt deze laatste optie.

Voorbeeld - Master-slave scale-out



Updates enkel toegelaten op de master Replicatie van gegevens van de master naar de slaves. Slaves worden enkel gebruikt voor lees operaties (**read replicas**)

Performance



Zeer effectief voor toepassingen met **veel select** en **weinig insert/update** operaties

Availability



Geldt enkel voor slaves of mechanisme nodig om slave tot master te promoten

Bij replicatie wordt meestal het "master-slave" scale-out model gebruikt. Er is één master waarop updates van de data worden toegelaten, op alle slaves worden er géén updates toegelaten maar dus enkel lees-operaties.

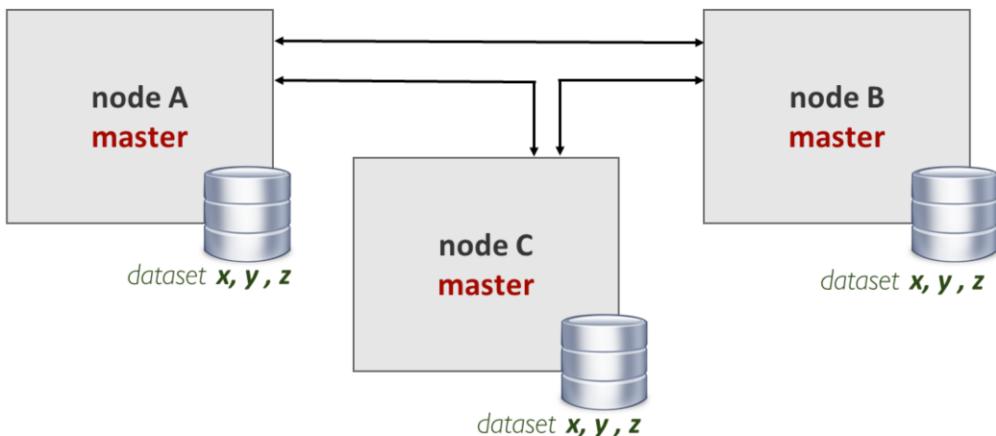
Als je een systeem hebt waar er véél selects gebeuren in verhouding tot de insert/updaten, dan win je hierbij véél performantie.

Het kan soms even duren vooraleer alle updates uitgevoerd zijn (lock waits, disk I/O, ...) en als er dan ondertussen nog zware select-statements geprocessed moet worden die op zich ook disk I/O vragen gaat alles hoe duren. Daarom worden de updates enerzijds - en de select-statements anderzijds - op verschillende nodes uitgevoerd. Op de master-omgeving gebeuren de wijzigingen aan de data, die dan op zich worden gekopieerd via het replicatie-mechanisme naar de slaves.

Als je dan de select-queries op de slaves uitvoert, verdeel je dus de ballast maar krijg je toch altijd alle data te zien.

In het fail-over scenario gebeurt er eigenlijk een automatische fail-over van node A naar node B indien de eerste node uitvalt. Bij replicatie is dit NIET. Als de master uitvalt, is het 'gedaan' met spelen... Je moet een mechanisme opzetten om zelf een slave dan te promoveren naar master.

Voorbeeld - Multiple master



Updates zijn op alle systemen toegelaten (**masterless**)

Duplex replicatie van gegevens tussen alle nodes

Performance



Enkel indien **consistentie** wordt verlaagd
(anders wachten tot update gekopieerd naar andere nodes)

Availability



Wanneer node uitvalt kan andere overnemen

Je kan ook replicatie opzetten met meerdere masters. Dit houdt in dat de replicatie niet alleen loopt tussen master en slave, maar dat meerdere nodes zich als master gedragen. De data wordt dus tussen alle servers gekopieerd.

Net zoals bij fail-over clusters, heb je dus altijd een node die het kan overnemen als een andere node uitvalt. Het verschil echter is dat in dit systeem, je meerdere "actieve" nodes hebt en dus de load verdeeld kan worden.

Deze opstelling heeft echter wel enkele nadelen. Er kan een klein tijdsverschil zitten op de replicatie van de data tussen de verschillende nodes. Als er nieuwe data op node A geïnserted wordt, is er een kleine vertraging tijdens de replicatie-fase. Daardoor kan het zijn dat in die tussenfase een query uitgevoerd op node B waarin deze nieuwe data opvraagt, deze toch niet gevonden wordt.

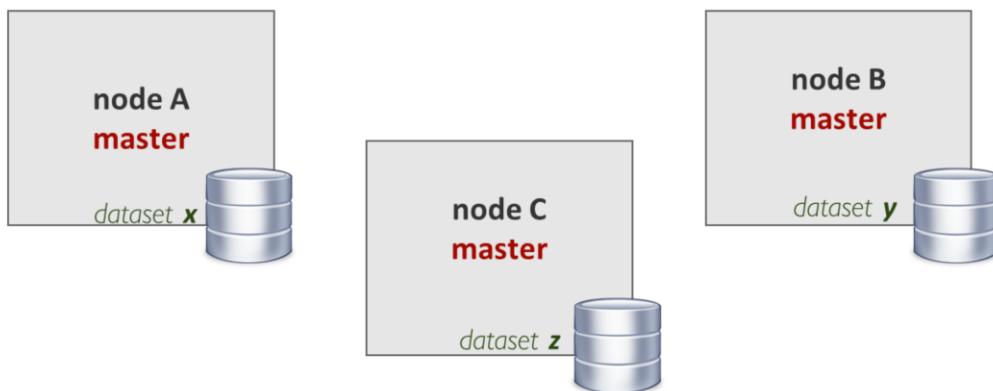
Dit effect op de consistentie moeten we langs de andere kant ook niet overdrijven, want gaat het vaak over luttele seconden of zelfs minder. De vraag is of dit een probleem kan vormen of niet voor je systeem.

Duplex replicatie wil zeggen dat de replicatie in alle richtingen mogelijk is. Dit geeft ook extra complexiteit. Stel dat met een verschil van 2 milliseconden eenzelfde rij wordt aangepast op twee masters. Wat wordt dan de uiteindelijke waarde: Een merge van beiden of gewoon de laatste?

Ook de load op het netwerk veroorzaakt door dit replicatie-proces mag je niet onderschatten. Per replicatie-fase moet er voor elke node de te repliceren data/actie

doorgestuurd worden. Zeker bij systemen met een aanzienlijke belasting, mag je de impact hiervan niet onderschatten. Je kan dan wel performantie winnen door de load van alle queries te verspreiden over de diverse nodes, maar je moet oppassen dat de netwerktrafiek naar de nodes toe geen bottleneck begint te vormen. Dat de replicatie vertraging oploopt, heeft alleen maar impact op de consistentie, maar als ook de binnenkomende requests van de clients vertraging oplopen, doe je misschien alle winst qua performantie teniet.

Sharding



Data wordt **verdeeld** over verschillende nodes

Elke node heeft toegang tot een **deel van de data (= een shard)**.

Performance

Vraag voor dataset z kan door node C afgehandeld worden parallel met vraag voor dataset x door node A.

Availability

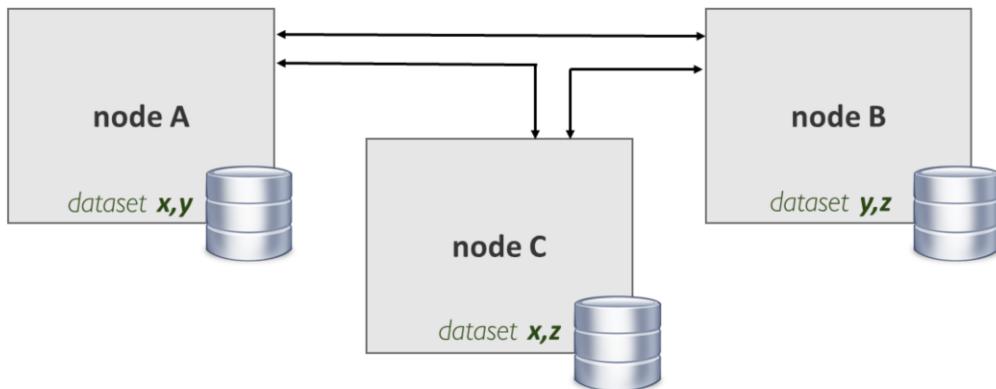
Als node uitvalt is deel van de data niet beschikbaar

Een ander concept waar we het nog niet hebben over gehad, is data sharding. Dat houdt in dat we de data gaan verdelen over verschillende nodes en dus bijgevolg elke node 'slechts' een subset van de data bevat. Zulk een subset noemt men dan een 'shard'.

Grote voordeel hiervan is dat er parallel verschillende datasets opgevraagd kunnen worden tegelijkertijd. In plaats van dat de node query per query alle requests gaat afwerken, kan je nu bij elke node die je nodig hebt een (deel van de) query laten uitvoeren en krijg je dus sneller resultaat.

Naar high-availability toe hebben we echter met deze techniek weinig verbetering, eigenlijk zelfs een achteruitgang. Als een node uitvalt, blijven de andere nodes actief maar je kan wel niet meer aan de data aan die op de uitgevallen node stond.

Sharding en replicatie



Combinatie van technieken

Performance



Availability



Complex om op te zetten

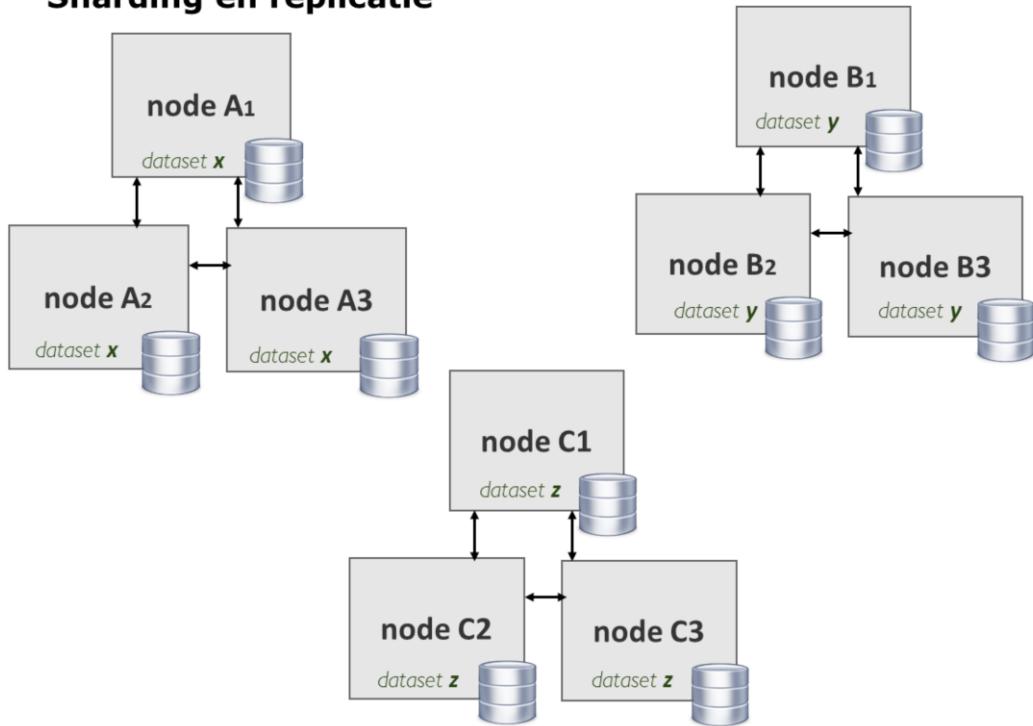


Als je 'the best of all worlds' wil, moet je dus eigenlijk een combinatie maken van replicatie met multiple masters en data sharding. Dit houdt in dat je over de nodes heen de data in shards verdeelt, en dat je die shards ook repliceert over 2 (of meer) nodes.

Zo kan de load over de verschillende nodes heen verdeeld worden maar beperk je de replicatie-overhead. Bovendien heb je ook een high-availability oplossing. Als er één node uitvalt, staan de shards die daar opstonden ook nog op minstens één andere node.

Het enige nadeel van deze oplossing, is dat deze complex is om op te zetten. Je moet immers zowel de replicatie met multiple masters opzetten alsook de opsplitsing doen in data shards. Die shards moet je eigenlijk ook nog best zodanig verdelen over de nodes heen dat een bepaalde combinatie van shards maar maximum op één node voorkomt.

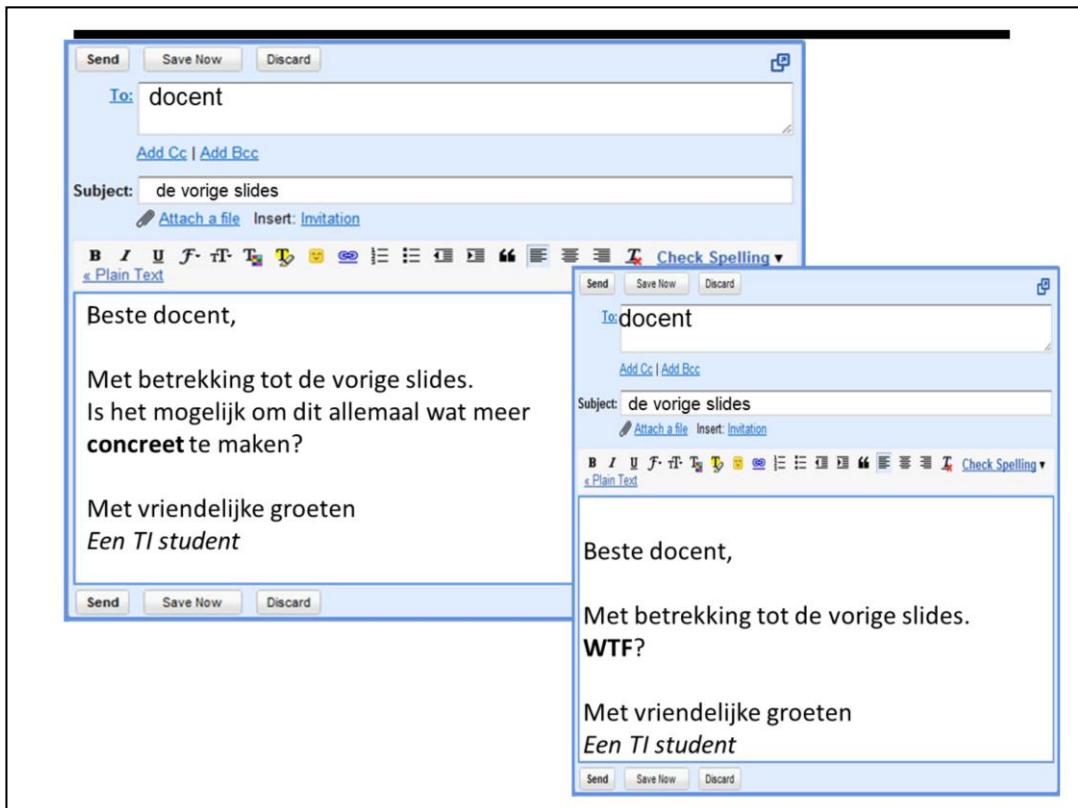
Sharding en replicatie



Een alternatieve manier om sharding en replicatie te combineren, is om subclusters van shards te maken. Als je dan 3 shards hebt, kan je bijvoorbeeld elke shard op 3 nodes bijhouden. In dat geval, heb je 3×3 (9) nodes nodig.

Enerzijds heb je dan de high-availability doordat elke shard op 3 nodes beschikbaar en er zelfs 2 nodes mogen uitvallen.

Daarnaast is deze oplossing naar performantie ook voordelig. Je beperkt zoveel mogelijk het replicatieproces van een shard en zelfs bij een uitval van een node kan je nog de load verdelen over de twee resterende nodes.



Allemaal mooie theoretische voorbeelden, maar hoe zit dat nu in de praktijk? Welke scenario's worden vaak gebruikt? Ondersteunen alle DBMS-systemen bovenstaande scenario's, of is dit louter theorie?

Relational Database sharding

EXAMPLE

- Rijen van een tabel verspreiden over verschillende database servers

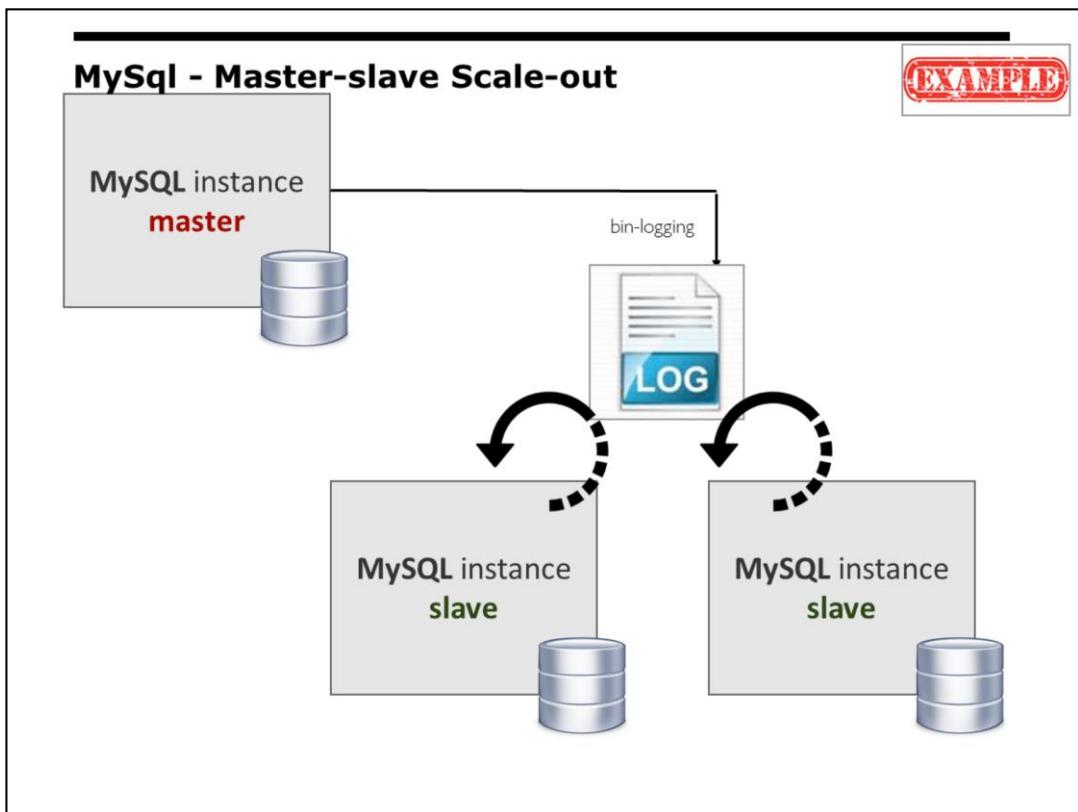


<https://www.cubrid.org/manual/en/9.1.0/shard.html>

Tegenwoordig ondersteunen de meeste gangbare relationele DBMS systemen al een heel aantal scenario's. Je kan dan bepaalde tabellen uit je databank aanmaken in afzonderlijke databanken, die dan weer draaien op afzonderlijke nodes.

Maar omdat de opzet van zulke klassieke RDBMS daar niet op voorzien is, merk je dat de complexiteit om bijvoorbeeld sharding op te zetten hoog is en bovendien een aantal beperkingen heeft. Zo zijn er bij sommige RDMBS systemen restricties op het datatype van de primary key velden (bv. auto-numbering / sequences worden niet toegelaten).

Een goede implementatie en uitleg rond het sharden van data kan je vinden bij Cubrid (zie ook de link op de slide: <https://www.cubrid.org/manual/en/9.1.0/shard.html>).



In MySQL kan je een master-slave scale-out moeten opzetten. Daar gaan we door middel van replicatie vanuit de master ook een slave opzetten.

Deze slave kan je vervolgens gebruiken om SELECT-queries op uit te voeren, terwijl de master alle insert/update/delete queries krijgt.

Bij MySQL, als je werkt met InnoDB als Storage Engine, ga je meestal via de 'bin-logging' de replicatie laten uitvoeren. Dit is eigenlijk een vorm van transaction log waarin alle (gecommitte) wijzigingen in worden bijgehouden. Op geregelde tijdstippen gaan de slaves kijken bij de master of er sinds de laatste keer dat ze bij de master gekeken hebben, er nog nieuwe gegevens in de 'bin-logging' zijn toegekomen.

Ook SQL Server heeft een soortgelijke manier van werken (zie ook hoofdstuk Optimalisatie).

Oracle RAC

EXAMPLE



Geavanceerde technologie, waarbij een Oracle server over een uitbreidbare pool van servers wordt gelegd.

Wel gebaseerd op een *shared storage* model.

<http://www.oracle.com/us/products/database/options/real-application-clusters/overview/index.html>

Ook Oracle heeft een "product" bovenop hun RDBMS waarmee je met behulp van een 'cluster server' verschillende oracle nodes in een pool kan groeperen en beheren.

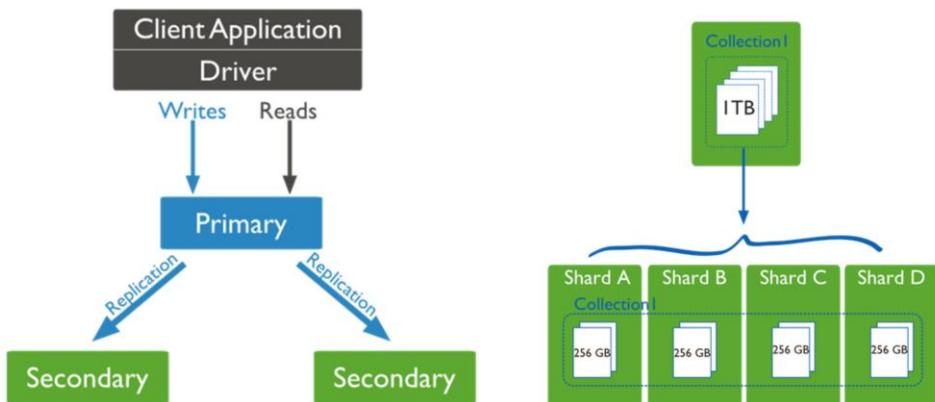
Is wel redelijk prijzig, om en bij de € 20.000 per processor, maar dat is helaas typisch met zulke producten.

De cluster opzet is ook niet ideaal want de data moet nog centraal worden opgeslagen en is dus het single point of failure.

Dit illustreert dat de grote fabrikanten van RDBMS producten door hun jarenlange dominantie zich lang in een veilige positie hebben gevoeld en nu eigenlijk geen goed antwoord kunnen bieden op de huidige noden.

MongoDB sharding en replication

EXAMPLE



ZIE VERDER

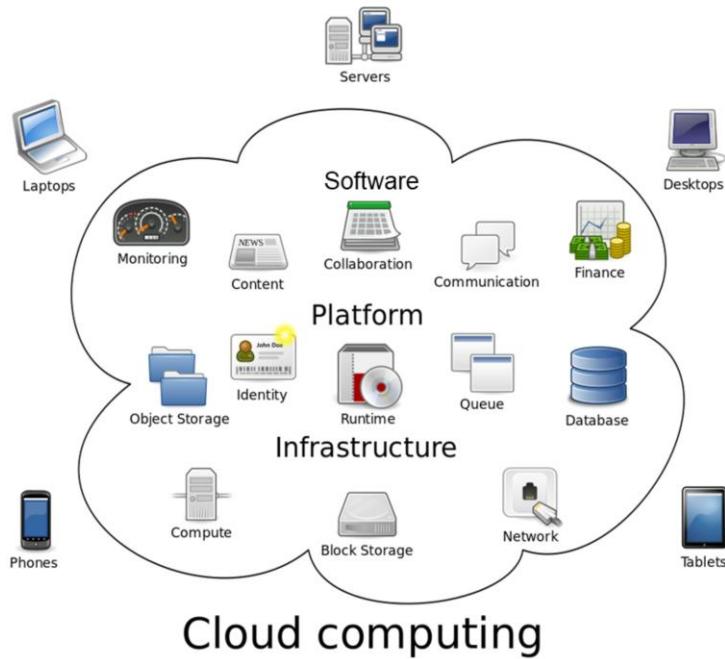
Als we het meer specifiek gaan hebben over een NoSQL database, meer bepaald MongoDB, gaan we merken dat deze standaard werkt met sharding en replicatie (replica sets-. We gaan hier in het volgende hoofdstuk verder op in.

En wat komt er nog aan?

Tot hier toe hebben we de gangbare clustering technieken bekeken waarmee we eigenlijk stilaan aan de limieten van de mogelijkheden zitten. Zeker de klassieke Relationale Database Management Systemen zijn niet eenvoudig te upscalen en kunnen vaak niet de beste scenario's aan. Echter de nieuwere NoSql databases worden van begin af aan ontwerpen met concepten zoals sharding in hun achterhoofd, waardoor ze ook enerzijds eenvoudiger op te zetten zijn maar ook alle mogelijke availability scenario's aankunnen.

De evolutie de laatste jaren stopt echter niet.

Cloud



De laatste 5 jaar is het Cloud-verhaal echt aan't boomen, maar hoe past dit Cloud-verhaal nu in dit plaatje?

Wel, in plaats van zelf clustering-technieken toe te passen en de nodige hardware hiervoor te voorzien, kan je dat eigenlijk uitgeven aan een Cloud provider zoals Amazon AWS, Microsoft Azure, IBM Bluemix, ...

Deze vendors zorgen zelf voor de nodige backup's, opzet van clusters, ... zonder dat jij zelf daar allemaal moet voor zorgen. Je betaalt wel een maandelijkse fee afhankelijk van hoe groot je cluster is en hoe zwaar deze gebruikt worden. Achterliggend maken de vendors gebruik van de opgegeven clustering technieken, afhankelijk dus van het type contract en/of de instellingen die je gebruikt. Daarom is het nog steeds van belang dat - desondanks je vaak iets minder zelf zulke clusters van nul af aan moet opzetten - je toch goed begrijpt hoe een cluster in elkaar zit en wat de voor- en nadelen zijn van de verschillende vormen.

Als we over Cloud spreken, heb je eigenlijk drie vormen: IaaS, PaaS en SaaS.

- SaaS (Software As A Service) => één applicatie, meestal pay-per-user/usage/..., denk aan Office, KdG-mail studenten, Dropbox/OneDrive/GoogleDrive/...

- PaaS (Platform As A Service) => frameworks en/of software die je kan integreren in je eigen applicaties. Meestal gaat het om meer technische software die je kan gebruiken in je eigen software. Denk daarbij aan Identity providers (zodanig dat je je accountbeheer eigenlijk uitgeeft), databanken, ...

- IaaS (Infrastructure As A Service) => dit gaat over de infrastructuur (VM's, Docker images, OS, ...) die je gaat in de cloud zetten (bv. voor scalability), maar meestal ga je nog zelf de nodige applicaties moeten installeren, onderhouden en/of configureren. De focus ligt dus

voornamelijk op het uit handen geven van hardware en vaak ook onderhoud van het OS

Je ziet ook hoe langer hoe meer combinaties van deze diensten verschijnen. Voor maatwerksoftware wordt ook hoe langer hoe meer gebruik gemaakt van PaaS (bv. de databank & Identity management) om dan deze in een IaaS oplossing te deployen.

Zulke Cloud-platvormen bieden als grote voordeel dat je enerzijds niet zelf altijd volledig zelf de vaak complexe setup en vooral onderhoud moet uitvoeren, maar dat een vendor deze doet. Ook de beveiliging van zulke systemen is sterker dan dat je zelf zomaar kan implementeren (sterkere firewalls, snellere security updates, ...), al zijn de cloud platforms wel meer gevoeliger voor aanvallen. Daarnaast is het relatief eenvoudig (en in verhouding goedkoop) om al dan niet tijdelijk je cluster uit te breiden met extra nodes indien je in bepaalde periodes zware loads verwacht. Denk maar aan Webshops tijdens de solden periode, Tax-On-Web enkele dagen voor de deadline, Ticketsales voor een populair concert of festival zoals Tomorrowland, ...

Het zal je dan ook niet verbazen dat Cloud-platvormen de laatste jaren een enorme opmars hebben gemaakt en nu echt main-stream zijn.

Quantum Computing



<https://www.vrt.be/vrtnws/nl/2017/09/27/zal-de-kwantum-computer-de-wereld-redden-/>

<https://www.dwavesys.com/quantum-computing>

<http://newscientist.nl/dossiers/quantumcomputer/>

<http://newscientist.nl/nieuws/race-om-quantumcomputer-gaat-gevangen-ionen-en-microchips/>

The next step om performance te verbeteren, is een stuk radicaler. Met de huidige technologieën is er nog maar weinig hardware-winst mogelijk, dus zoekt men andere mogelijkheden om de klassieke performantie en availability uitdagingen aanpakken.

De opkomst van BigData en de laatste jaren ook AI maakt dat onze sector voor diverse uitdagingen staat. Er zijn verschillende technieken en tools ontwikkelt waarmee we hoe langer hoe sneller grote blokken data kunnen verwerken, maar stilaan lopen we tegen de limieten van de hardware mogelijkheden. Zo is er de laatste jaren qua winst in processorsnelheden weinig vooruitgang geboekt t.o.v.. de periode ervoor. De meeste performantiewinsten zijn ontstaan door verbeteringen in de manier van parallelisatie, zowel binnen één server alsook over meerdere servers. Ook zijn er de laatste jaren vél softwarematige oplossingen gekomen die zulke technieken ten volle benutten, waar er op dat gebied wel nog een enorme vooruitgang geboekt is.

Weet echter dat de evolutie naar hardware alles behalve stilstaat!

Daarom is men op zoek naar alternatieve mogelijkheden en de meest veelbelovende is quantum computing, een concept waarbij de quantummechanica wordt gecombineerd met electronica. Kort samengevat gevatt gaan we in plaats van alle gegevens te stockeren in bits - die een 0 of 1 bevatten - de gegevens bundelen in quantum bits (QuBits). Dit zijn bijzonder kleine objecten (elementaire deeltjes) die vél minder opslagplaats nodig hebben dan de traditionele bits, maar bovendien ook diverse states (0'en en 1'en) tegelijk kunnen bevatten d.m.v. de positionering van elektronen (of fotonen) van zulk een deeltje. Je kan dus als het ware een heel aantal bytes gaan stockeren in één quantum bit.

Waar zit nu het grote voordeel? Wel, door de omgeving van zulk een quantum bit (gecontroleerd) aan te passen, kan je al die 0'en en 1'en binnen een quantum bit tegelijkertijd veranderen. Dit kan je doen door een specifieke elektrische lading (stroom), magneetvelden, omgevingstemperatuur (druk), ... aan te passen. Als je die omgevingsfactoren optimaal beheert en laat aansturen door bepaalde algoritmes, heeft dat tot gevolg dat dus eigenlijk elke quantum bit een processor wordt i.p.v. dat alle bits één voor één door een processor gejaagd moeten worden.

De technologie staat nog niet op punt want het is enorm moeilijk om de quantum deeltjes in de juiste positie te krijgen en vooral te houden, maar diverse grote bedrijven zoals Google, IBM en Microsoft investeren enorm veel in deze technologie. Maar eens dat proces onder controle is, kan je door de quBits om een bepaalde manier te manipuleren,

Het filmpje in deze slide legt op een vrij begrijpelijk manier uit wat quantum computing juist is, al moet je de laatste seconden van het filmpje even de reclame van Microsoft aanhoren ☺.

Mocht deze technologie slagen, zou dit betekenen dat grote sets van data vél sneller verwerkt kunnen doordat elke qubit zichzelf 'intern' kan processen. Bepaalde algoritmes die men uitvoert op ontzettend grote datasets die zelfs met de huidige technologie 10-tallen jaren nodig hebben, zou met deze technologie op enkele uren verwerkt kunnen worden.

Beloeftevolle technologie nog, maar zelfs al krijgt men dat proces binnen enkele jaren onder controle, zal het nog jaren duren vooraleer dit type computing effectief beschikbaar kan gemaakt voor iedereen, dus voorlopig doen we het met de technieken die vandaag de dag gangbaar zijn. Het feit dat de volgende grote vooruitgang in Hardware Quantum Computing zal zijn, staat echter zo goed als vast.

2016: Google 9 Qubits

2017: IBM has 50Qubit die hun state kunnen behouden voor 90 microseconden

2018: Google 72 Qubits

...

Conclusies

Om te begrijpen waarom NoSql hoe langer hoe populairder wordt, hebben we in de eerste stap vooral gekeken waarom en wanneer we naar dit type databanken kijken. Weet dat NoSql niet "the next-step" is qua DMBS, maar een volwaardig alternatief waarvan je moet kunnen inschatten wanneer je deze best gebruikt. Daarom dat we in dit hoofdstuk vooral rond de ontstaansreden van dit type databank hebben toegelicht en we volgende conclusies kunnen trekken.

NoSQL - Alleen maar voordelen?

NoSql heeft vele voordelen t.o.v. de klassieke RDBMS:

- Relaxing ACID
- Eenvoudig en vaak beter in clusters op te zetten
- Dynamisch schema
- Meer gericht op efficiënt gebruik van data i.p.v. efficiënt bewaren van data

Is het dan alleen maar een good-news show?

NoSQL - Alleen maar voordelen?

- Relaxing ACID?
 - Soms is ACID ook echt wel noodzakelijk om functionele redenen
 - "Transacties zijn een noodzakelijk kwaad"
- Dynamisch schema
 - Lastig als er nog andere systemen gekoppeld zijn, zoals een DWH, een applicatie, ...
- Meer gericht op efficiënt gebruik van data i.p.v. efficiënt bewaren van data
 - Andere modeleringstechnieken te leren, elk type databanken heeft zijn eigen specifieke modeleringstechniek

NoSql is dus géén "holy grail", maar naargelang de context van je project wél een zéér goed alternatief

Polyglot persistence



Polyglot = iemand die veel talen spreekt

We hebben voor de verschillende types ook kort aangehaald in welke situatie je typisch voor dat soort van databank kiest. Met de informatie die jullie nu gekregen hebben zouden jullie in principe in staat moeten zijn om zelf op gefundeerde basis een databanktechnologie te kiezen. Vanaf nu mag je dus niet zonder nadenken kiezen voor een relationele databank.

De vraag is natuurlijk hoe dit in een echte bedrijfscontext in zijn werk gaat. Als ontwerper van de architectuur kan je best uitgaan van het principe van polyglot persistence. Je bedrijf moet in staat zijn verschillende talen te spreken.

Slide 3 - remember?

Bij de start van een IT project....veel vragen

Platform (Java, .NET,...)?
Application server?
Front-end framework?
Back-end framework?
Mobiele strategie (native, web,...)?
Methodologie (Scrum,...)?
Database vendor (Oracle, MySQL,...)?
Infrastructuur (Linux, Unix, Windows,...)?
Cloud?
...



Bij de start van een IT project....2 constanten

Type programmeertaal? **Object oriented**
Type database? **Relationeel**

We zijn deze boeiende uitleg gestart met bovenstaande slide: welke vragen moet je beantwoorden bij de opstart van een IT-project...

Alsof dit nog niet genoeg vragen waren, hebben we de enige twee 'constanten' nog in vraag gesteld, waarvan we enkel maar zijn ingegaan op het 'type database'...

Wat nu?

Bij de start van een IT project nog meer vragen

Platform (Java, .NET,...)?

Application server?

Front-end framework?

Back-end framework?

Mobiele strategie (native, web,...)?

Methodologie (Scrum,...)?

Database vendor (Oracle, MySQL,...)?

Infrastructuur (Linux, Unix, Windows,...)?

Cloud?

...

Type database (relationeel, document,...)?

Database vendor (Oracle, MongoDB,...)?

Clustering? Sharding? Replication?

Type programmeertaal (OO, functioneel,...)?



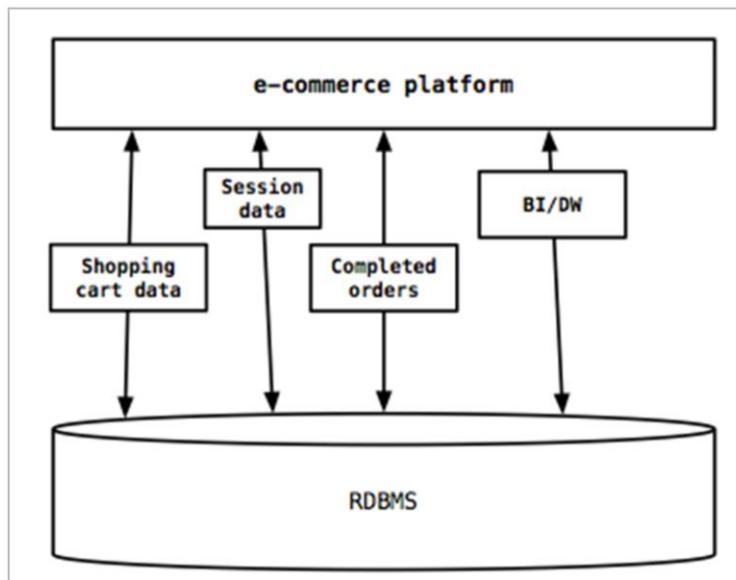
Sinds enkele jaren krijgen we er bij de start van een IT project dus een aantal belangrijke vragen bij:

- Welke databank gaan we gebruiken en kloppen we daarvoor aan bij een oude bekende vendor of gaan we voor de opkomende talenten (open of closed source)
- Hoe zorgen voor de scalability? Passen we sharding en/of replication toe?

Gezien de evolutie van de hoeveelheid en diversiteit van data, kunnen we dus niet meer 'standaard' voor onze RDBMS kiezen. Het kan goed zijn dat er voor jouw project immers een veel beter alternatief is. Je mag NOOIT vergeten dat de technologie maar een hulpmiddel is. Elke technologie heeft zijn sterke en minder goede kanten, dus kies voor elk project de technologie die voor dat specifieke project het meeste geschikt is.

Polyglot persistence

Evolutie van...

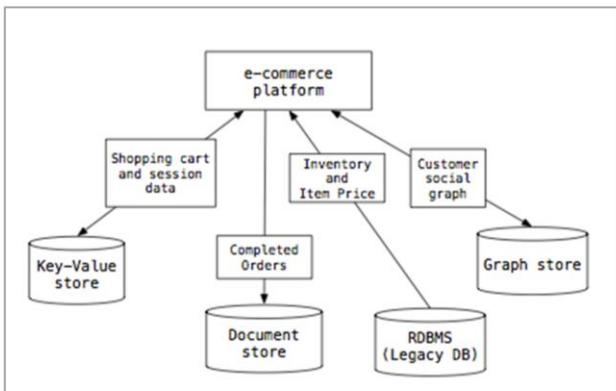


De tijd dat alle systemen in een onderneming worden uitgewerkt met een RDBMS is gedaan... We botsen immers tegen te veel 'limieten' aan van deze klassieke architectuur. Elk systeem heeft zijn eigen specifieke noden en door altijd alles te moeten omvormen naar een vorm die de klassieke RDBMS verwacht, gaat er veel tijd verloren.

Polyglot persistence

...naar

al dan niet met één of
meerdere onderdelen in
de cloud



Perfecte afstemming van de data store op de noden van het probleem domein
Performantie beter met als gevolg ook real-time analyses mogelijk



Complexe infrastructuur
Complexere data integratie
Disaster-recovery scenario's: risico op data-inconsistentie vergroot

... in deze tijd kiezen we er voor om voor elk project/architectuurcomponent/deelfunctionaliteit na te gaan op welke manier we de data het best opslagen. We kiezen voor polyglot persistence.

We voorzien per type applicatie en gegevensstroom de best mogelijke afstemming van de data store. We gaan niet meer het domein omvormen naar de datastore, maar we passen de data store aan. Hiermee kunnen we vél efficiëntie winnen, maar qua infrastructuur wordt het helaas wel iets complexer. Echter in de praktijk wordt er hoe langer hoe meer toch gekozen voor het aanpassen van de data stores omdat daar het meeste efficiëntie te winnen is.

Bedenk maar eens hoe je klassieke relationele data model eruit zou zien als je je eigen social netwerksite zoals FaceBook zou gaan opstellen....

Het grote voordeel van polyglot persistence is dat je door de betere afstemming van de data store op het gebruik, je ook de performantie aanzienlijk kan verbeteren. Queries en/of ETL's die vroegere uren in beslag namen, kunnen nu een pak sneller of worden zelfs overbodig omdat je je data vél beter kan structureren naargelang hoe je ze het meest nodig hebt. Real-time analyses worden daardoor mogelijk waardoor bedrijven vél sneller zich kunnen aanpassen.

Je kan ook één of meerdere stores gaan hosten in de cloud om zo de voordelen van zulk een cloud systeem te kunnen benutten.

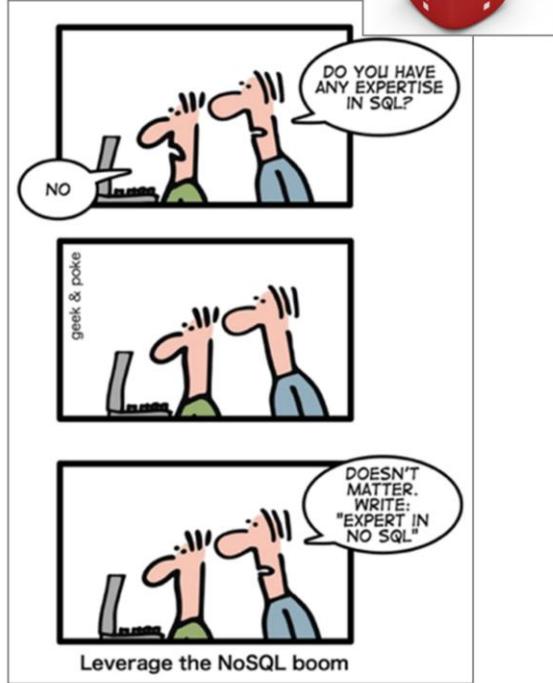
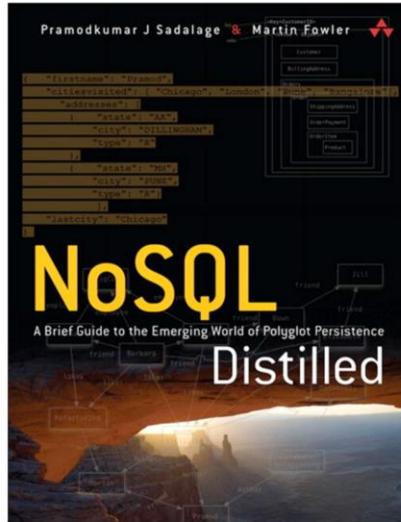
Uiteraard moet je ook goed bewust zijn van de nadelen van dit type persistence model. Enerzijds wordt de infrastructuur een pak complexer, omdat je verschillende systemen en

platvormen hebt die je moet gaan onderhouden. Als je gegevens over deze platvormen heen wil gaan verzamelen voor bv. analytische doeleinden (denk aan BI), moet je ook meer ETL's gaan schrijven en onderhouden.

Een ander groot nadeel is de complexiteit bij een disaster-recovery scenario: wat als er om één of andere reden gegevens (bv. bij een crash van één van de systemen) moeten teruggezet worden naar een vorige versie? Wat was de data in de Key-Value store op het moment dat de Document Store uitviel? Hoe kan ik ervoor zorgen in het order systeem géén orders staan waarvoor er in het klant systeem géén klantgegevens zijn omdat daar een crash was op een bepaald moment? De afhankelijkheden tussen het systeem kan je technisch zo los mogelijk maken, maar je moet je goed bewust zijn dat de data op zich in sommige scenario's inconsistent kan worden over de systemen heen.

Ook bij de micro-services architectuur die momenteel een enorme hype zijn, heeft deze identieke problematiek.

Vragen? Leesvoer!



Een interessant boek trouwens rond polyglot persistance databases en NoSql is trouwens het boek van Pramodkumar J. Sadalage en Martin Fowler (gerenommeerde auteur in ons IT-wereldje).

In de volgende slide-deck gaan we meer concreet ingaan om de meest gangbare vormen van NoSql databases.