Masterproef voorgedragen tot het behalen van de graad van
Master in de informatica/ICT/kennistechnologie

NoSQL: een vergelijkende studie

Auteur:

Glenn Cich

Promotor:

Prof. dr. Frank Neven

Begeleiders:

dr. Tom Ameloot & Bas Ketsman

Assessoren:

Prof. dr. Fabian Di Fiore & Jonny Daenen

Academiejaar: 2013 - 2014

Inhoudsopgave

Dankwoord **vi**

Onderzoeksvraag **vii**

Abstract **ix**

1 Relationele Databases **1**
1.1 Inleiding . . . 1
1.2 Geschiedenis van de Database . . . 1
1.3 Algemene Info . . . 2
1.3.1 Data Modellen . . . 2
1.3.2 Een Uitgewerkt Voorbeeld . . . 3
1.4 ACID . . . 6
1.5 Voorbeeld: ACID . . . 7
1.5.1 Situatieschets . . . 7
1.5.2 Toepassen van ACID op een Voorbeeld . . . 8
1.6 Schaalbaarheid . . . 10
1.6.1 Verticaal Schalen . . . 10
1.6.2 Horizontaal Schalen . . . 11
1.7 Gedistribueerde Databases . . . 12
1.7.1 Doelen . . . 12
1.7.2 Twee Soorten Gedistribueerde Databases . . . 13
1.7.3 Synchroon en Asynchroon . . . 14
1.7.4 Technieken voor Gedistribueerde Databases . . . 15
1.8 Conclusie . . . 18

2 CAP **19**
2.1 Inleiding . . . 19
2.2 De Drie Systeemeisen . . . 20
2.2.1 Consistentie . . . 20
2.2.2 Beschikbaarheid . . . 22
2.2.3 Partitie Tolerantie . . . 23
2.3 Bewijzen van de CAP Theorie . . . 23
2.3.1 Intuïtief Bewijs . . . 24
2.3.2 Formeel Bewijs: Asynchrone Systemen met Communicatieverlies . . . 25
2.3.3 Formeel Bewijs: Asynchrone Systemen, zonder Communicatieverlies . . . 29

2.3.4 Formeel Bewijs: Synchroon Systeem 30
2.3.5 Opmerking 32
2.4 Verdere Inzichten 32
2.4.1 Misverstanden 34
2.4.2 Consistentie geeft Vertraging 35
2.4.3 Updates naar Alle Nodes Tegelijk 36
2.4.4 Updates Eerst naar een Master Node 36
2.4.5 Updates Eerst naar een Willekeurige Node 38
2.5 Partities 38
2.5.1 Beslissingen tijdens een Partitie 40
2.5.2 Herstellen van een Partitie 41
2.5.3 Compenseren van Fouten 42
2.6 Conclusie 44

3 Soorten Consistentie **45**
3.1 Inleiding 45
3.2 Zwakke en Sterke Consistentie 45
3.3 Uiteindelijke Consistentie 47
3.3.1 Liveness en Safety 48
3.3.2 Hoe Goed is Uiteindelijke Consistentie? 49
3.3.3 Compensaties 50
3.3.4 CALM Stelling 51
3.3.5 Commutative, Replicated Data Types 52
3.3.6 Separation of Concerns 54
3.3.7 Bloom 55
3.3.8 BASE 55
3.4 Causale Consistentie 56
3.4.1 Twee Soorten Causale Consistentie 56
3.4.2 Terminologie 57
3.4.3 Causale$^+$ Consistentie 58
3.4.4 Implementatie 59
3.4.5 Mogelijke Gevaren 60
3.4.6 Van Potentiële naar Expliciete Causale Consistentie . . . 63
3.5 Conclusie 64

4 NoSQL **66**
4.1 Inleiding 66
4.2 Wat is NoSQL 66
4.3 Mogelijke Oorzaken NoSQL 67
4.3.1 Big Data 68
4.3.2 Structuur van de Data 69
4.3.3 Drang naar Parallellisatie 70
4.4 Argumenten Relationele Databases vs. NoSQL Databases 70
4.4.1 Pro Relationele Databases 71
4.4.2 Pro NoSQL Databases 73
4.5 Problemen met Specialisatie 76
4.6 Kritiek op NoSQL 76
4.7 Conclusie 78

5 Soorten NoSQL **79**
5.1 Inleiding . . . 79
5.2 Voorbeeld in het Relationeel Model . . . 79
5.3 Key Value Store . . . 81
5.4 Document Store . . . 82
5.5 Graph Databases . . . 85
5.6 Multivalue Databases . . . 85
5.7 Wide Column Store . . . 86
5.8 Conclusie . . . 87

6 Voldemort **89**
6.1 Inleiding . . . 89
6.2 Wat is Voldemort? . . . 89
6.3 Model . . . 90
6.3.1 Naïeve Data Distributie . . . 90
6.3.2 Consistent Hashen . . . 92
6.3.3 Consistentie . . . 94
6.4 Praktische Configuratie van Voldemort . . . 96
6.4.1 Configuratie . . . 97
6.5 Situering in de CAP Theorie . . . 98
6.6 Experiment . . . 99
6.6.1 Doel . . . 99
6.6.2 Idee . . . 100
6.6.3 Opstelling . . . 101
6.6.4 Uitvoering . . . 101
6.6.5 Invloed van W op de Uitvoeringstijd . . . 108
6.7 Conclusie . . . 109

7 MongoDB **111**
7.1 Inleiding . . . 111
7.2 Wat is MongoDB? . . . 111
7.3 Model . . . 112
7.3.1 Replicaties . . . 112
7.3.2 Verkiezing van een Nieuw Primair Component . . . 113
7.3.3 Hoge Beschikbaarheid . . . 114
7.3.4 Lezen en Schrijven . . . 115
7.4 Praktische Configuratie van MongoDB . . . 117
7.4.1 Leesvoorkeuren . . . 117
7.4.2 Schrijfvoorkeuren . . . 118
7.5 Situering in de CAP Theorie . . . 119
7.6 Experiment . . . 119
7.6.1 De Uitvoering . . . 119
7.6.2 Invloed van W op de Uitvoeringstijd . . . 126
7.7 Conclusie . . . 127

8 Conclusie **129**
8.1 Samenvatting . . . 129
8.2 Algemene Conclusie . . . 131

Lijst van figuren

1.1 Schematische voorstelling van verticaal schalen 11
1.2 Schematische voorstelling van horizontaal schalen 12
1.3 Schematische voorstelling van een gedistribueerd homogeen systeem. 14
1.4 Schematische voorstelling van een gedistribueerd heterogeen systeem. 14

2.1 Schematische voorstelling CAP theorie. 20
2.2 Voorbeeld om consistency uit te leggen. 21
2.3 Een voorbeeld waar beschikbaarheid niet in orde is. 22
2.4 Een voorbeeld waar de communicatielijn tussen de twee nodes is verbroken. 24
2.5 Situatie die kan voorkomen als er geen partitie aanwezig is. . . . 25
2.6 Schematische voorstelling van het ontstaan van een partitie. . . . 25
2.7 Situaties die kunnen voorkomen wanneer er een partitie aanwezig is. 26
2.8 Een voorstelling van Algoritme A bij een asynchroon systeem. . . 27
2.9 Schematische voorstelling van het voorbeeld uit het formeel bewijs van asynchrone systemen met communicatieverlies. 28
2.10 Geen communicatie mogelijk tussen N_1 en N_2. 28
2.11 Uitvoering α_2 leest w_0 van node N_2. 29
2.12 Een voorstelling van algoritme A bij een synchroon systeem. . . . 31
2.13 Schematische voorstelling PACELC. 33
2.14 Schematische voorstelling van een systeem waar een partitie ontstaat en die vervolgens hersteld wordt. 39
2.15 Schematische voorstelling die weergeeft hoe we van een gepartitioneerde staat naar een niet-gepartitioneerde, consistente staat gaan. 43

3.1 Schematische voorstelling om de twee groepen consistenties uit te leggen. 46
3.2 Onderscheid tussen sterke en zwakke consistentie. 47
3.3 Schematische voorstelling van een increment operatie. 53
3.4 Schematische voorstelling van een G-counter. 54
3.5 Een causale graaf (potentiële consistentie). 57
3.6 Een causale graaf (expliciete consistentie). 57
3.7 Drie nodes die elke $\frac{A}{3}$ schrijfoperaties kunnen verwerken. 61
3.8 Een voorbeeld van een potentiële causale graaf. 62

3.9 Een voorbeeld van een expliciete variant van Figuur 3.8. 64

4.1 De drie karakteristieken van big data [45]. 69
4.2 Een eenvoudig voorbeeld van een graafdatabase. 72
4.3 Schematische voorstelling van schalen door gebruik te maken van het promotiefilmpje van DynamoDB. 75
4.4 De vergelijking tussen algemene en gespecialiseerde systemen (Kumar [33]). 77

5.1 Voorstelling van het "Student" -"Loopbaan" voorbeeld in een graafvorm. 86

6.1 Schematische voorstelling om het verdelen van data duidelijk te maken. 91
6.2 Schematische voorstelling van consistent hashen. 93
6.3 Schematische voorstelling van consistent hashen (verwijderen en toevoegen van een node). 93
6.4 Bij $N = 4$ is node 1 verantwoordelijk voor de blauwe nodes. . . . 94
6.5 Gebruik van replica's in een hash ring. 94
6.6 Schematische voorstelling van de replication-factor. 98
6.7 Schematische voorstelling van de required-writes. 99
6.8 Schematische voorstelling van de required-reads. 100
6.9 Schematische voorstelling van Voldemort in de CAP theorie. . . . 100
6.10 Schematische voorstelling van de connectie tussen twee laptops via twee routers. 101
6.11 Schematische voorstelling van de opstelling van het experiment. . 102
6.12 Onderling verband tussen de uitvoeringstijden van $W = 1$ en $W = 4$. 110

7.1 Schematische voorstelling van een replica set met verschillende configuraties. 113
7.2 Schematische voorstelling van een verkiezing met een arbiter tijdens een netwerkpartitie. 115
7.3 Schematische voorstelling van het schrijven met $W = 1$. 118
7.4 Schematische voorstelling van het schrijven met $W = 2$. 119
7.5 Schematische voorstelling MongoDB in de CAP theorie. 120
7.6 Schematische voorstelling van de gebruikte replica set. 121
7.7 Schematische voorstelling van een verkiezing tijdens een netwerkpartitie (Primair component bevindt zich in de netwerkpartitie met de meerderheid). 123
7.8 Schematische voorstelling van een verkiezing tijdens een netwerkpartitie (Primair component bevindt zich in de netwerkpartitie met de minderheid). 123
7.9 Schematische voorstelling van een verkiezing tijdens een netwerkpartitie (Beide delen van de netwerkpartities hebben evenveel componenten). 124
7.10 Onderling verband tussen de uitvoeringstijden van $W = 1$ en $W = 4$. 127

Dankwoord

Deze masterthesis is niet alleen door mij tot stand gekomen. Daarom zou ik graag iedereen willen bedanken die mij hiermee rechtstreeks en onrechtstreeks geholpen heeft.

Eerst zou ik graag mijn promotor Prof. dr. Frank Neven en mijn begeleiders dr. Tom Ameloot & Bas Ketsman willen bedanken. Als ik vragen had, stonden zij meteen klaar om mij verder te helpen. Ook hebben ze mijn tekst altijd voorzien van feedback. Tijdens de vergaderingen gaven ze mij ook nieuwe inzichten die nuttig waren voor de thesis.

Daarnaast wil ik ook Jonny Daenen bedanken. Hij heeft dit thesisonderwerp aan mij voorgesteld en heeft mij in het begin van de thesis geholpen als begeleider.

Mensen die ik ook niet mag vergeten zijn mijn ouders. Zij hebben mij altijd gesteund tijdens mijn studies en zonder hen was studeren immers ook niet mogelijk geweest.

Mijn vrienden mag ik ook zeker niet vergeten. Zij zorgden immers altijd voor de nodige ontspanning.

Als laatste zou ik mijn vriendin Mandy willen bedanken. Ik heb enorm veel steun aan haar gehad gedurende mijn hele universitaire loopbaan. Daarnaast stond ze ook altijd klaar om mijn thesis na te lezen.

Helchteren
11 Juni, 2014

Onderzoeksvraag

In deze moderne tijden is er een overvloed aan informatie. Er is data van koopgedrag, wetenschappelijke informatie over DNA en sociale data zoals de profielen van Facebook. Er is een zeer sterke groei van informatie die moet verwerkt worden. Relationele databases zijn altijd geschikt geweest om data te bewaren, op te vragen en te bewerken. De laatste jaren is er echter in bepaalde domeinen zoveel data dat relationele databases minder geschikt zijn om dergelijke hoeveelheden data efficiënt te verwerken. Met efficiënt bedoelen we bijvoorbeeld omgaan met consistentie of data in real time kunnen verwerken. Er is tegenwoordig een grote opkomst van de NoSQL databases die beweren dit probleem op te kunnen lossen omdat ze gemakkelijk te schalen zijn.

In deze masterproef focussen we ons daarom op deze NoSQL databases. We bekijken enkele technieken die gebruikt worden in deze databases en we bekijken ook enkele bestaande systemen.

Deze thesis heeft eigenschappen van twee soorten masterproeftypes: enerzijds een *vergelijkende studie* en anderzijds een *domeinstudie*. In de vergelijkende studie bekijken we enkele vormen van consistentie en bekijken we verschillende technieken die gebruikt worden bij bestaande NoSQL databases. Daarnaast worden verschillende design keuzes vergeleken tussen de relationele databases en de NoSQL databases. De domeinstudie vertaalt zich in een onderzoek naar de NoSQL databases en bekijkt voor- en nadelen ten opzichte van de relationele databases. Ook worden hier de verschillende keuzes belicht waar we rekening mee moeten houden bij het kiezen of implementeren van NoSQL databases.

Kort samengevat staat de volgende onderzoeksvraag centraal in deze thesis:

"Wat is NoSQL en wat is het verband met consistentie?"

Het eerste deel van deze onderzoeksvraag gaat dieper in op het ontstaan van de NoSQL databases. We bekijken waarom ze tegenwoordig zo populair zijn en wie er gebruik van maakt. Daarnaast maken we een vergelijking tussen de NoSQL databases en de relationele databases. We bekijken hier vooral de schaalbaarheid en de design keuzes van beide soorten databases. Het beantwoorden van deze vraag is belangrijk omdat er tegenwoordig heel wat NoSQL databases worden ontwikkeld. Daarom moeten we voorzichtig zijn. Het lijkt immers niet verstandig om zomaar over te schakelen van een relationele database naar een NoSQL databases zonder te weten wat de effectieve voor- en nadelen zijn.

Het tweede deel van deze onderzoeksvraag zal het grootste deel van deze thesis in beslag nemen. NoSQL databases zijn gedistribueerde systemen. Een gedistribueerd systeem is een verzameling van onderling verbonden servers, ge-

naamd nodes. Als alle verbindingen werken, zijn er weinig problemen aangezien iedereen met iedereen kan communiceren. Problemen ontstaan wanneer er effectief een communicatielijn verbroken wordt. Als er een communicatielijn verbroken wordt, ontstaan er twee delen in het systeem. Dit noemen we een netwerkpartitie. Vanaf dat moment moeten er keuzes gemaakt worden. We bekijken in deze thesis deze keuzes. Het belangrijkste punt van deze keuze is wat we doen met de consistentie van ons systeem. Als er namelijk geen communicatie mogelijk is, is er een kans dat data niet verder gepropageerd kan worden en dat sommige nodes niet de meest recente data bevatten. Dit noemen we inconsistentie. We stellen ons de vraag hoe dit kan worden opgelost. Hierbij vragen we ons ook af wat de gevolgen zijn van dergelijke inconsistenties. Daarnaast stellen we ons de vraag wat er gebeurt als we tijdens een netwerkpartitie toch consistentie willen bewaren. Is dat mogelijk en wat zijn de gevolgen hiervan? Het beantwoorden van dergelijke vragen is belangrijk omdat we ons op deze manier kunnen voorbereiden op bepaalde situaties die zich kunnen voordoen tijdens een netwerkpartitie. Het is noodzakelijk dat we weten wat er gebeurt en waarom dit gebeurt. Verschillende applicaties hebben andere verwachtingen wat betreft consistentie en deze vragen beantwoorden wat er in het achterhoofd moet gehouden worden wat betreft voor- en nadelen.

Abstract

We leven tegenwoordig in een samenleving waarin we graag alles opslaan wat we zien, horen of doen. De hoeveelheid data blijft maar stijgen, waardoor we met big data te maken krijgen. Voorbeelden van grote organisaties die te maken hebben met big data zijn Google en Facebook. Denk maar aan alle profielen, berichten, foto's en video's die beide bedrijven moeten opslaan van miljoenen gebruikers. Data wordt traditioneel opgeslagen in relationele databases. Deze databases hebben hiervoor reeds vele jaren perfect dienst gedaan. Ze zijn immers zeer betrouwbaar en zijn bestend tegen het verlies van data. Het probleem duikt nu op dat deze databases minder geschikt zijn voor het opslaan en verwerken van big data. Dit soort databases zijn perfect geschikt zijn om verticaal te schalen. Dit wil zeggen dat we een betere en snellere server aankopen. Het probleem is dat ze minder geschikt zijn om horizontaal te schalen. Hiermee bedoelen we dat we een gedistribueerd systeem bouwen waarbij alle nodes in dat systeem met elkaar verbonden zijn. De data kan dan toegevoegd en opgevraagd worden aan en van deze verschillende nodes.

Databases die tegenwoordig heel populair zijn en die gebruik maken van deze gedistribueerde techniek zijn de NoSQL databases. NoSQL kan op twee manieren worden geïnterpreteerd, maar de meest aangenomen definitie is "Not Only SQL". Dit komt neer op het feit dat dergelijke databases niet alleen maar SQL gebruiken om hun data op te vragen of toe te voegen. Er zijn tegenwoordig heel wat verschillende soorten NoSQL databases. Dit kan gaan van key-value stores tot graafdatabases. Ze kampen echter wel allemaal met hetzelfde probleem en dat is dat ze gedistribueerd zijn en dat ze daarom onderhevig zijn aan de CAP theorie.

De CAP theorie staat voor consistentie (**c**onsistency), beschikbaarheid (**a**vailibility) en partitie tolerantie (**p**artition tolerance). Volgens de CAP theorie kunnen er slechts twee van deze drie eigenschappen tegelijk voldaan zijn in een gedistribueerd systeem. Bij een gedistribueerd systeem en dus ook bij NoSQL databases is het belangrijk dat een systeem om kan gaan met netwerkpartities. Het uitvallen van een communicatielijn tussen twee nodes mag dus geen zware hinder veroorzaken aan het systeem. Daarom moet er dus een keuze gemaakt worden tussen enerzijds beschikbaarheid (kunnen we het systeem queryen?) en anderzijds consistentie (zien alle nodes in het systeem dezelfde data?).

Het doel van deze thesis is om alle aspecten van deze keuzes te onderzoeken. Als we voor consistentie kiezen, wat zijn dan de voor- en nadelen? Heeft dit een bepaalde kost? In welke gevallen kiezen we best voor consistentie? Als we voor beschikbaarheid kiezen, wat gebeurt er dan met de consistentie? Kunnen we ondanks alles nog een bepaalde graad van consistentie bewaren? We zien in de thesis dat er zoiets bestaat als uiteindelijke consistentie. Deze consistentie zorgt

ervoor dat we er zeker van zijn dat het systeem ooit naar een consistente staat zal evolueren. Naast de keuzes bespreken we ook NoSQL in het algemeen. We bespreken onder andere wat de pro's en contra's zijn tegenover de traditionele relationele databases.

Het eerste deel van de thesis bespreekt de bovenstaande onderwerpen. Dit is dus een vrij theoretisch stuk, waar we ons verdiepen in de literatuur.

Het tweede deel is een beetje praktischer. We bespreken hier twee bestaande NoSQL databases. We bekijken welke technieken ze toepassen en waar ze zichzelf plaatsen binnen de CAP theorie. Verder worden er nog enkele experimenten uitgevoerd in verband met consistentie en beschikbaarheid om te kijken hoe de databases reageren op bepaalde situaties.

We sluiten ten slotte af met een algemene conclusie. Hierin geven we nog een korte samenvatting en herhalen we de kernpunten van onze thesis.

Hoofdstuk 1

Relationele Databases

1.1 Inleiding

Om een goede vergelijking te kunnen maken tussen relationele databases en NoSQL databases, is een korte, algemene uitleg over relationele databases onmisbaar.

In dit Hoofdstuk geven we eerst een algemene uitleg over het relationeel model.

Vervolgens bekijken we het strikte transactie model van relationele databases, namelijk het ACID model. Dit is een compleet verschillend model dan hetgene dat wordt gebruikt bij de NoSQL databases.

Verder gaan we dieper in op het schalen van databases. Dit is een belangrijk punt in de thesis, aangezien er een duidelijk verschil is tussen het schalen van relationele databases ten opzichte van het schalen van NoSQL databases.

Ten slotte bespreken we gedistribueerde databases.

1.2 Geschiedenis van de Database

Het verhaal van de relationele databases begint in 1970, toen E.F. Codd het relationeel model introduceerde. Dit model zou de oplossing moeten worden voor de problemen die er waren bij de netwerk- en hiërarchische modellen. Het probleem bij deze modellen was dat er maar weinig ondersteuning was voor *data onafhankelijkheden*. Data onafhankelijkheid is een belangrijke eigenschap van de huidige databases. Dit houdt in dat we onze data beheren, los van alle applicaties die ze gebruiken. Een ander probleem dat academici hadden bij de netwerk- en hiërarchische modellen was dat er geen theoretische onderbouwing voor bestond.

De paper van Codd over het relationeel model werd met succes onthaald door de academische wereld. Er waren echter wel groepen die twijfels hadden over de prestaties van deze relationele databases [32].

Het duurde tot de jaren '80 vooraleer de relationele databases echt populair werden. Dit kwam dankzij *System R*, een experimentele database die ontwikkeld is door IBM. Dit systeem werd ontwikkeld om aan te tonen dat de voordelen van het relationeel model effectief gewaarborgd kunnen worden in een alledaags

systeem met hoge prestaties [14]. Vanaf dit moment waren de relationele databases de grote spelers op de markt van de databases en werden ze op een competitieve manier ontwikkeld.

1.3 Algemene Info

In deze Paragraaf bespreken we de algemene terminologie die noodzakelijk is om te kunnen spreken over relationele databases.

1.3.1 Data Modellen

Als we over relationele databases spreken, is het woord *data modellen* een fundamenteel begrip. Garcia-Molina et al. [27] bespreekt een data model als een manier om data op een eenduidige manier te kunnen beschrijven. We bespreken data modellen aan de hand van drie begrippen:

- de structuur van de data
- de operaties op de data
- de beperkingen op de data

We geven voor alle drie de begrippen een korte uitleg, eventueel in combinatie met een voorbeeld.

Structuur van de Data

Als we aan datastructuren denken in de context van programmeertalen, dan denken we aan structuren zoals arrays of gelinkte lijsten. In relationele databases spreken we over een *database schema*. Dit schema bespreekt hoe de data wordt gerepresenteerd en op welke manier de data wordt opgeslagen. Zo kan een voornaam bijvoorbeeld een VARCHAR(100) zijn en een leeftijd bijvoorbeeld een INTEGER [27].

Operaties op de Data

Als we eenmaal de data op een degelijke manier kunnen representeren, kunnen we denken aan de *operaties* die we erop kunnen verrichten. Met operaties bedoelen we handelingen waarmee we de data kunnen beïnvloeden. In standaard programmeertalen hebben we een grote vrijheid wat betreft de operaties. In database systemen zijn we echter beperkt tot twee grote groepen operaties: enerzijds *queries* om data op te vragen en anderzijds *modificaties* om de data aan te passen. We hebben met andere woorden minder vrijheid, maar dit is eerder een sterkte dan een zwakte. Door de beperktheid in operaties kunnen we op een hoger niveau programmeren en kunnen de operaties optimaal worden uitgevoerd door het *Database Management System* (*DBMS*) [27].

Beperkingen op de Data

Om ervoor te zorgen dat we data op een realistische en juiste manier kunnen weergeven, kunnen er *beperkingen* worden opgelegd in database systemen. Deze beperkingen kunnen van allerlei vormen zijn. We kunnen bijvoorbeeld eisen dat het loon van een werknemer nooit groter mag zijn dan 4000 Euro, of dat een kind exact één biologische moeder heeft. Deze beperkingen zorgen ervoor dat we niet alles kunnen toevoegen aan een database. Als we dit toch proberen, zal het DBMS een foutmelding geven [27].

1.3.2 Een Uitgewerkt Voorbeeld

In deze Paragraaf leggen we het *Relationeel (data) Model* [27] uit aan de hand van de beschrijving die we gegeven hebben in Paragraaf 1.3.1. Het relationeel model is een databasemodel dat gebaseerd is op het gebruik van *tabellen*. Aan de hand van deze tabellen of ook wel *relaties* genoemd, proberen we de data voor te stellen.

In deze Paragraaf gebruiken we een fictief voorbeeld over een vliegtuigmaatschappij.

We beginnen met het definiëren van de structuur. Dit doen we in een database door gebruik te maken van een database schema waarin we voor elke relatie de bijhorende attributen definiëren. Merk op dat we de keys van elke relatie onderstrepen. Dit database schema is te vinden in Listing 1.1 en de bijhorende SQL statements in Listing 1.2.

```
Personeel (
        persoonsID: integer,
        naam: string,
        functie: string,
        jarenDienst: integer
)

Vliegtuig (
        vliegtuigID: integer,
        type: string,
        aantalMotoren: integer,
        aantalPassagiers: integer
)

PilootVliegtuig (
        persoonsID: integer,
        vliegtuigID: integer
)
```

Listing 1.1: Database schema voor het voorbeeld van de vliegtuigmaatschappij.

```
CREATE TABLE Personeel (
        persoonsID INT,
        naam VARCHAR(256),
```

```
        functie VARCHAR(256),
        jarenDienst INT,
        PRIMARY KEY (persoonsID)
);

CREATE TABLE Vliegtuig (
        vliegtuigID INT,
        type VARCHAR(256),
        aantalMotoren INT,
        aantalPassagiers INT,
        PRIMARY KEY (vliegtuigID)
);

CREATE TABLE PilootVliegtuig (
        persoonsID INT,
        vliegtuigID INT,
        FOREIGN KEY (persoonsID)
            REFERENCES Personeel (persoonsID),
        FOREIGN KEY (vliegtuigID)
            REFERENCES Vliegtuig (vliegtuigID)
);
```

Listing 1.2: SQL statements om het schema uit Listing 1.1 te creëren.

Als de structuur gedefinieerd is, kunnen we de database opvullen met data. Voorbeeld instanties van de drie database schema's uit Listing 1.1 zijn respectievelijk te vinden in Tabel 1.1, Tabel 1.2 en Tabel 1.3.

persoonsID	**naam**	**functie**	**jarenDienst**
1	John	Steward	5
2	Lisa	Stewardess	7
3	Bert	Piloot	6
4	Ann	Piloot	2

Tabel 1.1: Voorbeeld instantie van het database schema "Personeel".

vliegtuigID	**type**	**aantalMotoren**	**aantalPassagiers**
100	Boeing747	4	660
200	Airbus A330-300	2	440
300	Boeing777	2	440

Tabel 1.2: Voorbeeld instantie van het database schema "Vliegtuig".

Elke relatie bevat een aantal *attributen* samen met *attribuutwaardes*, zo is "persoonsID" bijvoorbeeld een attribuut van de relatie "Personeel" (Tabel 1.3) en zijn "1,2,3,4" de bijhorende attribuutwaardes van "persoonsID". Verder wordt elke rij met waardes een *tupel* genoemd. Zo is $(1, \textit{John}, \textit{Steward}, 5)$ een tupel waarvoor geldt:

persoonsID	vliegtuigID
3	100
3	200
3	300
4	200
4	300

Tabel 1.3: Voorbeeld instantie van het database schema "PilootVliegtuig".

$$(1, \textit{John}, \textit{Steward}, 5) \in \textit{Personeel}$$

$(1, \textit{John}, \textit{Steward}, 5)$ is dus een tupel van de relatie Personeel. Bovenstaande redeneringen kunnen triviaal worden doorgetrokken naar de relaties van Tabel 1.2 en Tabel 1.3.

Op deze relaties kunnen we nu allerlei queries uitvoeren. We kunnen bijvoorbeeld opzoeken hoelang "John" al werkt bij de maatschappij, dit kan door een eenvoudig SQL query. Deze query wordt weergegeven in Listing 1.3.

```
SELECT  jarenDienst
FROM    Personeel
WHERE   naam = 'John'
```

Listing 1.3: SQL statement om het aantal dienstjaren van "John" op te vragen.

Het resultaat van de query uit Listing 1.3 is te vinden in Tabel 1.4.

jarenDienst
5

Tabel 1.4: Het resultaat van de query uit Listing 1.3

We kunnen ook complexere queries uitvoeren door relaties met elkaar te *joinen*. Zo kunnen we bijvoorbeeld opzoeken met welk type vliegtuig "Ann" mag vliegen. Deze query is te vinden in Listing 1.4.

```
SELECT  type
FROM    Personeel AS P, Vliegtuig AS V,
        PilootVliegtuig AS PV
WHERE   naam = 'Ann'
        AND P.persoonsID = PV.persoonsID
        AND PV.vliegtuigID = V.vliegtuigID
```

Listing 1.4: SQL statement om de types van vliegtuigen waarmee "Ann" mag vliegen op te vragen.

type
Airbus A330-30
Boeing777

Tabel 1.5: Het resultaat van de query uit Listing 1.4

Het resultaat van de query uit Listing 1.4 is terug te vinden in Tabel 1.5.

Ten slotte bespreken we het gebruik van constraints. In het schema uit Listing 1.2 kunnen we reeds twee soorten constraints zien. Enerzijds hebben we een *primary key*. Dit wil zeggen dat dit attribuut uniek is en dat het dus maar één keer kan voorkomen in de relatie. Anderzijds hebben we een *foreign key*. Dit wil zeggen dat het attribuut uit die relatie ook in een andere relatie *moet* zitten. Zo zien we dat de attribuutwaardes van "persoonsID" van de relatie "PilootVliegtuig" ook in de relatie "Personeel" voorkomen.

In de volgende Paragraaf bekijken we hoe de betrouwbaarheid van transacties in relationele databases wordt gewaarborgd.

1.4 ACID

Relationele databases staan erom bekend dat ze heel betrouwbaar zijn. Dit brengt ons bij het *ACID* acroniem [27]. We leggen eerst de terminologie uit en passen deze daarna toe op een voorbeeld.

ACID staat voor vier eigenschappen die garanderen dat transacties op een betrouwbare en correcte manier worden uitgevoerd. De vier eigenschappen zijn:

1. atomair (Engels: atomic)
2. consistent (Engels: consistent)
3. geïsoleerd (Engels: isolated)
4. duurzaam (Engels: durable)

De eerste eis van ACID is *atomair* zijn. Hiermee bedoelen we dat tijdens het uitvoeren van een transactie een "alles of niets" situatie wordt gecreëerd. Met andere woorden wordt een transactie helemaal uitgevoerd, ofwel totaal niet. Als we bijvoorbeeld tijdens een transactie te maken krijgen met een error (wat allerlei soorten redenen kan hebben), dan zal de transactie worden geannuleerd en de database zal ervoor moeten zorgen dat het stuk van de transactie dat reeds is uitgevoerd, ongedaan wordt gemaakt. Om ervoor te zorgen dat de transactie de database toch aanpast, moeten we deze later opnieuw proberen uit te voeren. De atomaire eis moet worden gewaarborgd in alle situaties en tegen alle mogelijke fouten, zoals bijvoorbeeld het uitvallen van de database door een stroomtekort of een crash van het databasesysteem [16].

De tweede eis is gebaseerd op *consistentie*. In ACID gaat consistentie om het feit dat elke transactie de voorgedefinieerde beperkingen van de database niet mag schenden. Deze beperkingen hangen sterk af van de toepassing waarvoor we de database ontwerpen. Als een persoon bijvoorbeeld een overschrijving doet van zijn spaarrekening naar zijn zichtrekening is de beperking dat de totale som geld van deze persoon hetzelfde blijft. Als een transactie dergelijke beperkingen

wel schendt, zal de database deze transactie ongedaan maken en terugkeren naar een staat (vóór deze transactie) die wel consistent was. Als er daarentegen een transactie gebeurt die wel binnen de beperkingen valt, zal de database van de ene consistente staat naar de andere consistente staat overschakelen. Merk op dat de programmeur van de database deze beperkingen kan toevoegen via *triggers* of *assertions* die ook in het databaseschema zijn opgenomen [16].

De derde eis heeft te maken met het *isoleren* van transacties. De bedoeling hiervan is dat we er voor zorgen dat gelijktijdige transacties elkaar niet storen. Tijdens het uitvoeren van een operatie, kan een andere operatie niet op dezelfde data werken, omdat we op deze manier snel fouten kunnen maken. Als een transactie een bepaalde tupel opvraagt terwijl een andere transactie deze tupel aan het bewerken is, ontstaan er problemen. Als twee transacties daarentegen gelijktijdig tupels opvragen, is er geen probleem [16].

De laatste eis van ACID is de eis van *duurzaamheid*. Door duurzaamheid zijn we zeker dat uitgevoerde transacties niet verloren gaan. Hiervoor is het nodig dat we allerlei maatregelen treffen zoals het gebruik van bijvoorbeeld backups. Duurzaamheid moet blijven gelden, zelfs als er fouten voorkomen zoals het uitvallen van een database door een elektriciteitspanne. We moeten met andere woorden zeker zijn dat de data opgeslagen is op een *niet-vluchtig*, of in het Engels *non-volatile*, opslagmedium [16].

Om ACID nog beter te duiden, geven we in de volgende Paragraaf een voorbeeld waarin deze begrippen worden toegepast.

1.5 Voorbeeld: ACID

In deze Paragraaf schetsen we eerst de data waarmee we te maken krijgen samen met de bijhorende beperkingen. Vervolgens illustreren we de vier eigenschappen van ACID aan de hand van deze data.

1.5.1 Situatieschets

Stel dat we werken met een database die uit één relatie bestaat met twee attributen. Elk attribuut is een integer en hierop kunnen allerlei wiskundige bewerkingen worden uitgevoerd. Deze relatie zien we in Tabel 1.6.

X	**Y**
30	45
30	30
10	50
...	...

Tabel 1.6: Voorbeeldrelatie om ACID eigenschappen uit te leggen.

Elke tupel moet voldoen aan de volgende beperkingen:

$$(X + Y) < 80$$

$$X \leq Y$$

$$(X + Y) \geq 60$$

Dit zijn, zoals besproken in Paragraaf 1.4, voorgedefinieerde beperkingen. Merk op dat alle drie de eigenschappen voldaan moeten zijn om als "consistent" te worden gezien.

De drie weergegeven tupels in Tabel 1.6 voldoen alle drie aan deze beperkingen.

Nu we de data met de bijhorende beperkingen besproken hebben, kunnen we de ACID principes hierop toepassen.

1.5.2 Toepassen van ACID op een Voorbeeld

In deze Paragraaf illustreren we waarom de ACID eigenschappen van groot belang zijn. Dit doen we door voor elke eigenschap een situatie te creëren die ongewenste gevolgen met zich meebrengt als de ACID eigenschappen *niet* zouden nageleefd worden.

We bespreken alle principes aan de hand van de eerste tupel uit Tabel 1.6, namelijk $(30, 45)$.

Atomaire Eigenschap

We gebruiken de volgende transactie:

$$X - 15$$

$$Y + 15$$

Als alles goed verloopt, is er geen probleem aangezien de tupel dan nog steeds aan alle eisen zou voldoen ($\{15, 60\}$). We veronderstellen echter dat 15 wordt afgetrokken van X en op dat moment gebeurt er iets waardoor de optelling bij Y niet kan worden uitgevoerd. Hierdoor wordt een beperking overschreden, namelijk het feit dat de som van de twee integers groter moeten zijn dan 60. De atomaire eigenschap wordt ook geschonden aangezien ofwel de volledige transactie uitgevoerd moet worden ofwel niets.

Consistentie Eigenschap

We gebruiken de volgende transactie:

$$X - 30$$

We gaan ervan uit dat de volledige transactie slaagt waardoor de atomaire eigenschap voldaan is. Na elke transactie wordt echter ook gecontroleerd of het resultaat nog steeds voldoet aan de opgelegde beperkingen. In dit geval zijn de eerste en de tweede beperking voldaan, maar is er een probleem met de laatste, aangezien $(X + Y) \ngeq 60$. Deze beperking is niet voldaan, dus moet de database terugkeren naar de staat van vóór deze transactie $(30, 45)$.

Isolatie Eigenschap

Stel dat er twee transacties tegelijk een bewerking willen uitvoeren op de tupel $\{30, 45\}$. De eerste transactie t_1:

$$X - 5$$

$$Y + 5$$

De tweede transactie t_2:

$$Y - 5$$

$$X + 5$$

We merken op dat door het toepassen van deze transacties de consistentie-eis niet wordt geschonden.

We tonen nu aan waarom isolatie noodzakelijk is. In normale omstandigheden zal de database isolatie toepassen. Stel dat in ons voorbeeld t_1 eerst toegang krijgt tot de data en daarna t_2. In dit geval zouden we de volgende volgorde van modificaties krijgen:

- t_1: X - 5
- t_1: Y + 5
- t_2: Y - 5
- t_2: X + 5

Dit is de normale manier van werken, want stel nu dat t_1 na het aftrekken van 5 van X op een of andere manier faalt, dan zal de database door de atomaire eis terugkeren naar de staat van vóór t_1. Er is met andere woorden niets gebeurd. Vervolgens kan t_2 zonder problemen zijn transacties uitvoeren op de correcte data. Daarna kan het systeem eventueel opnieuw t_1 proberen uit te voeren.

Stel dat we niet met de isolatie-eis werken. De verschillende operaties van t_1 en t_2 mogen met andere woorden door elkaar worden uitgevoerd. Een volgorde van bewerkingen kan zijn:

- t_1: X - 5
- t_2: Y - 5
- t_2: X + 5
- t_1: Y + 5

Als alles normaal verloopt zal er in feite geen probleem zijn. We eindigen met net hetzelfde resultaat als in de vorige situatie. Het probleem met deze situatie is het feit dat operaties kunnen falen. Als we dezelfde situatie simuleren zoals in het vorige voorbeeld, dan bekomen we een verkeerd resultaat. Eerst trekt t_1 probleemloos 5 af van X. Vervolgens wordt t_2 volledig correct uitgevoerd. Tijdens het optellen van 5 bij Y van t_1, gebeurt er een fout, waardoor de atomaire eis wordt ingeschakeld. Vanaf dit moment zitten we met een probleem, want t_2 heeft met succes zijn operaties toegepast op de tupel. Hierdoor kan het systeem veel moeilijker terugkeren naar een correcte staat.

Duurzaamheid Eigenschap

We gebruiken de volgende transactie:

$$Y - 5$$

De database voert deze transactie uit en zendt een bericht naar de gebruiker dat zijn bewerking met succes is uitgevoerd. Veronderstel echter dat de bewerking nog in een schrijfbuffer staat om later weggeschreven te worden. Op dit moment valt de stroom uit en gaat de bewerking verloren. De database moet dus altijd voorbereid zijn om dergelijke fouten op te kunnen vangen.

1.6 Schaalbaarheid

Een belangrijk punt in deze thesis is het begrip schaalbaarheid omdat hier een groot verschil is tussen NoSQL databases en relationele databases. In deze Paragraaf beschrijven we eerst het begrip schaalbaarheid en kijken we op welke manieren we kunnen schalen.

Met *schaalbaarheid* [24] bedoelen we de mogelijkheid van een systeem om telkens verhoogde werklasten te kunnen opvangen. Als bijvoorbeeld meer clients gebruik maken van een server, dan zal deze server moeten worden uitgebreid om de toenemende werklast te kunnen opvangen.

Er zijn twee mogelijkheden om deze schaalbaarheid uit te breiden:

- verticaal schalen
- horizontaal schalen

We bespreken deze mogelijkheden in de volgende Paragrafen.

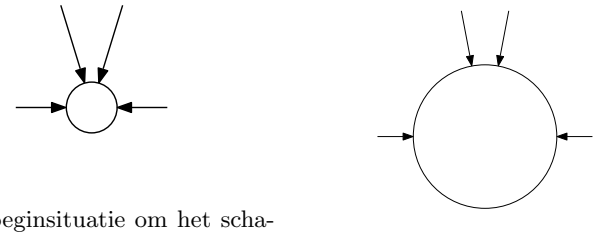
1.6.1 Verticaal Schalen

Verticaal schalen wordt toegepast op één *node*. Met een node bedoelen we een verwerkende eenheid, zoals in ons geval bijvoorbeeld een database. Als we verticaal schalen, proberen we de huidige hardware te upgraden. Zo kunnen we bijvoorbeeld snellere processoren of meer geheugen toevoegen.

Deze manier biedt het voordeel dat er in principe geen bijkomende algoritmes moeten worden geïmplementeerd, maar dat het systeem wel grotere hoeveelheden data en transacties kan verwerken.

Het nadeel van deze manier van werken, is dat we gelimiteerd zijn tot de hardware. De hardware kan namelijk niet oneindig sneller worden. Een ander probleem is dat goede hardware snel heel duur wordt en dat de extra kost misschien niet opweegt tegen de behaalde voordelen [24].

Om verticaal schalen te verduidelijken, geven we in Figuur 1.1 een schematisch voorbeeld van deze situatie. In Figuur 1.1a zien we de beginsituatie. De cirkels stellen een verwerkende eenheid (node) voor zoals bijvoorbeeld een database. De grootte van deze cirkels geven de verwerkingskracht van de node aan en de pijlen stellen de operaties voor die binnenkomen bij de node. We zien dus dat deze database vier operaties krijgt toegestuurd. Als we dit systeem verticaal schalen, bekomen we een situatie zoals voorgesteld in Figuur 1.1b. We zien dat



(a) De beginsituatie om het schalen van een database voor te stellen.

(b) Verticaal schalen van 1.1a.

Figuur 1.1: Schematische voorstelling van verticaal schalen

de verwerkende eenheid groter geworden is. Hiermee geven we aan dat hij meer verwerkingskracht heeft gekregen.

Deze manier van schalen wordt meestal gebruikt bij de traditionele relationele databases die bijvoorbeeld maar uit één node bestaan.

1.6.2 Horizontaal Schalen

De tweede manier is *horizontaal schalen*. Horizontaal schalen is het toevoegen van verschillende nodes. Met andere woorden voegen we verwerkende eenheden toe en laten deze met elkaar communiceren. Hierdoor wordt het werk verdeeld over de verschillende nodes. In tegenstelling tot verticaal schalen vereist deze manier van werken niet telkens betere hardware. We hebben bijvoorbeeld een systeem dat uit één node bestaat en deze node krijgt twintig transacties om te verwerken. De node zal al deze twintig transacties zelf moeten afhandelen. Als we daarentegen het systeem uitbreiden naar vier nodes, kan de werklast zo verdeeld worden dat elke node maar vijf transacties moet afhandelen.

Het voordeel van deze manier van werken, is dat we geen dure hardware moeten aanschaffen. We kunnen door relatief goedkope hardware de potentiële werklast van het systeem blijven verhogen door (oneindig) veel nieuwe nodes toe te voegen.

Het grote probleem hierbij is dat we andere algoritmes nodig hebben en dat er veel communicatie nodig is. Een communicatielijn die uitvalt tussen verschillende nodes kan al snel leiden tot grote problemen [24].

In Figuur 1.2 geven we opnieuw een schematisch voorbeeld. De beginsituatie is hetzelfde als in ons vorig voorbeeld (Paragraaf 1.6.1) over verticaal schalen. We zien de beginsituatie in Figuur 1.2a en we gebruiken dezelfde terminologie als in het voorbeeld van Paragraaf 1.6.1. Dit keer passen we horizontaal schalen toe, waarvan het resultaat te zien is in Figuur 1.2b. We zien dat er meer nodes (verwerkende eenheden) zijn toegevoegd en dat elke node nu één oproep moet verwerken.

(a) De beginsituatie om het schalen van een database voor te stellen.

(b) Horizontaal schalen van 1.2a.

Figuur 1.2: Schematische voorstelling van horizontaal schalen

1.7 Gedistribueerde Databases

We merken eerst op dat we gedistribueerde database bespreken om later een duidelijk verschil te kunnen zien met NoSQL databases.

Gedistribueerde databases bestaan uit verschillende verwerkende eenheden, die we in deze thesis *nodes* noemen. Deze nodes zijn onderling met elkaar verbonden om communicatie mogelijk te maken. Het is belangrijk dat hoewel we met gedistribueerde databases te maken hebben, we toch met het relationeel model werken. De principes van ACID zoals besproken in Paragraaf 1.4 moeten in alle tijden blijven gelden.

De nodes beschikken allemaal over een database management system (DBMS). Er zijn verschillende manieren hoe deze DBMS'en met elkaar samenwerken en elke manier beschrijft een ander type van gedistribueerde databases [29].

1.7.1 Doelen

Een *gedistribueerde database* is één logische database die verspreid is over verschillende verwerkende eenheden (nodes). Deze nodes kunnen ver van elkaar liggen. Er kan bijvoorbeeld één node in New York staan, terwijl de andere node zich in Hasselt bevindt. Anderzijds kan een node ook op de onderste verdieping van een bedrijf gevestigd zijn, terwijl de andere node op de bovenste verdieping gevestigd is. Deze nodes zijn onderling verbonden via een communicatielijn. Dit is noodzakelijk aangezien ze in staat moeten zijn om veranderingen van de database aan elkaar door te geven. Daarnaast is het ook belangrijk om op te merken dat deze cluster van nodes transparant is voor de gebruikers. Met andere woorden lijkt het voor de gebruiker dat hij met *één* relationele database aan het werken is.

Het doel van gedistribueerde databases is dat verschillende mensen die zich op verschillende locaties bevinden, toch dezelfde database kunnen gebruiken. Een voorbeeld hiervan zijn grote bedrijven die op verschillende plaatsen in de wereld vestigingen hebben.

Voor de gebruikers van de database is het belangrijk dat ze geen last ondervinden van het feit dat de databases zich op verschillende plaatsen in de wereld bevinden. Daarom is het belangrijk dat er een soort van *locatie transparantie*

(Engels: *location transparency*) aanwezig is. Hiermee bedoelen we dat gebruikers van de database *niet* moeten weten waar de data effectief is opgeslagen als ze bijvoorbeeld een query uitvoeren. De database moet dus zelf ervoor zorgen dat een request van een gebruiker naar de juiste database gestuurd wordt. Een request vanuit België kan bijvoorbeeld doorgestuurd worden naar Spanje om een resultaat te kunnen krijgen. Zoals eerder vermeld, krijgen we op deze manier een systeem dat voor de gebruikers er uitziet als een gewone lokale database. Merk op dat het dan ook mogelijk is om tabellen met elkaar te joinen die niet in dezelfde lokale database zijn opgeslagen.

Een tweede belangrijk punt bij gedistribueerde databases is *lokale autonomie* of in het Engels *local autonomy*. Dit wil zeggen dat elke node in het gedistribueerd systeem toch op zichzelf moet kunnen werken als er op een of andere manier geen connectie meer mogelijk is tussen de verschillende nodes. Dus elke node moet zelf transacties kunnen uitvoeren, logs bijhouden en recoveren als er een fout optreedt. Op deze manier wordt er geen coördinatie meer georganiseerd door een master node [29].

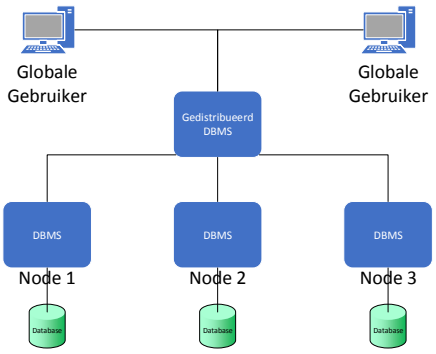
1.7.2 Twee Soorten Gedistribueerde Databases

Er zijn twee verschillende soorten gedistribueerde databases. We hebben enerzijds de *homogene gedistribueerde databases* en anderzijds de *heterogene gedistribueerde databases*.

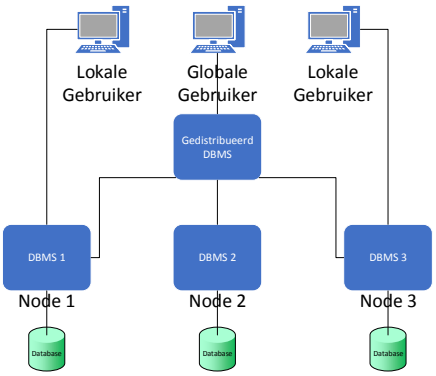
De homogene databases worden gekenmerkt door het feit dat op alle nodes dezelfde DBMS draait. Dit soort databases kan ook nog worden onderverdeeld in enerzijds de *autonome databases* en anderzijds de *niet-autonome databases*. Het verschil tussen deze twee mogelijkheden is dat deze op een andere manier data en updates verwerken. Bij de autonome databases werken alle DBMS'en onafhankelijk van elkaar. Met andere woorden zorgen alle DBMS'en zelf voor het verspreiden van hun eigen updates en data. De niet-autonome databases werken daarentegen met een centrale DBMS die als master node wordt aanzien. Alle updates en data passeren dus eerst via deze node en deze zal vervolgens de updates en data verspreiden naar de nodes in het netwerk. Een voorbeeld van een (niet-autonoom) homogeen systeem zien we in Figuur 1.3. We zien een centrale gedistribueerde database die voor de coördinatie zorgt en beschikt over het globale schema. De data wordt vervolgens verspreid over de verschillende nodes. Op deze manier hebben we geen exclusieve lokale data en zien we in elke node dezelfde data. Tenslotte zien we dat elke node dezelfde DBMS gebruikt.

De heterogene databases hebben de mogelijkheid om met verschillende soorten DBMS'en te werken. Ook hier bestaan er verschillende soorten heterogene systemen. We bespreken deze echter niet omdat dit ons te ver zou leiden. Een mogelijk voorbeeld van een heterogeen systeem zien we in Figuur 1.4. We zien opnieuw dat de data wordt verdeeld over de verschillende nodes. Het grote verschil ten opzichte van Figuur 1.3 is het feit dat we hier met verschillende DBMS'en te maken hebben. Een ander verschil is dat lokale gebruikers ook gebruik kunnen maken van de database. Deze toegang moet bijgevolg niet gaan via de gedistribueerde database.

Er moet altijd een keuze gemaakt worden tussen een homogeen en een heterogeen systeem om mee te werken. In veel bedrijven is het moeilijk om een homogeen systeem op te leggen. Het probleem bij heterogene systemen is het feit dat ze moeilijker te onderhouden zijn [29].



Figuur 1.3: Schematische voorstelling van een gedistribueerd homogeen systeem.



Figuur 1.4: Schematische voorstelling van een gedistribueerd heterogeen systeem.

1.7.3 Synchroon en Asynchroon

Bij het kiezen van een gedistribueerd systeem is er een belangrijke keuze tussen enerzijds een *synchroon systeem* en anderzijds een *asynchroon systeem*. In een synchroon systeem wordt alle data in het netwerk continu up to date gehouden. Op deze manier kunnen gebruikers op eender welk tijdstip en op eender welke plaats de data queryen met de garantie dat ze altijd en overal dezelfde data terugkrijgen. Met andere woorden wordt bij een update van de data, deze verandering onmiddellijk verder gepropageerd naar alle andere kopies van deze data. Als dit niet lukt, wordt de operatie stopgezet. Op deze manier wordt data integriteit gegarandeerd en is het zoeken naar de nieuwste kopie van de data niet moeilijk aangezien elke node de nieuwste waarde bevat. Het grote nadeel is dat dit soort systeem zeer traag kan worden, aangezien er altijd moet gecontroleerd worden of een update effectief succesvol is doorgestuurd naar de rest van het netwerk.

Bij asynchrone systemen worden veranderingen lokaal onmiddellijk toegepast, maar het doorsturen naar de andere nodes wordt niet onmiddellijk uitgevoerd. Het probleem hierbij is dat we in sommige gevallen met inconsistente

data te maken hebben als een node bijvoorbeeld gequeried wordt die nog niet geüpdatet is [29].

1.7.4 Technieken voor Gedistribueerde Databases

In deze Paragraaf bespreken we enkele belangrijke technieken die gebruikt worden bij het ontwerpen en ontwikkelen van gedistribueerde databases. We bespreken in deze Paragraaf de volgende onderwerpen:

- data replicatie
- horizontaal partitioneren
- verticaal partitioneren

Data Replicatie

De eerste techniek die we bespreken is *data replicatie*. Deze techniek wordt enerzijds gebruikt om de data effectief te verdelen over de verschillende nodes en anderzijds ook voor de tolerantie van fouten. Data replicatie wil simpelweg zeggen dat we exacte kopies van relaties opslaan in verschillende nodes. De personeelsrelatie uit Paragraaf 1.3.2 kan zowel in de node van België als die van Spanje worden opgeslagen. Als we ervoor zorgen dat elke relatie opgeslagen is in alle nodes, spreken we van een *volledige replicatie*. Hoewel dit een ideale oplossing lijkt, is dit in de praktijk (bij grote bedrijven) niet haalbaar.

Data replicatie biedt een aantal voordelen. Het belangrijkste voordeel is dat dezelfde data op verschillende plaatsen is opgeslagen en dat de data daarom lokaal snel toegankelijk is. Daarnaast biedt het ook het voordeel dat bij het uitvallen van een node, de data toch kan geraadpleegd worden door gebruik te maken van een andere node.

Daarnaast zijn er ook enkele nadelen. Zo moet elke node minstens evenveel opslagruimte hebben, aangezien ze dezelfde data moeten opslaan. Het grootste probleem heeft echter te maken met het telkens opnieuw kopiëren van de data naar de verschillende andere nodes. Dit is een heel complexe operatie waarbij vooral rekening moet gehouden worden met het feit of de data al dan niet consistent is.

Data replicatie wordt daarom meestal gebruikt in systemen die vooral *read-only* zijn. Op deze manier kunnen verschillende gebruikers de data lezen, maar zal de data nooit drastisch veranderen, waardoor de overhead van het kopiëren van de data voor veel minder problemen zorgt [29].

Naast deze algemene techniek bestaan er nog een aantal specifieke manieren om data te repliceren. Het bespreken van al deze technieken zou ons te ver leiden. Voor meer informatie verwijzen we naar [29].

Horizontaal Partitioneren

Horizontaal partitioneren is een techniek waarbij we rijen van een relatie opsplitsen en verdelen over verschillende nodes. Met andere woorden bevat elke node een deel van de relatie. Het gemakkelijkste om deze techniek uit te leggen is door gebruik te maken van een voorbeeld. We geven in Tabel 1.7 een eenvoudig voorbeeld over werknemers. In deze relatie staan de ID's, de namen, de locatie waar ze werken en het loon.

ID	Naam	Locatie	Loon
6435	René	België	2400
5849	Miguel	Spanje	3100
9302	Thomas	België	3400
1234	Els	België	2200
5632	Peter	België	3000
9847	Carlos	Spanje	4000

Tabel 1.7: Voorbeeldrelatie van werknemers.

Stel dat we nu een afdeling hebben in België en eentje in Spanje. We kunnen dan bijvoorbeeld Tabel 1.7 opsplitsen over de nodes van deze twee afdelingen aan de hand van de locatie waar ze werken. Dit zien we in Tabel 1.8 en Tabel 1.9.

ID	Naam	Locatie	Loon
6435	René	België	2400
9302	Thomas	België	3400
1234	Els	België	2200
5632	Peter	België	3000

Tabel 1.8: Voorbeeldrelatie van werknemers voor de locatie België.

ID	Naam	Locatie	Loon
5849	Miguel	Spanje	3100
9847	Carlos	Spanje	4000

Tabel 1.9: Voorbeeldrelatie van werknemers voor de locatie Spanje.

Het voordeel van deze manier van werken is dat we op deze manier de relaties verdelen over de nodes waar de data het meest wordt gebruikt. De data over België zal waarschijnlijk het meest worden gebruikt in België, terwijl de data over Spanje het meest zal gebruikt worden in Spanje. Met andere woorden kunnen deze queries zonder problemen lokaal worden uitgevoerd. Als er in sommige gevallen in België toch data nodig is van Spanje, zal de query moeten worden doorgestuurd naar Spanje om daar uitgevoerd te worden.

Het grote voordeel hiervan is dat de data efficiënter gebruikt kan worden. De nadelen van deze manier van werken is dat bij het falen van een node, de data niet meer bereikt kan worden en dat er een groot verschil kan zijn in toegangstijden aangezien er enerzijds lokaal en anderzijds op een externe node gequeried kan worden [29].

Verticaal Partitioneren

Bij *verticaal partitioneren* verdelen we niet de rijen van relaties over verschillende nodes, maar wel de kolommen. Bij het opsplitsen van deze relaties is het belangrijk dat beide opsplitsingen een gemeenschappelijk attribuut behouden zodat de originele tabel altijd terug heropgebouwd kan worden. Ook dit kunnen we best illustreren aan de hand van een voorbeeld.

Stel dat we over een database beschikken die gebruikt wordt voor het maken van auto's. Stel dat dit bedrijf twee afdelingen heeft, namelijk een afdeling om de auto mechanisch in elkaar te zetten en een afdeling om de auto af te werken. In Tabel 1.10 zien we een mogelijke database voor dit bedrijf.

ID	**Merk**	**Type**	**Motor**	**Afwerking**	**Kleur**
4356	Audi	A5	diesel	leer	zwart
6478	Volkswagen	Jetta	benzine	stof	blauw
2349	Audi	A4	diesel	stof	wit
9283	Volkswagen	Passat	diesel	leer	grijs
5544	Audi	Q5	diesel	leer	wit
8783	Audi	A3	benzine	stof	rood

Tabel 1.10: Voorbeeldrelatie van een auto fabriek.

Merk op dat de afdeling "Mechaniek" geen interesse heeft in de kleur van de auto en de afwerking van de zetels. Het enige waar deze afdeling interesse in heeft, is hoe de auto in elkaar moet gezet worden. Dit geldt ook voor de afdeling "Afwerking". Zij zijn immers niet geïnteresseerd in hoe de auto in elkaar is gezet. Stel nu dat beide afdelingen zich op een verschillende locatie bevinden (bijvoorbeeld in een andere stad of een ander land). We kunnen dan de database opsplitsen in de informatie die voor beide afdelingen nuttig zijn. Een voorbeeld van de "Mechaniek" afdeling zien we in Tabel 1.11 en een voorbeeld van de afdeling "Afwerking" zien we in Tabel 1.12.

ID	**Merk**	**Type**	**Motor**
4356	Audi	A5	diesel
6478	Volkswagen	Jetta	benzine
2349	Audi	A4	diesel
9283	Volkswagen	Passat	diesel
5544	Audi	Q5	diesel
8783	Audi	A3	benzine

Tabel 1.11: Voorbeeldrelatie van een auto fabriek voor de afdeling "Mechaniek".

ID	**Afwerking**	**Kleur**
4356	leer	zwart
6478	stof	blauw
2349	stof	wit
9283	leer	grijs
5544	leer	wit
8783	stof	rood

Tabel 1.12: Voorbeeldrelatie van een auto fabriek voor de afdeling "Afwerking".

Merk op dat we het ID in beide tabellen bijhouden, op deze manier kunnen we altijd de originele tabel heropbouwen door ze met elkaar te joinen. De voor- en nadelen van deze techniek kunnen we vergelijken met de voor- en nadelen van

horizontaal partitioneren uit Paragraaf 1.7.4. Merk wel op dat het samenvoegen van informatie moeilijker is bij verticaal partitioneren omdat we dan de keys met elkaar moeten vergelijken om de relaties te kunnen joinen [29].

1.8 Conclusie

In dit Hoofdstuk zijn we eerst kort ingegaan op de geschiedenis van de relationele databases. Vervolgens hebben we de basis overlopen van deze databases. We hebben enerzijds de terminologie toegelicht die gebruikt wordt bij relationele databases. Anderzijds zijn we dieper ingegaan op het ACID acroniem. Dit acroniem zorgt dankzij de vier eigenschappen atomair, consistent, geïsoleerd en duurzaam dat relationele databases heel betrouwbaar zijn. Het ACID principe speelt nog een belangrijke rol in het vervolg van de thesis als we bijvoorbeeld de design principes van NoSQL databases bespreken.

Ten slotte zijn we dieper ingegaan op gedistribueerde databases. Hierbij is het belangrijk om te onthouden dat we te maken hebben met verschillende nodes die met elkaar communiceren en dat er grote problemen kunnen ontstaan wanneer zo een communicatielijn uitvalt.

Hoofdstuk 2

CAP

2.1 Inleiding

Als we met *zeer schaalbare systemen* (zoals bijvoorbeeld een webservice) willen werken, is het gebruik van het eerder besproken acroniem ACID moeilijk. Brewer kwam daarom met drie eigenschappen waaraan een zeer schaalbaar systeem zou moeten voldoen. Deze drie eigenschappen vormen het acroniem *CAP* [13]. Met zeer schaalbare systemen bedoelen we bijvoorbeeld het huidige web 2.0. of grote NoSQL systemen die gebruikt worden door Google. CAP staat voor de volgende drie systeemeisen:

1. consistentie (Engels: consistency)
2. beschikbaarheid (Engels: availability)
3. partitie tolerantie (Engels: partition tolerance)

Brewer besprak de CAP theorie[1] in zijn keynote in 2000. Het was een reactie op de ontwerpen van geclusterde databases uit de jaren '90 [57].

Brewer beweert dat bij het ontwerpen van schaalbare gedistribueerde systemen slechts rekening kan gehouden worden met twee van de bovenstaande systeemeisen. We moeten met andere woorden telkens één systeemeis negeren of minder streng naleven.

Over de CAP theorie zijn er zeer veel bronnen ter beschikking en het is eveneens ook een groot discussiepunt op het internet. De oorzaak hiervan is dat CAP een heel high level model is met veel assumpties.

In dit Hoofdstuk bespreken we eerst deze drie systeemeisen zodat we een goed beeld krijgen van wat hier precies mee bedoeld wordt. Vervolgens geven we eerst een algemeen intuïtief bewijs van de CAP theorie, gevolgd door een formeel bewijs voor synchrone en asynchrone systemen. Daarna bespreken we enkele diepere inzichten omtrent de CAP theorie.

Ten slotte gaan we dieper in op het kopiëren van data en hoe er moet omgegaan worden met partities.

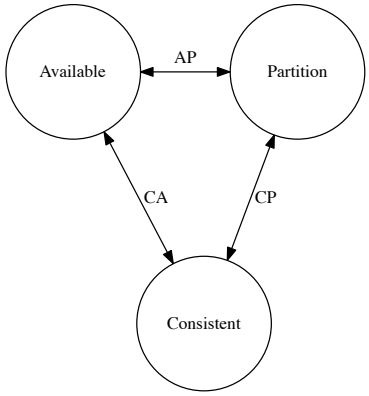
[1] Het was oorspronkelijk een vermoeden.

2.2 De Drie Systeemeisen

Zoals eerder besproken, kan een gedistribueerd systeem slechts voldoen aan twee van de drie systeemeisen. We kunnen dus drie verschillende soorten combinaties creëren [1]:

1. CA: dit is een consistent en beschikbaar systeem.
2. CP: dit is een consistent en partitie tolerant systeem.
3. AP: dit is een beschikbaar en partitie tolerant systeem.

Een schematische voorstelling hiervan is te zien in Figuur 2.1.



Figuur 2.1: Schematische voorstelling CAP theorie.

Om dit beter te begrijpen, bespreken we eerst de betekenis van de verschillende systeemeisen en lichten we ze toe met een voorbeeld.

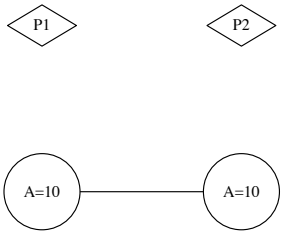
2.2.1 Consistentie

Met *consistentie* wordt bedoeld dat als we data opvragen, we altijd de juiste data terugkrijgen [13]. In Figuur 2.2 zien we een eenvoudig voorbeeld om dit te illustreren. In dit voorbeeld stellen we een persoon (gebruiker) voor door een ruit en een cirkel wordt gebruikt om een node (dit is opnieuw een verwerkende eenheid) aan te duiden. Verder worden gebruikers en nodes verbonden door een gerichte boog. Een boog richting een node wil zeggen dat een gebruiker een request of een transactie naar een node stuurt en een boog richting de gebruiker geeft aan dat hij een antwoord krijgt van een node. Ten slotte worden de nodes verbonden door een niet gerichte boog, deze boog wil zeggen dat er een communicatielijn is tussen twee nodes. Om een gebroken communicatielijn aan te duiden gebruiken we een stippellijn.

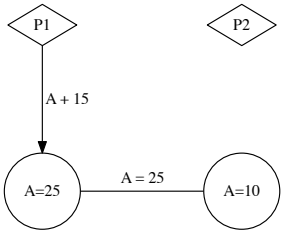
In Figuur 2.2a zien we dat er twee personen gebruik maken van een systeem dat onderverdeeld is in twee nodes. Initieel bevindt het systeem zich in een consistente staat. Alle nodes bevatten namelijk de waarde 10.

In Figuur 2.2b verhoogt persoon 1 de waarde van A met 15. In de eerste node zal A onmiddellijk worden aangepast naar de waarde 25. Om consistent te zijn, moeten vanaf dit moment alle andere nodes in het systeem de waarde 25 teruggeven als de waarde van A wordt opgevraagd.

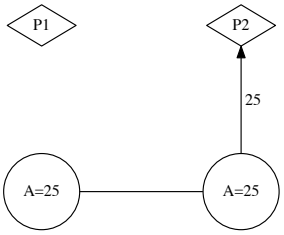
In Figuur 2.2c zien we dat de tweede node de waarde van A aanpast en zo ook de juiste informatie aan persoon 2 kan geven.



(a) Beginsituatie: persoon 1 en persoon 2 maken beiden gebruik van een systeem dat bestaat uit twee nodes.



(b) Persoon 1 verandert de waarde van A in de eerste node.



(c) Persoon 2 vraagt de waarde van A op bij de tweede node.

Figuur 2.2: Voorbeeld om consistency uit te leggen.

Zoals eerder vermeld is het belangrijk dat alle nodes de laatst geschreven waarde teruggeven. Vanaf dat moment kunnen we ervan uitgaan dat de eis van consistentie is vervuld. Als persoon 2 de waarde 10 zou teruggekregen hebben, is de eis van consistentie niet voldaan.

Het is duidelijk dat consistentie sterk afhankelijk is van het soort applicatie we gebruiken. Er zijn systemen waar consistentie heel belangrijk is en er zijn systemen waar het van minder belang is. Een voorbeeld van waar er minder aandacht wordt geschonken aan consistentie is YouTube. YouTube houdt bij hoeveel keer een bepaalde video bekeken is. Stel dat persoon A en persoon B op tijdstip X het aantal views bekijken van dezelfde video, dan is de kans zeer klein dat ze exact hetzelfde aantal views uitkomen. Per seconde worden er namelijk enorm veel video's bekeken op YouTube en het aantal views varieert dus per seconde. Persoon A en persoon B zullen het niet erg vinden dat ze niet exact het juiste aantal views zien.

Als we nu bijvoorbeeld op Sherpa tickets willen kopen voor een voorstelling van Urbanus, dan is het exact aantal beschikbare plaatsen wel belangrijk. Stel dat persoon A het aantal beschikbare tickets opvraagt en hij ziet dat er nog één plaats beschikbaar is. Als hij eenmaal betaald heeft, blijkt echter dat er helemaal geen tickets meer beschikbaar waren. Het systeem was bijgevolg niet consistent toen het aantal tickets werd opgevraagd. Dergelijke systemen moeten wel een soort van locking aanbieden. We bedoelen hiermee dat als persoon A een optie op een ticket neemt, dit ticket dan voor een bepaalde periode gelockt wordt. Op deze manier is het niet mogelijk dat een persoon die sneller betaalt, het ticket eerder kan bemachtigen.

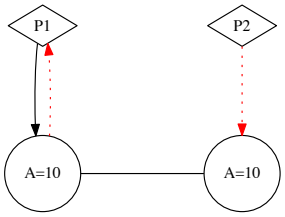
In het eerste voorbeeld zijn er weinig mensen die wakker liggen van het feit of een video 100 000 keer of 105 403 is bekeken. In het tweede geval daarentegen is het wel belangrijk dat we altijd de correcte informatie krijgen.

We merken ook op dat de C (consistentie) in ACID en CAP een andere betekenis heeft. In ACID verwijst consistentie naar het feit dat alle toepassingsafhankelijke constraints moeten nageleefd worden, terwijl bij de CAP theorie consistentie verwijst naar het feit dat overal in het systeem dezelfde data aanwezig moet zijn.

2.2.2 Beschikbaarheid

Beschikbaarheid is de eis dat op elke request van de client een response wordt teruggestuurd [13]. Als we bijvoorbeeld de tweets willen opvragen van Vincent Kompany, dan willen we effectief die tweets zien. Het zou niet gebruiksvriendelijk zijn als we een melding krijgen dat de tweets momenteel niet beschikbaar zijn.

We tonen dit aan met een eenvoudig voorbeeld. We hebben net zoals in Figuur 2.2a weer twee personen die elk gebruik maken van een verschillende node. In Figuur 2.3 zien we dat er problemen zijn met de beschikbaarheid. Persoon 1 kan zonder problemen een request versturen naar de eerste node, maar hij krijgt geen antwoord terug. In het tweede geval kan persoon 2 geen request versturen naar de node. In beide gevallen is er een probleem met de beschikbaarheid omdat de personen geen antwoord krijgen op hun request.



Figuur 2.3: Een voorbeeld waar beschikbaarheid niet in orde is.

Een andere visie op beschikbaarheid kan zijn dat het algoritme dat zorgt voor de verwerking en het ophalen van de informatie eindigt. Als we hier verder over nadenken, is dit een vrij zwakke eis. Een algoritme kan immers één milliseconde duren, maar anderzijds kan het ook tien minuten duren totdat het algoritme is

afgelopen. Tegenwoordig zal niemand tien minuten willen wachten totdat hij bijvoorbeeld tweets ziet. Anderzijds is het een sterke eis aangezien we zeker zijn dat het algoritme eindigt en we dus ook ooit de gewenste data krijgen (zelfs als het systeem faalt) [28]. We merken op dat deze definitie van beschikbaarheid gebruikt wordt om de CAP theorie te bewijzen in Paragraaf 2.3.4.

2.2.3 Partitie Tolerantie

Partitie tolerantie zorgt ervoor dat het systeem blijft werken zelfs als er één of meerdere nodes van het netwerk falen of als er berichten tussen de nodes verloren gaan [13].

Als een van de nodes faalt (en er dus een partitie optreedt) hebben we in principe de keuze uit drie mogelijkheden: in het eerste geval kan het systeem elke schrijfoperatie weigeren waardoor het systeem voor iedereen consistent en beschikbaar blijft bij leesoperaties, maar niet meer beschikbaar is voor schrijfoperaties.

In het tweede geval kunnen we schrijfoperaties toelaten aan een bepaalde partitie. Welk deel van de partitie schrijfoperaties mag afhandelen, is afhankelijk van de implementatie. In Paragraaf 7.6.1 over MongoDB geven we hierover een uitgebreid voorbeeld. De partitie die schrijfoperaties mag afhandelen, blijft bijgevolg consistent en beschikbaar bij lees- en schrijfoperaties. De andere partities zullen een eigen keuze moeten maken:

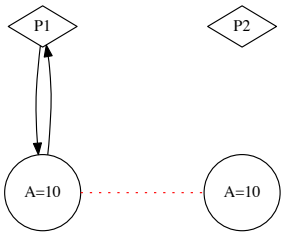
- Weiger schrijf- en leesoperaties om een strikte consistentie te bewaren.
- Laat leesoperaties toe zodat deze partitie *uiteindelijk consistent* (zie Paragraaf 3.3) kan zijn.

Als laatste kan het systeem ervoor kiezen om elke schrijf- en leesoperatie binnen elke partitie toe te laten. Op deze manier blijft het systeem beschikbaar, maar zal het consistentie moeten opofferen omdat schrijfoperaties in partities niet verder gecommuniceerd kunnen worden. We zien later in Paragraaf 3.3 dat er dan een andere vorm van consistentie wordt geïntroduceerd, namelijk uiteindelijke consistentie [11].

We illustreren het falen van de communicatie tussen twee nodes aan de hand van een voorbeeld. In Figuur 2.4 zien we dat de communicatielijn tussen de eerste en de tweede node verbroken is. Ze kunnen met andere woorden geen updates meer naar elkaar sturen. Wat we wel zien, is dat persoon 1 zonder problemen informatie kan vragen aan de eerste node. Het systeem blijft werken, ook al is het systeem gepartitioneerd.

2.3 Bewijzen van de CAP Theorie

In deze Paragraaf bewijzen we de CAP theorie in verschillende omstandigheden. We bewijzen eerst intuïtief dat het niet mogelijk is om aan de drie voorwaarden tegelijkertijd te voldoen. Dit doen we om de achterliggende gedachte goed weer te kunnen geven. Vervolgens bewijzen we de CAP theorie formeel. We baseren ons op het bewijs van Lynch et al. [28]. In dit bewijs gebruiken we de volgende concrete definitie van consistentie, genaamd atomaire consistentie: hiermee wordt bedoeld dat er een totale orde is op de operaties. Op deze manier



Figuur 2.4: Een voorbeeld waar de communicatielijn tussen de twee nodes is verbroken.

lijkt het alsof de operaties sequentieel worden uitgevoerd op slechts één node. We eisen met andere woorden dat elke leesoperatie de waarde van de meest recente schrijfoperatie teruggeeft.

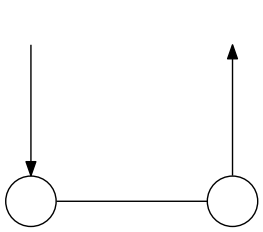
Daarnaast gebruiken we de definitie van beschikbaarheid die we hebben geïntroduceerd aan het einde van Paragraaf 2.2.2. We gaan er dus van uit dat het algoritme dat we gebruiken om gegevens op te halen altijd eindigt.

2.3.1 Intuïtief Bewijs

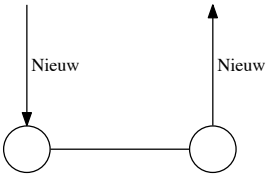
We leggen dit intuïtief bewijs uit aan de hand van enkele Figuren. In deze Figuren worden cirkels gebruikt om nodes (verwerkende eenheden) voor te stellen. Een gerichte boog duidt aan in welke richting een request of respons wordt gestuurd en een ongerichte boog stelt een communicatielijn tussen twee nodes voor. Dit is dezelfde notatie als we gebruikt hebben in Paragraaf 2.2.1. We beginnen met een situatie waarin we geen partities hebben. We zien de beginsituatie in Figuur 2.5a. We schrijven vervolgens een waarde in de linkse node en door het feit dat er communicatie mogelijk is tussen de twee nodes, zal bij een leesoperatie van de rechtse node de nieuwe waarde gelezen worden. Merk op dat we in deze situatie beschikbaar en consistent zijn, maar niet partitie tolerant. Als er geen communicatie mogelijk was, zou de nieuwe waarde immers niet gecommuniceerd kunnen worden naar de rechtse node.

Het volgende wat we bespreken zijn de situaties die kunnen voorkomen wanneer er wel een partitie aanwezig is. We zien opnieuw de beginsituatie in Figuur 2.6a. In Figuur 2.6b zien we dat er een partitie optreedt. Dit is aangeduid door de rode stippellijn tussen de twee nodes.

Er kunnen zich nu twee situaties voordoen. In de eerste situatie schrijven we een nieuwe waarde in de linkse node en lezen we de rechtse node uit. De rechtse node zal de oude waarde lezen aangezien er geen communicatie mogelijk is. Het systeem is met andere woorden beschikbaar en partitie tolerant, maar niet consistent. Deze situatie is voorgesteld in Figuur 2.7a. In de andere situatie schrijven we opnieuw een waarde in de linkse node, maar in deze situatie willen we consistentie waarborgen. De enige manier om dit te doen is door de rechtse node te laten wachten tot de partitie is genezen. Op deze manier is het systeem consistent en partitie tolerant, maar niet beschikbaar. Deze situatie is te zien in Figuur 2.7b.

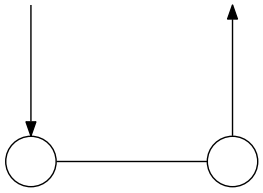


(a) De beginsituatie van het intuïtief bewijs.

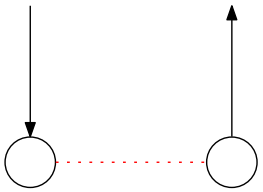


(b) Een nieuwe waarde wordt in de linkse node geschreven en deze waarde wordt ook gelezen uit de rechtse node.

Figuur 2.5: Situatie die kan voorkomen als er geen partitie aanwezig is.



(a) De beginsituatie van het intuïtief bewijs.



(b) Er ontstaat een partitie, de rode stippellijn geeft een probleem met de communicatielijn weer.

Figuur 2.6: Schematische voorstelling van het ontstaan van een partitie.

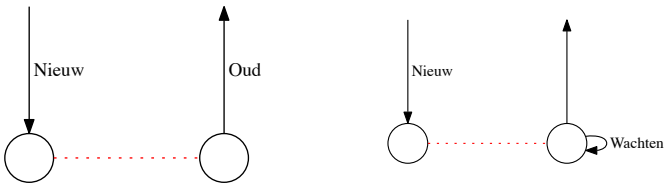
2.3.2 Formeel Bewijs: Asynchrone Systemen met Communicatieverlies

In *asynchrone systemen* is er geen klok ter beschikking om onderlinge synchronisatie tussen nodes te regelen. De enige manier om iets van elkaar te weten te komen, is door onderling berichten uit te wisselen en door zelf berekeningen te maken. Het grote probleem bij de afwezigheid van een klok is het feit dat nodes geen onderscheid kunnen maken tussen of hun bericht effectief verloren is gegaan of dat er veel vertraging op de communicatielijn zit [28].

Lynch et al. [28] construeerde hieruit de volgende stelling:

Stelling 1. *Het is niet mogelijk om in een asynchroon systeem een data object te implementeren met lees- en schrijfoperaties zodat dit object gelijktijdig de volgende eigenschappen heeft in alle faire uitvoeringen:*

- *beschikbaarheid*



(a) Een nieuwe waarde wordt in de linkse node geschreven en de oude waarde wordt uit de rechtse node gelezen.

(b) Een nieuwe waarde wordt in de linkse node geschreven. Om de consistentie te bewaren, zal de rechtse node wachten tot de partitie genezen is.

Figuur 2.7: Situaties die kunnen voorkomen wanneer er een partitie aanwezig is.

- *atomaire consistentie*

Met het oog op alle geldige uitvoeringen*, dus ook uitvoeringen waar onderlinge berichten verloren gaan. De implementatie is met andere woorden partitie tolerant [28].*

We geven eerst de opstelling van ons bewijs, hierin bespreken we onder andere welke assumpties we aannemen. Om deze opstelling goed te begrijpen geven we vervolgens een voorbeeldsituatie. Ten slotte bewijzen we Stelling 1 formeel.

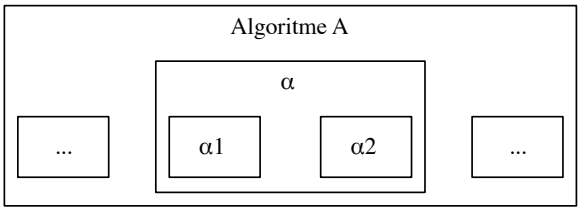
Opstelling

We bewijzen deze stelling door contradictie. We proberen een situatie te creëren die onmogelijk is in onze opstelling. Voor onze opstelling gebruiken we een algoritme A. Dit algoritme wordt gekenmerkt door de drie eisen van de CAP theorie, het is dus (atomair) consistent, beschikbaar en partitie tolerant. We proberen aan de hand van dit algoritme een situatie te creëren opdat deze een inconsistent resultaat zal teruggeven, waardoor we een contradictie bekomen.

In ons bewijs maken we gebruik van een systeem dat uit minstens twee nodes bestaat. Merk op dat we in de gebruikte Figuren effectief twee nodes gebruiken om het overzicht te bewaren. Vervolgens delen we de nodes van ons netwerk (in ons geval $\{N_1, N_2\}$) op in twee disjuncte, niet lege subsets $\{N_1\}$ en $\{N_2\}$. Beide nodes bevinden zich in het begin van het bewijs in een consistente staat. Ze bevinden zich met andere woorden in exact dezelfde staat, namelijk w_0.

Algoritme A bestaat uit verschillende operaties. Met operaties bedoelen we bijvoorbeeld rekenkundige bewerkingen, leesoperaties en schrijfoperaties. In onze opstelling is er een uitvoering α_1 die een stukje van het algoritme A voorstelt waarin er een waarde w_1, waar $w_1 \neq w_0$ geschreven wordt naar N_1. Uitvoering α_1 eindigt met het afsluiten van de schrijfoperatie. Daarnaast bestaat er ook een andere uitvoering van het algoritme A, namelijk α_2. Uitvoering α_2

leest een waarde uit N_2 en eindigt met het afsluiten van deze leesoperatie. We benadrukken dat α_1 en α_2 slechts eindige uitvoeringen zijn. Een voorstelling van de volledige situatie van algoritme A is te zien in Figuur 2.8.



Figuur 2.8: Een voorstelling van Algoritme A bij een asynchroon systeem.

Voorbeeld

Nu we de situatie geschetst hebben, geven we eerst een voorbeeld van hoe deze situatie zou verlopen onder ideale omstandigheden, dus wanneer er niets fout gaat in de communicatie tussen N_1 en N_2. In dergelijke situaties verloopt dit proces zoals weergegeven in Figuur 2.9.

In Figuur 2.9a zien we de beginopstelling. Merk op dat de communicatie deze keer met twee gerichte bogen wordt weergegeven om goed te kunnen aangeven welke node welke data communiceert.

Vervolgens wordt in Figuur 2.9b weergegeven hoe uitvoering α_1 de waarde w_0 overschrijft met de waarde w_1.

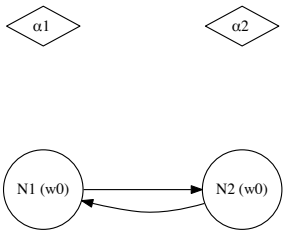
De waarde is nu aangepast in N_1 en dit moet gecommuniceerd worden aan de andere nodes in het netwerk. Vervolgens geeft N_1 het bevel aan N_2 om zijn waarde ook aan te passen naar w_1. Dit zien we gebeuren in Figuur 2.9c.

Als laatste kan uitvoering α_2 nu de waarde opvragen bij N_2. Zoals eerder vermeld verloopt alles goed en zal de waarde w_1 teruggegeven worden door N_2 [13]. Een voorstelling hiervan zien we in Figuur 2.9d.

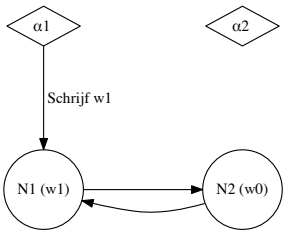
We weten nu hoe het proces zou verlopen als er geen fouten gebeuren. We bewijzen nu deze stelling door contradictie. Zoals reeds vermeld, creëren we een situatie waarin we een inconsistent resultaat bekomen. We gaan, in tegenstelling tot ons eerder voorbeeld, er niet vanuit dat alles correct verloopt. We stellen dat alle berichten tussen N_1 en N_2 verloren gaan. De twee nodes kunnen dus niet met elkaar communiceren. We zien deze situatie in Figuur 2.10.

Bewijs

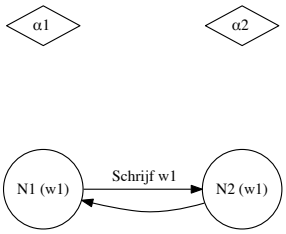
Bewijs. Zij α_1 en α_2 de eindige uitvoeringen zoals hierboven gedefinieerd. We gaan nu een nieuwe uitvoering α opstellen, waarin we eerst uitvoering α_1 laten werken en daarna uitvoering α_2. Het bewijs verloopt dan als volgt: stel dat uitvoering α_1 begint met zijn schrijfoperatie van w_1, waar $w_1 \neq w_0$. We nemen aan dat tijdens deze schrijfoperaties geen enkele andere operatie wordt ingediend



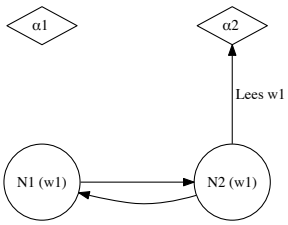
(a) De beginopstelling van het voorbeeld.



(b) De waarde in N_1 wordt overschreven met w_1.

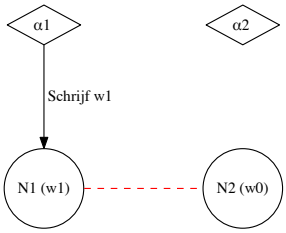


(c) De communicatie tussen N_1 en N_2.



(d) De leesoperatie op N_2.

Figuur 2.9: Schematische voorstelling van het voorbeeld uit het formeel bewijs van asynchrone systemen met communicatieverlies.

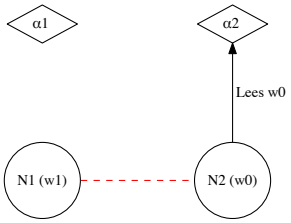


Figuur 2.10: Geen communicatie mogelijk tussen N_1 en N_2.

door andere clients bij N_1 of N_2. Zoals eerder vermeld weten we dat er geen communicatie mogelijk is tussen N_1 en N_2. Dankzij de eis van beschikbaarheid weten we echter zeker dat uitvoering α_1 zal eindigen.

Het volgende dat er gebeurt, is het starten van uitvoering α_2. Hier gelden dezelfde regels als bij de uitvoering α_1, namelijk dat er geen communicatie is tussen N_1 en N_2 en dat uitvoering α_2 het enige actieve proces is. Uitvoering α_2 leest de waarde van N_2 en ook nu zijn we zeker dat uitvoering α_2 zal eindigen dankzij de beschikbaarheidseis. Omdat er geen communicatie mogelijk was, zal

uitvoering α_2 de oude waarde w_0 lezen. We zien deze situatie in Figuur 2.11.



Figuur 2.11: Uitvoering α_2 leest w_0 van node N_2.

We zien in uitvoering α dat bij de leesoperatie van N_2, de oude waarde w_0 wordt gelezen in plaats van de nieuwe waarde w_1. We kunnen dus besluiten dat uitvoering α de eigenschap van atomaire consistentie heeft geschonden: de schrijfoperatie van de nieuwe waarde w_1 in N_1 kwam duidelijk vóór de leesoperatie in N_2, en toch geeft de leesoperatie nog de oude waarde w_0 terug. Bijgevolg, als het systeem beschikbaar en partitie tolerant is, dan zal het systeem geen atomaire consistentie kunnen garanderen.

We kunnen dus met zekerheid zeggen dat een algoritme A niet kan bestaan en daarmee hebben we Stelling 1 bewezen [28]. □

2.3.3 Formeel Bewijs: Asynchrone Systemen, zonder Communicatieverlies

We hebben de CAP theorie in Paragraaf 2.3.2 bewezen voor asynchrone systemen waarbij we rekening houden met alle geldige uitvoeringen. Dit wil dus ook zeggen dat we uitvoeringen moeten beschouwen waarbij berichten tussen twee nodes eventueel ook verloren kunnen gaan. Hieruit leiden we het volgende Gevolg af:

Gevolg 2. *Het is niet mogelijk om in een asynchroon systeem een data object te implementeren met lees- en schrijfoperaties zodat dit object gelijktijdig de volgende eigenschappen heeft in alle faire uitvoeringen: [28]:*

- *beschikbaarheid bij alle* geldige uitvoeringen*, zelfs waar berichten mogen verloren gaan*
- *atomaire consistentie bij* geldige uitvoeringen *waar geen berichten verloren gaan*

Bewijs Idee

Het idee achter dit bewijs steunt op het feit dat we in een asynchroon systeem in principe nooit weten of een bericht effectief verloren is gegaan of dat het enorm veel vertraging heeft in het netwerk. Als we daarom een algoritme zouden kunnen ontwikkelen dat atomaire consistentie zou garanderen in geldige uitvoeringen waar geen berichten verloren gaan, dan zou bijgevolg ook een algoritme

moeten bestaan dat atomaire consistentie garandeert in alle geldige uitvoeringen. We hebben echter in Stelling 1 bewezen dat een dergelijk algoritme niet bestaat. Dit leidt dus tot een contradictie.

Bewijs

Bewijs. Stel dat er een algoritme A bestaat dat altijd eindigt (beschikbaarheidseis). Daarnaast is het algoritme A ook atomair consistent bij geldige uitvoeringen waar geen berichten verloren gaan. Gecombineerd met Stelling 1, moet er dus een geldige uitvoering α bestaan waarin berichten verloren gaan, en waarin A geen atomaire consistentie kan waarborgen. Dit wil zeggen dat op *een* moment in α, algoritme A een niet atomaire respons teruggeeft.

Stel dat uitvoering α' een prefix is van uitvoering α, dat eindigt met een dergelijke ongeldige respons. Het volgende wat we kunnen doen is uitvoering α' uitbreiden naar uitvoering α'' waarin alle berichten aankomen. We hebben van α'' een geldige uitvoering gemaakt waarin alle berichten aankomen, maar het probleem is dat deze uitvoering nog steeds niet atomair is. Dit komt omdat we nog steeds met de ongeldige respons van uitvoering α' te maken hebben. Het is dus niet mogelijk om zo een algoritme A te construeren en bijgevolg is Gevolg 2 bewezen [28]. □

2.3.4 Formeel Bewijs: Synchroon Systeem

We hebben de CAP theorie bewezen voor asynchrone systemen, maar in de praktijk zijn er ook heel veel synchrone systemen. Geldt de CAP theorie ook voor dergelijke systemen?

Met een *synchroon systeem* bedoelen we een systeem dat gebruik maakt van klokken. Met behulp van deze klokken kunnen nodes zelf beslissen/schatten of een bericht verloren is gegaan en dit bericht dus bijvoorbeeld opnieuw gestuurd moet worden. Dit in tegenstelling tot asynchrone systemen, waar nodes in feite nooit weten of hun bericht effectief aankomt of gewoon veel vertraging heeft.

In het bewijs gebruiken Lynch et al. [28] een *gedeeltelijk synchroon systeem.* Elke node heeft zijn eigen klok die wordt verhoogd met dezelfde waarde. De klokken onderling zijn niet gesynchroniseerd. Hiermee bedoelen we dat de klokken op hetzelfde tijdstip een andere waarde kunnen weergeven, de klokken werken dus als een persoonlijke timer. Op deze manier kunnen nodes een bericht sturen en, als ze na X klokstappen nog steeds geen antwoord hebben gekregen, hetzelfde bericht opnieuw zenden. Op deze manier kunnen we toch een betrouwbaarder systeem creëren. We gebruiken een gedeeltelijk synchroon systeem omdat nodes op deze manier zelf kunnen schatten hoe sterk een bericht vertraagd is [28].

Maar ook met een synchroon systeem is het niet mogelijk om aan de drie eigenschappen van CAP te voldoen en dat bewijzen we aan de hand van de volgende stelling:

Stelling 3. *Het is niet mogelijk om in een gedeeltelijk synchroon systeem een data object te implementeren met lees- en schrijfoperaties zodat dit object gelijktijdig de volgende eigenschappen heeft in alle faire uitvoeringen:*

- *beschikbaarheid*

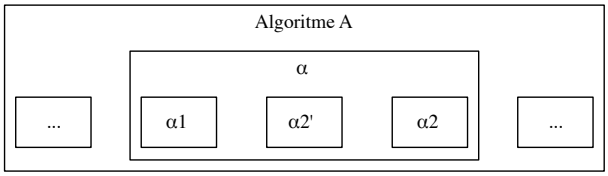
- *atomaire consistentie*

Met het oog op alle geldige uitvoeringen*, dus ook uitvoeringen waar onderlinge berichten verloren gaan. De implementatie is met andere woorden partitie tolerant [28].*

Opstelling

Het bewijs verloopt min of meer analoog aan het bewijs uit Paragraaf 2.3.2. We hebben opnieuw een opstelling waarin minstens twee nodes in het systeem zitten. Deze nodes delen we opnieuw op in twee niet lege, disjuncte subsets $\{N_1\}$, $\{N_2\}$. Ook deze stelling bewijzen we door middel van contradictie. We construeren opnieuw een situatie waarin we een inconsistente waarde lezen.

We creëren een algoritme A dat voldoet aan de drie eisen van de CAP theorie. Beschouw een uitvoering α_1 waarin α_1 een stukje van het algoritme is dat een waarde w_1 zal schrijven in N_1 waar $w_1 \neq w_0$. Beschouw ook een uitvoering α_2 waarin een leesoperatie wordt uitgevoerd in N_2. Daarnaast construeren we ook uitvoering α_2'. Dit stukje zal minstens zolang wachten als de totale uitvoeringstijd van uitvoering α_1. Uitvoering α construeren we opnieuw door uitvoering α_1 te groeperen samen met uitvoering α_2' en uitvoering α_2. Hierbij wordt eerst α_1 uitgevoerd, waarop nog een deeluitvoering α_2' wordt geplaatst en dan pas wordt α_2 uitgevoerd. Op deze manier garanderen we dat uitvoering α_1 volledig is uitgevoerd en zijn we dus zeker dat de nieuwe waarde geschreven is in N_1. Een beschrijving van algoritme A is te zien in Figuur 2.12



Figuur 2.12: Een voorstelling van algoritme A bij een synchroon systeem.

Bewijs

Bewijs. We maken een uitvoering α als volgt: eerst voeren we α_1 uit, en daarbovenop plaatsen we de uitvoering α_2' gevolgd door uitvoering α_2. De stilte in de prefix α_2' conflicteert niet met uitvoering α_1, maar zorgt er dus wel voor dat uitvoering α_2 pas wordt uitgevoerd na α_1. We merken op dat ook in dit bewijs er geen communicatie mogelijk is tussen N_1 en N_2, met als gevolg dat er geen updates kunnen worden gezonden. Merk op dat de nodes in N_2 zich hetzelfde zullen gedragen in uitvoering α als in de uitvoering $\alpha_2'+\alpha_2$. Daardoor zal tijdens uitvoering α de leesoperatie binnen N_2 dus de oude waarde w_0 teruggeven.

Omdat de schrijfoperatie binnen N_1 dus duidelijk eerst gebeurde in uitvoering α, is de eigenschap van atomaire consistentie geschonden.

Bijgevolg is deze stelling bewezen door contradictie [28].

□

2.3.5 Opmerking

We merken op dat het niet mogelijk is om een gevolg uit Stelling 3 te trekken zoals we gedaan hebben in Gevolg 2. Zoals eerder vermeld is de zwakte van een asynchroon systeem het feit dat we nooit met zekerheid weten of een bericht verloren is gegaan in het netwerk. Er bestaan echter wel gedeeltelijk synchrone algoritmes die enerzijds atomaire consistentie garanderen in situaties waarin alle berichten worden afgeleverd (bijvoorbeeld systemen zonder partities) en anderzijds geven deze algoritmes enkel inconsistente data terug als er berichten verloren gaan. Een voorbeeld van zo een algoritme is het *gecentraliseerd protocol*. Dit algoritme werkt met een centrale node. Als er dus een lees- of schrijfoperatie wordt verstuurd naar een bepaalde node, wordt er een bericht gestuurd naar de centrale node. Deze centrale node verwerkt vervolgens de lees- of schrijfoperatie en stuurt het antwoord van de leesoperatie of een bevestiging van de schrijfoperatie terug naar de node. Om te beslissen of een bericht verloren is gegaan, gebruiken we de twee tijdparameters t_{msg} en t_{local}. De tijdparameter t_{msg} is de tijd waarin een bericht in normale omstandigheden zou moeten aankomen en t_{local} is de tijd die een node nodig heeft om een bericht te verwerken. Als er na $2 \times t_{msg} + t_{local}$ geen antwoord van de centrale node ontvangen is, wil dit zeggen dat het bericht verloren is gegaan. Er wordt dan een bericht naar de gebruiker gestuurd met de laatst gekende data van de node. In dit geval is de atomaire consistentie niet in orde [28].

2.4 Verdere Inzichten

Abadi [1] heeft enkele bemerkingen op de CAP theorie.

Zoals we ondertussen weten, zijn er drie eisen waar ons gedistribueerd systeem aan zou moeten voldoen, maar omwille van de CAP theorie kunnen we slechts aan twee van de drie eisen voldoen. We kunnen met andere woorden een consistent en beschikbaar systeem ontwikkelen dat niet partitie tolerant is (CA), een consistent en partitie tolerant systeem dat niet beschikbaar is (CP) of een beschikbaar en partitie tolerant systeem dat niet consistent is (AP).

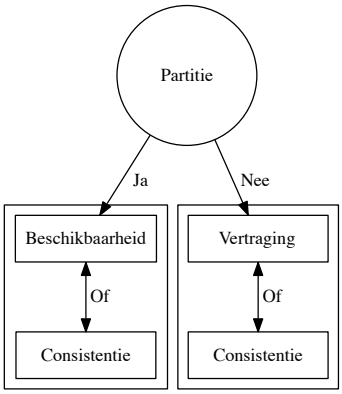
Als we CP beter bekijken, lijkt er een vreemde definitie te ontstaan: "consistent en partitie tolerant, maar niet beschikbaar". Wil dit zeggen dat het systeem nooit beschikbaar is? Het antwoord hierop is natuurlijk nee, want met dergelijke systemen kunnen we niets doen. We zouden namelijk nooit informatie kunnen opvragen. Maar wat wil dit dan wel zeggen? Deze definitie betekent dat we beschikbaarheid opofferen als er een partitie optreedt in een netwerk. Dit wil zeggen dat de beschikbaarheid en consistentie in de CAP theorie een *asymmetrische* rol vervullen. Dit komt omdat systemen waar consistentie genegeerd wordt, *neigen om deze consistentie altijd op te offeren* en niet alleen als er een partitie gevormd wordt. Dit is in tegenstelling tot beschikbaarheid dat, zoals eerder besproken, enkel wordt opgeofferd als er een partitie ontstaat. Het waarom hiervan leggen we in het vervolg van deze Paragraaf uit.

Verder stelt Abadi [1] dat er geen/weinig verschil is tussen de CP en CA. We leggen uit waarom: zoals eerder besproken, offert CP beschikbaarheid enkel

op als er een partitie optreedt. Als we nu CA beter bekijken, zien we dat deze combinatie staat voor: "consistent en beschikbaar, maar niet partitie tolerant". We kunnen nu de vraag stellen wat er gebeurt als er wel degelijk een partitie optreedt. Het systeem is niet partitie tolerant, dus als er een partitie optreedt, zal het systeem stoppen met functioneren. Het systeem is vanaf nu dus niet meer beschikbaar, maar wel nog consistent. Dus we kunnen besluiten dat CA en CP min of meer hetzelfde zijn. Bij het optreden van een partitie zal in beide gevallen de consistentie eigenschap overleven. Volgens Abadi kunnen we dus maar tussen twee combinaties kiezen, namelijk CA/CP en AP.

Het laatste punt van kritiek van Abadi [1] gaat over de trade-off tussen consistentie en beschikbaarheid. In deze thesis hebben we het over NoSQL systemen. Dit zijn vaak gedistribueerde systemen, dus willen we zeker zijn dat ons systeem partitie tolerant is. We moeten met andere woorden altijd kiezen om consistentie te behouden of beschikbaarheid te behouden. In veel gevallen wordt consistentie opgeofferd voor beschikbaarheid omdat dit meestal belangrijker is. Dit geeft ons in feite een foutief beeld van de werkelijke bedoeling. Een belangrijke eis die niet in CAP voorkomt, is *vertraging* of in het Engels *latency*. We komen hier later in Paragraaf 2.4.2 op terug. Een systeem kan perfect beschikbaar zijn, maar hoelang duurt het voordat we ons antwoord krijgen? Een systeem kan consistent zijn, maar hoelang duurt het voordat alle nodes gesynchroniseerd zijn? Dit zijn allemaal vragen die te maken hebben met vertraging en daarom ook heel belangrijk zijn voor deze thesis.

Volgens Abadi [1] past het *PACELC* acroniem beter in dit verhaal: als er een partitie (**p**artition) is, kiezen we dan tussen beschikbaarheid (**a**vailibility) of consistentie (**c**onsistency) en anders (**e**lse) als er geen partitie optreedt, kiezen we dan tussen vertraging (**l**atency) of consistentie (**c**onsistency)? Een schematische voorstelling van dit acroniem is te vinden in Figuur 2.13.



Figuur 2.13: Schematische voorstelling PACELC.

Systemen waar een partitie in voorkomt, kiezen in veel gevallen voor beschikbaarheid in de consistentie/beschikbaarheidsafweging. Deze keuze wordt verdergezet als er geen partitie voorkomt in het systeem. In dergelijke gevallen

wordt er ook vaak snelheid verkozen boven consistentie. Met andere woorden wil dit zeggen dat we liever een snel, maar mogelijk inconsistent antwoord krijgen, dan een traag en consistent antwoord. Dit is exact de reden van het asymmetrische verband tussen consistentie en beschikbaarheid zoals we eerder vermeld hebben.

Voor meer informatie over de consistentie/vertraging trade-off verwijzen we naar Paragraaf 2.4.2

Naast problemen met de CAP theorie zijn er ook heel wat problemen die ontstaan door het verkeerd begrijpen van deze theorie. Hier gaan we in de volgende Paragraaf verder op in.

2.4.1 Misverstanden

We hebben nu al heel wat achtergrondinformatie over de CAP theorie, maar tot op de dag van vandaag is er nog steeds onduidelijkheid over deze theorie. We leggen in deze Paragraaf uit waarom. De CAP theorie laat uitschijnen dat we maar twee van de drie systeemeisen kunnen kiezen. De derde eis wordt met andere woorden niet uitgevoerd. Laten we de discussie toespitsen op het thesisonderwerp, namelijk NoSQL systemen. Dergelijke systemen staan bekend om hun grote schaalbaarheid. Deze schaalbaarheid bereiken ze door enorm veel systemen met elkaar te verbinden om zo een sterk systeem te bouwen. Het is duidelijk dat in dergelijke gedistribueerde systemen het van noodzakelijk belang is dat deze systemen partitie tolerant zijn. We willen zeker niet dat bij het uitvallen van één node, ons volledige systeem platligt [2]. De keuze die moet gemaakt worden is:

- Kiezen we voor een hoge consistentie?
- Kiezen we voor een hoge beschikbaarheid?

Deze keuze lijkt op het eerste zicht logisch, maar het is niet de partitie tolerantie *alleen* die de oorzaak is van de keuze, maar een combinatie van twee zaken:

- Is het systeem partitie tolerant?
- Treedt er een partitie op in het systeem?

De aanwezigheid van een partitie bepaalt met andere woorden of er een trade-off moet gebeuren tussen consistentie en beschikbaarheid. De afwezigheid van een partitie impliceert dat de drie voorwaarden kunnen voldaan zijn. Hierbij houden we onze bemerkingen van Paragraaf 2.4 in het oog, namelijk dat het opofferen van consistentie niet altijd te maken heeft met de trade-off tussen consistentie en beschikbaarheid, maar dat we ook rekening moeten houden met de vertraging die kan optreden in het systeem. De kans dat er een partitie optreedt, is afhankelijk van verschillende zaken, zoals [2]:

- Welke hardware gebruiken we?
- Is het een uitgestrekt netwerk of een lokale cluster?

2.4.2 Consistentie geeft Vertraging

Met *vertraging* bedoelen we de tijd die nodig is om bijvoorbeeld de data van een bepaalde node te synchroniseren en de data terug te geven. Deze willen we liefst zo klein mogelijk houden. Dat vertraging belangrijk is, is vrij duidelijk. Als we tegenwoordig surfen op het internet willen we dat alles *snel* geladen wordt. Met andere woorden willen we geen tien seconden wachten tot een webpagina geladen is, want dat is voor veel gebruikers al te lang. De aanwezigheid van dergelijke vertragingen op websites kan leiden tot het feit dat gebruikers deze website niet meer willen gebruiken.

Een achterliggende oorzaak van vertraging (bij bijvoorbeeld NoSQL databases) is het feit dat nodes onderling moeten communiceren om consistentie te kunnen garanderen. Als we een consistent systeem willen, moeten nodes op elk ogenblik proberen de meest recente data ter beschikking te hebben. Dit doen ze door telkens updates door te sturen naar elkaar en dit allemaal synchroniseren, kost tijd. Dit leidt dus tot vertraging.

We merken op dat beschikbaarheid en vertraging sterk aan elkaar gerelateerd zijn. Als we een systeem hebben met een heel lage beschikbaarheid, dan moeten we ook heel lang wachten op een antwoord. Dit is eigenlijk hetzelfde als zeggen dat het systeem een grote vertraging heeft.

De keuze tussen consistentie en vertraging hangt, zoals eerder vermeld, niet af van een mogelijke partitie en staat daarom eigenlijk volledig los van het CAP verhaal.

Daarnaast heeft de keuze tussen vertraging en consistentie ook een onderling verband. Het is logisch dat als we onze consistentie verhogen (bijvoorbeeld van soms consistent naar altijd consistent), onze vertraging ook groter zal worden. We moeten dus langer wachten op een antwoord omdat de consistentie in orde moet gebracht worden bij de nodes. Als we daarentegen de consistentie verlagen, moeten we ook minder lang wachten op een antwoord en zal de vertraging verlaagd worden.

Een ander belangrijk punt is het bereiken van beschikbaarheid. Als we dit willen bereiken, is het belangrijk dat er regelmatig kopies worden gemaakt van de data en dat deze worden doorgestuurd naar andere nodes. Als er nu bijvoorbeeld een node uitvalt, dan zal die data pas terug beschikbaar worden vanaf het moment dat deze terug actief wordt, tenzij deze node regelmatig kopies van zichzelf doorstuurt naar andere nodes. Bij het uitvallen van deze node kan er dan beroep gedaan worden op andere nodes. Hier merken we nog op dat, in tegenstelling tot de trade-off bij CAP, het ontstaan van een partitie deze keuze in werking zet. In de trade-off tussen consistentie en vertraging wordt deze keuze in werking gesteld door de *mogelijkheid* tot het ontstaan van een partitie [2].

De trade-off tussen consistentie en vertraging begint vanaf het moment dat een gedistribueerd systeem data wil kopiëren naar andere nodes. We bespreken deze trade-off voor drie mogelijkheden waarop nodes data kunnen kopiëren naar elkaar [2]:

- Het systeem verzendt de updates naar alle nodes tegelijk.
- Het systeem verzendt de updates eerst naar een master node, dit is een fysieke node die deze updates vervolgens verder stuurt naar de andere nodes.

- Het systeem verzendt de updates eerst naar een willekeurige node, die vervolgens de updates verder stuurt naar de andere nodes.

2.4.3 Updates naar Alle Nodes Tegelijk

We kunnen alle nodes tegelijk updaten door gebruik te maken van twee strategieën, namelijk door enerzijds een soort overeenkomstprotocol uit te voeren en anderzijds door dit niet te doen. De keuze die we hier maken zal weer bijdragen tot meer consistentie en meer vertraging of minder consistentie en minder vertraging.

Als we updates niet eerst door een overeenkomstprotocol laten verwerken, kunnen er problemen ontstaan. Dit probleem kan een sterke afwijking van de kopies zijn. Stel bijvoorbeeld dat verschillende clients tegelijk een bepaalde schrijfoperatie uitvoeren waardoor al deze updates tegelijk moeten worden verwerkt. Elke node kan een andere volgorde aannemen voor het updaten van de andere nodes, met als gevolg dat we een inconsistent resultaat krijgen.

Aan de andere kant kunnen we wel gebruik maken van een overeenkomstprotocol, maar dit heeft een kost. Door gebruik te maken van dergelijke protocollen is het mogelijk dat de nodes een overeenkomst maken in verband met de volgorde waarin ze de verschillende nodes updaten. Het resultaat hiervan is dat we meer consistentie bereiken, maar deze overeenkomst kost tijd. Dit leidt tot een verhoogde vertraging [2].

2.4.4 Updates Eerst naar een Master Node

Deze manier van updaten verloopt als volgt: als er een data element wordt gewijzigd op een bepaalde node, laat deze node aan de master node weten dat er een update gebeurd is. Merk op dat verschillende nodes updates zullen sturen naar de master node. De master node bepaalt vervolgens in welke volgorde alle nodes worden geüpdatet.

Ook hier zijn er verschillende mogelijkheden om dit te realiseren:

- synchroon kopiëren
- asynchroon kopiëren
- een combinatie van synchroon en asynchroon kopiëren

Synchroon Kopiëren

Bij het synchroon kopiëren zal de master node ervoor zorgen dat hij helemaal zeker is dat alle nodes correct zijn geüpdatet. Dit biedt een hoge consistentie, maar ook een verhoogde vertraging aangezien er veel berichten op en af moeten gestuurd worden om te weten te komen of alles goed verlopen is. Aangezien de master node ook wacht tot alle nodes geüpdatet zijn, zal de vertraging zo groot zijn als de traagste node [2].

Asynchroon Kopiëren

Net als bij het synchroon kopiëren, zal bij het asynchroon kopiëren de master node de updates verzenden naar de andere nodes. Vanaf het moment dat

hij één antwoord terug krijgt van één van de nodes, zal hij de update als voltooid beschouwen. Ondertussen worden de overige updates in de achtergrond uitgevoerd.

Het is duidelijk dat dit niet zonder risico is. De trade-off tussen consistentie en vertraging hangt af van hoe het systeem omgaat met leesoperaties:

- Alleen de master node handelt leesoperaties af.
- Elke node kan leesoperaties afhandelen.

Als het systeem alle leesoperaties doorverwijst naar de master node, zal er geen verlies zijn van consistentie, aangezien alle updates eerst naar de master node worden gestuurd en deze dus ook consistente data bevat. Met de vertraging daarentegen kunnen er wel problemen optreden:

- Het eerste probleem in verband met vertraging is het feit dat de leesoperatie eerst naar de master node moet worden gestuurd, zelfs als er in de buurt een node aanwezig is met consistente data. Het is duidelijk dat het sturen naar een master node heel wat tijd in beslag kan nemen, aangezien deze node ook effectief op een lange afstand kan liggen van bijvoorbeeld de gebruiker die de leesoperatie heeft uitgevoerd.
- Het andere probleem heeft te maken met de werklast die de master node te verduren krijgt. Als er verschillende leesoperaties worden verstuurd naar de master node, zullen er een heel aantal leesoperaties in de wachtrij komen te staan. Deze operaties moeten wachten tot de master node de andere leesoperaties verwerkt heeft. Naast een te grote werklast, kan er nog een groter probleem ontstaan, namelijk dat de master node niet meer werkt. Op dat ogenblik moeten alle leesoperaties wachten tot de master node op een of andere manier terug hersteld wordt. Een mogelijke manier om een master node te herstellen, is door een nieuwe master node te verkiezen. We bespreken dit in Paragraaf 7.3.2.

Een andere mogelijkheid om leesoperaties uit te voeren, is door alle nodes de leesoperaties te laten afhandelen. Het is duidelijk dat we de overhead van het doorsturen van de leesoperaties naar de master node niet meer moeten uitvoeren, met als gevolg dat de vertraging sterk afneemt. Het probleem bij deze benadering is het feit dat we met verschillende versies van het systeem te maken kunnen hebben. Zoals eerder vermeld zal de master node de update als voltooid beschouwen als één node de juiste waarde geschreven heeft, maar ondertussen kan het voorkomen dat andere nodes de update niet hebben binnengekregen of nog niet hebben uitgevoerd. We hebben met andere woorden te maken met inconsistente data aangezien alle nodes de leesoperaties kunnen afhandelen [2].

Combinatie van Synchroon en Asynchroon Kopiëren

De laatste mogelijkheid die betrekking heeft tot de master node is een combinatie van synchroon en asynchroon kopiëren. Het hoofdidee is hier dat we een subset van het systeem synchroon gaan behandelen en de rest asynchroon. Ook hier hangt de trade-off af van hoe de leesoperaties worden afgehandeld door het systeem:

- Als leesoperaties langs minstens één synchrone node worden geleid.
- Als leesoperaties langs asynchrone nodes worden geleid.

De eerste mogelijkheid is dat de leesoperaties langs minstens één synchrone node worden geleid. Met andere woorden is dit een node die deel uitmaakt van de subset van het systeem dat synchroon wordt behandeld. Door gebruik te maken van deze methode zijn er geen problemen met consistentie. We hebben daarentegen wel te maken met vertragingsproblemen die we reeds in deze Paragraaf besproken hebben (synchroon kopiëren en asynchroon kopiëren waar de leesoperaties allemaal via de master node gebeuren).

In het andere geval, waar leesoperaties enkel langs asynchrone nodes worden geleid, kunnen er inconsistente leesoperaties gebeuren zoals we ook reeds in deze Paragraaf besproken hebben (asynchroon kopiëren waar leesoperaties door elke node kunnen worden uitgevoerd) [2].

2.4.5 Updates Eerst naar een Willekeurige Node

De laatste mogelijkheid om updates uit te voeren, is door deze updates eerst naar een willekeurige node te sturen. Nadat deze node de update heeft ontvangen, zal hij deze verder zenden naar de rest van de nodes. Op het eerste zicht is er geen verschil tussen deze manier en de manier met een master node zoals besproken in Paragraaf 2.4.4, maar er is zeker een verschil. Het verschil is dat de node waar de update eerst naar verstuurd wordt, kan verschillen. Zo kan een update voor een data element A verstuurd worden naar node X en een update voor data element B naar node Y. Deze updates kunnen ook tegelijk worden uitgevoerd. De trade-off tussen vertraging en consistentie zal bij deze manier afhangen van twee mogelijkheden:

- Het kopiëren gebeurt synchroon.
- Het kopiëren gebeurt asynchroon.

Als het kopiëren synchroon gebeurt, hebben we te maken met dezelfde vertragingsproblemen zoals eerder vermeld in Paragraaf 2.4.4 (synchroon kopiëren). Merk op dat we hier extra vertraging kunnen creëren als updates voor hetzelfde data element worden verstuurd naar een verschillende start node. Deze extra vertraging wordt gecreëerd omdat beide master nodes nog eens opnieuw een synchronisatie onderling moeten uitvoeren.

Aan de andere kant kunnen we ook asynchroon kopiëren en hier hebben we dan dezelfde problemen als in Paragraaf 2.4.3 en in Paragraaf 2.4.4 (asynchroon kopieren) [2].

In de volgende Paragraaf bespreken we de verschillende mogelijkheden die er zijn als we te maken krijgen met partities.

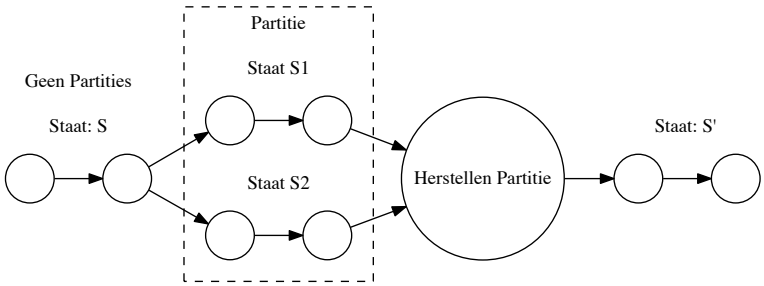
2.5 Partities

We hebben het reeds gehad over de verschillende trade-offs die moeten gemaakt worden bij het ontstaan van een partitie. De taak van elke designer is het effect op consistentie en beschikbaarheid tijdens een partitie zo klein mogelijk te houden. Dit wordt vaak gedaan door een soort van middenweg te kiezen tussen

consistentie en beschikbaarheid. We moeten echter wel voorzichtig omspringen met partities. We moeten enerzijds proberen om een partitie te herkennen en anderzijds moeten we het systeem kunnen herstellen nadat de partitie ongedaan is gemaakt. Een bijzonder geval is hier dat we te maken kunnen hebben met *invarianten*. Invarianten zijn condities die altijd waar moeten zijn in een systeem [12]. Bijvoorbeeld de conditie dat het loon van een werknemer nooit kleiner mag zijn dan nul of dat de examencijfers van een student aan de UHasselt nooit hoger mogen zijn dan twintig.

In deze Paragraaf bespreken we:

- Het zoeken naar een partitie.
- Het activeren van een bepaalde partitie mode.
- Het starten van het herstellingsproces nadat de partitie is hersteld.



Figuur 2.14: Schematische voorstelling van een systeem waar een partitie ontstaat en die vervolgens hersteld wordt.

We leggen deze stappen eerst schematisch uit aan de hand van Figuur 2.14 en gaan vervolgens dieper in op elke stap in de volgende Paragrafen. In Figuur 2.14 zien we een schematische voorstelling van een systeem doorheen de tijd dat eerst niet gepartitioneerd is, vervolgens gepartitioneerd wordt en deze partitie herstelt. We merken op dat we in Figuur 2.14 een andere terminologie gebruiken dan in de reeds geziene Figuren. In deze Figuur is er horizontaal tijdsverloop. De cirkels stellen een bepaalde staat van het systeem voor. We merken op dat de cirkel met "Herstellen Partitie" groter is dan de andere cirkels. De reden hiervoor is om aan te duiden dat alle staten in deze cirkel worden samengevoegd naar een consistente staat. Ten slotte betekenen de gerichte bogen tussen de staten de overgang van de ene staat naar de andere. Deze bogen hebben dus niets te maken met een communicatielijn.

In Figuur 2.14 begint het systeem in de staat S waarin het systeem onder normale omstandigheden werkt. Hiermee bedoelen we dat er geen partitie aanwezig is en dat het systeem zich in een consistente staat bevindt. We kunnen elke node zien als een staat van het systeem. Als er een operatie wordt uitgevoerd op een bepaalde node, springen we naar een nieuwe staat en dus ook naar

een nieuwe node. Aangezien we werken met atomaire operaties, zal een partitie altijd optreden tussen twee operaties.

Als het systeem een time-out ontvangt, is het mogelijk dat er een partitie aanwezig is. Het deel van het systeem dat de partitie heeft ontdekt, gaat dan in *partitie mode*. In Figuur 2.14 zien we dat er twee onafhankelijke paden worden gecreëerd, namelijk $S1$ en $S2$. We merken op dat dit een andere voorstelling is dan we tot nu toe gebruikt hebben. De essentie hier is dat de partitie twee delen creëert die elk een "eigen leven" gaan leiden. Ze kunnen dus niet meer met elkaar communiceren en dit kan leiden tot een inconsistent systeem.

Na bepaalde tijd wordt de partitie opgelost en gaan we door middel van een herstellingsproces terug naar een consistente staat S' [12].

In volgende Paragrafen bespreken we elke stap in detail, hierbij houden we Figuur 2.14 best in gedachte.

2.5.1 Beslissingen tijdens een Partitie

Wat we tijdens een partitie doen, hangt grotendeels af van de bestaande invarianten. We kunnen kiezen tussen:

- Het respecteren van de invarianten gedurende de partitie mode.
- Het schenden van de invarianten met het oog deze te herstellen gedurende de herstelling van de partitie.

We geven een voorbeeld: een systeem heeft als invariant dat alle keys uniek moeten zijn. We kunnen er in dit geval voor kiezen om deze invariant te schenden tijdens de partitie mode. Dit doen we omdat het herkennen van dubbele keys relatief gemakkelijk is. Tijdens de herstelling kunnen we vervolgens eenvoudig de keys samenvoegen [12].

In sommige gevallen is het wel nodig dat de invarianten niet worden geschonden tijdens een partitie. In dergelijke gevallen moeten we ervoor zorgen dat we operaties die deze invariant zouden kunnen schenden, verbieden of wijzigen. In de praktijk weten we echter nooit of de operatie de invariant echt zal schenden aangezien we de staat van het systeem "aan de andere kant" niet kennen.

Naast praktische zaken in verband met de consistentie en verwerking van gegevens is er ook een probleem met de interactie tussen het systeem en de gebruikers. Het systeem moet immers op een duidelijke manier communiceren met de gebruikers over het feit dat hun taak is opgenomen, maar nog niet voltooid is. De gebruiker weet met andere woorden dat er iets mis is, maar dat dit later opgelost zal worden.

Een reden waarom we gebruik maken van *atomaire operaties* is omdat we op deze manier relatief gemakkelijk de invloed op de invarianten kunnen ontdekken. Met atomaire operaties bedoelen we operaties die volledig worden uitgevoerd alvorens een andere operatie kan uitgevoerd worden. Om te weten te komen welke impact de operaties hebben op de invarianten, moeten we simpelweg het kruisproduct maken tussen de operaties en de invarianten. We hebben bijvoorbeeld de volgende operaties:

$$Operaties = \{X + 5,\ Y \times 5,\ Z - 100\}$$

Met de volgende invarianten:

$$Invarianten = \{X < 100,\ Y \geq 0,\ 50 \leq Z \leq 1000\}$$

We kunnen dan Tabel 2.1 berekenen en vervolgens besluiten welke operaties veilig zijn voor bepaalde invarianten en welke operaties we moeten uitstellen, wijzigen of verbieden.

Operatie	Invariant	Invloed?
$X + 5$	$X < 100$	Ja
$X + 5$	$Y \geq 0$	Nee
$X + 5$	$50 \leq Z \leq 1000$	Nee
$Y \times 5$	$X < 100$	Nee
$Y \times 5$	$Y \geq 0$	Nee
$Y \times 5$	$50 \leq Z \leq 1000$	Nee
$Z - 100$	$X < 100$	Nee
$Z - 100$	$Y \geq 0$	Nee
$Z - 100$	$50 \leq Z \leq 1000$	Ja

Tabel 2.1: Voorbeeld van het kruisproduct tussen de operaties en de invarianten.

Merk op dat bij een vermenigvuldiging met 5 een positief getal positief blijft en bijgevolg dus geen invloed zal hebben op de invariant $Y \geq 0$.

Naast de discussie over het wel of niet schenden van invarianten, is het ook belangrijk dat we op een of andere manier de geschiedenis van het systeem (ook van beide delen van de partitie) kunnen bijhouden. Dit maakt het later gemakkelijker om terug te keren naar een perfecte consistente staat. Een goede manier hiervoor is het gebruik maken van *vector klokken*. Deze vector klokken maken het mogelijk om *causale afhankelijkheden* van operaties weer te geven. Hiermee bedoelen we dat er een bepaalde volgorde van operaties nodig is. Ze moeten met andere woorden in die volgorde uitgevoerd worden. Dit is in tegenstelling tot *concurrente* operaties die tegelijk worden uitgevoerd. We geven nog een korte beschrijving van de vector klokken. Meer informatie over causale consistentie is terug te vinden in Paragraaf 3.4.

Elk element in de vector klok is een koppel (*node*, *tijd*). Hierbij geeft *node* elke node weer die een bepaalde update heeft uitgevoerd. Het element *tijd* geeft de tijd weer van de laatste update. Aan de hand van deze klokken kunnen we vervolgens kijken welke staat van een object het nieuwste is en kunnen we zo dus bepalen welke informatie we moeten houden. In het andere geval, als er geen vergelijking mogelijk is tussen verschillende klokken, dan weten we dat de operaties concurrent zijn uitgevoerd en dus mogelijk tot inconsistente situaties kunnen leiden [12].

2.5.2 Herstellen van een Partitie

Nu we weten wat er gebeurt tijdens partities, kunnen we kijken hoe het systeem hiervan herstelt. Er komt een moment in de tijd waarop de partitie ongedaan wordt gemaakt, dit kan bijvoorbeeld door het herstellen van een communicatielijn. We weten dat de verschillende delen afzonderlijk beschikbaar waren tijdens de partitie, maar over de consistentie zijn er twijfels. Tijdens de partitie hebben we ervoor gekozen om enerzijds invarianten te schenden of anderzijds ze te

respecteren. Door het zorgvuldig bijhouden van de vector klokken tijdens de partitie heeft het systeem een duidelijk beeld over de geschiedenis en de staat van de delen van de partitie. Deze geschiedenis gebruiken we om af te leiden welke invarianten er geschonden zijn gedurende de partitie. Verder moeten we nu twee belangrijke problemen oplossen: enerzijds moeten we de staat van beide delen consistent maken en anderzijds moeten we eventuele fouten rechtzetten [12].

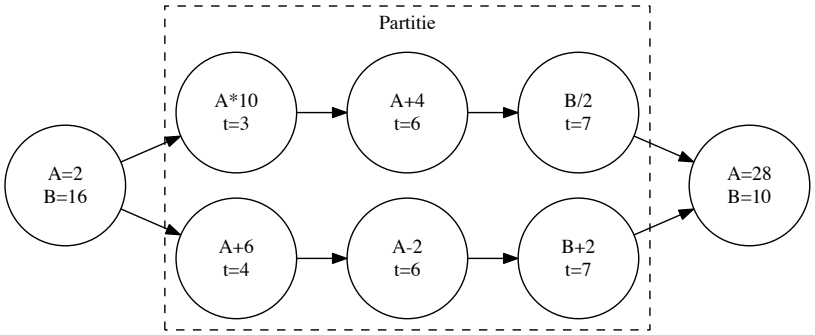
Een relatief gemakkelijke manier om naar een consistente staat terug te keren is door naar de plaats te gaan waar de partitie begon. Vanaf dit moment kunnen we de operaties van beide delen uitvoeren en op die manier zorgen dat we een consistente staat behouden.

Om dit te verduidelijken geven we een voorbeeld. We beschikken over een systeem (zonder invarianten) dat twee waardes bijhoudt, namelijk A en B. Op deze waardes kunnen we eender welke rekenkundige bewerking uitvoeren. In Figuur 2.15 zien we een situatie waarin we ons eerst in een consistente staat bevinden waar A de waarde 2 heeft en B de waarde 16 heeft. Verder zien we in Figuur 2.15 dat er een partitie ontstaat. We zien ook dat elke node bijhoudt welke operaties worden uitgevoerd en welke vector klok waardes ze bezitten. Uiteindelijk wordt de partitie hersteld en voegen we beide partities samen. We bekijken nu een manier hoe dit zou kunnen gebeuren. We keren terug naar het begin van de partitie en de eerste bewerking die we tegenkomen is $A \times 10$ op tijdstip 3. Aangezien dit de enige operatie is dit op tijdstip 3 is uitgevoerd, kunnen we deze zonder problemen uitvoeren. De waarde van A wordt bijgevolg 20. De volgende bewerking $A + 6$ gebeurt op tijdstip 4. Dit is ook de enige bewerking die gebeurt op tijdstip 4, dus is er geen probleem. We passen de waarde van A daarom aan naar 26. Op het volgende tijdstip 6 worden er twee bewerkingen uitgevoerd, namelijk $A + 4$ en $A - 2$. We weten dat optellen en aftrekken commutatieve bewerkingen zijn dus maakt het niet uit in welke volgorde we deze bewerkingen uitvoeren. A zal bijgevolg worden aangepast naar 28. Het laatste tijdstip 7 bevat ook twee operaties, namelijk $\frac{B}{2}$ en $B+2$. Hier zitten we met een probleem aangezien we hier niet te maken hebben met commuterende operaties. Welke operatie nu eerst zal worden uitgevoerd, zal afhangen van het systeem waarmee we werken. In ons voorbeeld hebben we besloten om de algemeen bekende "volgorde van bewerkingen" aan te houden, B zal met andere woorden worden aangepast naar 10 omdat een deling vóór een aftrekking komt in de volgorde van bewerkingen.

Een andere manier is het gebruik van commutative, replicated data types (CRDTs), deze bespreken we niet in deze Paragraaf, we verwijzen hiervoor naar Paragraaf 3.3.5.

2.5.3 Compenseren van Fouten

Zoals we al enkele keren hebben opgemerkt, kunnen er tijdens de partitie invarianten worden geschonden. Deze moeten bijgevolg gecompenseerd worden na de partitie, we moeten er met andere woorden voor zorgen dat de invariant na de herstelling van de partitie niet meer geschonden is. Het effectief vinden van operaties die de invariant hebben geschonden is op zich geen groot probleem aangezien we een overzicht hebben van wat er in de partitie mode gebeurd is. Meestal worden deze fouten opgemerkt tijdens de partitie herstelling en op dat moment zal er ook een gepaste maatregel moeten worden getroffen.



Figuur 2.15: Schematische voorstelling die weergeeft hoe we van een gepartitioneerde staat naar een niet-gepartitioneerde, consistente staat gaan.

Dergelijke invarianten kunnen leiden tot een *geëxternaliseerde fout*. We geven eerst een voorbeeld waar de fout niet geëxternaliseerd zal worden. Wanneer we bijvoorbeeld gebruik maken van Dropbox of iCloud worden er tijdens een partitie verschillende versies van een bepaald bestand geüpload. Aan het einde van de rit worden alle oude versies genegeerd en enkel de nieuwste versie wordt behouden. De gebruikers zullen hier geen last van ondervinden.

In het vorige voorbeeld wordt een eenvoudige techniek gebruikt om de invarianten te herstellen, namelijk "last-writer-wins". Deze methode negeert alle vorige operaties en laat enkel de laatste operatie doorgaan. Er zijn ook slimmere methodes die verschillende operaties samenvoegen.

Een ander voorbeeld waar een fout wel geëxternaliseerd wordt, is bijvoorbeeld bij online boekingen. Stel dat we eigenaar zijn van hotel en dat er online boekingen kunnen worden gemaakt. Als er tijdens een partitie een overboeking wordt gemaakt en deze pas wordt waargenomen tijdens de herstelling van een partitie, dan zitten we met een groot probleem. In een worst-case scenario kan het hotel bijvoorbeeld dubbel volzet zitten, waardoor het hotel enorm veel mensen zal moeten afbellen.

Om geëxternaliseerde fouten te compenseren, is het noodzakelijk dat we de geschiedenis kennen.

Als we dit doortrekken naar echte systemen, dan kan het gebeuren dat bepaalde operaties twee keer worden uitgevoerd tijdens een partitie. Als het systeem de mogelijkheid heeft om twee doelbewuste bestellingen te onderscheiden van twee duplicate bestellingen, is er geen probleem. Het systeem kan immers één van de duplicaten verwijderen en de gebruiker zal nooit weten dat er iets is misgelopen. Als we dit te laat opmerken en de staat van het systeem wordt geëxternaliseerd, dan zal er een andere manier worden gebruikt om te compenseren. Het systeem kan bijvoorbeeld een e-mail sturen naar de klant om hem te laten weten dat de bestelling dubbel is geregistreerd en dat het bedrijf dit zal oplossen. Voor het ongemak kan de klant bijvoorbeeld korting krijgen op zijn volgende bestelling. Zonder enige vorm van geschiedenis zou het niet mogelijk zijn om dergelijke problemen op te vangen en zou de klant twee net dezelfde

producten opgestuurd krijgen, met de nodige frustratie tot gevolg [12].

2.6 Conclusie

In dit Hoofdstuk hebben we het belangrijke acroniem CAP besproken. De CAP theorie zegt dat het niet mogelijk is om in een gedistribueerd systeem aan de drie eigenschappen van de CAP theorie tegelijk te voldoen. Dit leidt tot heel wat beperkingen voor gedistribueerde systemen. Belangrijke kritiek die op deze stelling gegeven wordt, is dat er geen rekening wordt gehouden met de vertraging die er kan optreden in systemen. Daarom hebben we in dit Hoofdstuk het PACELC acroniem geïntroduceerd. Toch vormt de oorspronkelijke CAP theorie, zoals behandeld in Paragraaf 2.2, een belangrijke basis voor de onderwerpen die we nog zullen bespreken in het vervolg van deze thesis.

Verder hebben we in dit Hoofdstuk besproken hoe er in een gedistribueerd systeem data gekopieerd kan worden naar verschillende nodes. We hebben gezien dat dit kan door naar alle nodes tegelijk te sturen, eerst naar een master node of naar een willekeurige node. De keuzes die we hier maken, heeft gevolgen voor ofwel de consistentie ofwel de vertraging. Er moet bijgevolg goed nagedacht worden over welke manier we gebruiken. Dit hangt echter sterk af van de toepassing die we willen creëren.

Ten slotte hebben we gezien wat er allemaal gebeurt tijdens een partitie en hoe we deze partitie kunnen herstellen.

Hoofdstuk 3

Soorten Consistentie

3.1 Inleiding

In het vorige Hoofdstuk hebben we de CAP theorie uitgebreid besproken. Hier hebben we gezien dat we (tijdens een partitie) maar aan twee van de drie systeemeisen tegelijk kunnen voldoen. In het kader van onze thesis zijn partitie tolerantie en beschikbaarheid de belangrijkste eisen. We hebben het namelijk over NoSQL databases die vaak gebruikt worden in systemen waar we in alle situaties snel een antwoord willen krijgen. Het is minder belangrijk dat dit antwoord consistent is. Voorbeelden van dergelijke systemen zijn bijvoorbeeld Twitter of Youtube. Wil dit zeggen dat consistentie totaal niet belangrijk is? Het antwoord hierop is uiteraard nee.

In dit Hoofdstuk bekijken we manieren om toch een bepaalde graad van consistentie te bereiken door het gebruik van verschillende soorten consistenties en andere technieken. Een belangrijke en veel gebruikte soort consistentie is uiteindelijke consistentie of in het Engels eventual consistency.

Daarnaast bespreken we ook de causale consistentie of in het Engels causal consistency.

We beginnen met een overzicht te geven van de twee grote groepen consistenties.

3.2 Zwakke en Sterke Consistentie

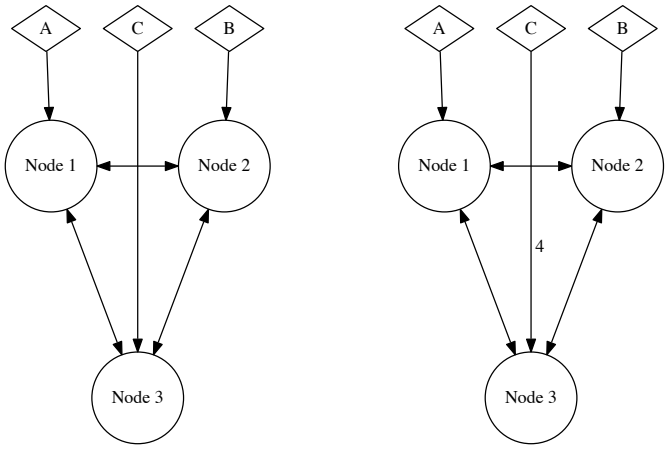
Er zijn twee grote groepen van consistenties: enerzijds hebben we *zwakke consistentie* of in het Engels *weak consistency* en anderzijds hebben we *sterke consistentie* of in het Engels *strong consistency*. Sterke consistentie is in feite de meest strikte vorm van consistentie. Deze consistentie zegt dat alle nodes in het systeem op eender welk ogenblik dezelfde data zien. Als we dit bekijken in de context van de CAP theorie uit Hoofdstuk 2, zien we dat we dit soort consistentie kunnen bereiken als we ofwel beschikbaarheid opofferen (CP) ofwel partitie tolerantie opofferen (CA). Er is namelijk een bepaalde kost (herinner Paragraaf 2.4.3) om ervoor te zorgen dat een systeem voldoet aan sterke consistentie.

Zwakke consistentie wil zeggen dat nodes in een systeem op eender welk ogenblik verschillende data kunnen zien, maar dat er toch nog bepaalde consis-

tentie gerelateerde eigenschappen gelden. Het ene ogenblik zal een node oude data zien, maar het andere ogenblik kan deze node de nieuwste data zien. Dit soort consistentie wordt ingevuld in de CAP stelling als we met een AP (beschikbaar en partitie tolerant) systeem te maken hebben. We weten dat er een mogelijkheid is dat er inconsistente data in bepaalde nodes zit, maar we zijn zeker dat deze data ooit naar een consistente staat zal evolueren [52].

We verduidelijken beide groepen aan de hand van een voorbeeld. Stel dat we een beginsituatie hebben zoals voorgesteld in Figuur 3.1a. We zien drie nodes die onderling met elkaar verbonden zijn en op deze manier informatie over updates en dergelijke kunnen uitwisselen. We stellen dat elke node één waarde bezit en dat deze initieel op "0" staat.

We gaan verder met ons voorbeeld en proces C schrijft de waarde 4 naar node 3. Dit zien we gebeuren in Figuur 3.1b.



(a) De beginsituatie met drie nodes die een waarde "0" bevatten.

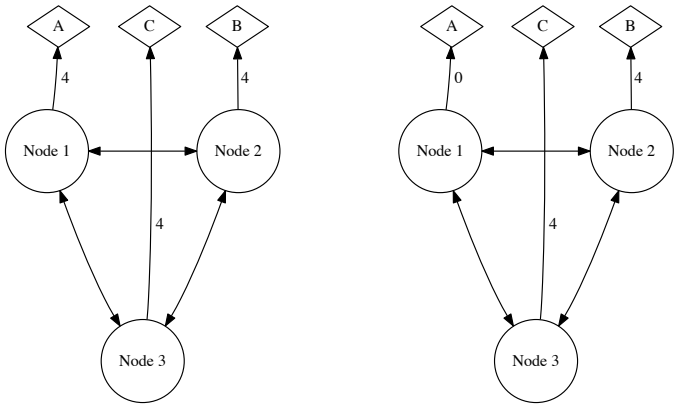
(b) Proces C schrijft de waarde 4 naar node 3.

Figuur 3.1: Schematische voorstelling om de twee groepen consistenties uit te leggen.

Als er na de schrijfoperatie een leesoperatie gebeurt door proces A, proces B en proces C, dan zou bij sterke consistentie deze leesoperatie overal dezelfde waarde 4 moeten teruggeven. Dit zien we in Figuur 3.2a. We herinneren dat sterke consistentie wil zeggen dat we altijd de meest recente en dus de meest correcte data willen terugkrijgen bij een leesoperatie [52].

We gebruiken dezelfde beginsituatie samen met dezelfde schrijfoperatie zoals in Figuur 3.1a en Figuur 3.1b om het zwakke consistentiemodel uit te leggen. Als er in dit model een leesoperatie wordt uitgevoerd na een schrijfoperatie, dan kan het systeem niet garanderen dat elke node de laatste en dus juiste waarde zal teruggeven. Dit zien we in Figuur 3.2b. We merken op dat proces B en proces C de juiste waarde teruggekregen hebben en dat proces A een oudere versie van de data heeft teruggekregen. In dergelijke systemen is er sprake van een *inconsistentievenster* of in het Engels een *inconsistency window*. Dit is de

periode vanaf de update (schrijfoperatie) tot het systeem kan garanderen dat alle processen de nieuwste data terugkrijgen bij een leesoperatie. In ons voorbeeld krijgen, na het verstrijken van het inconsistentievenster, alle nodes de waarde 4 terug, op voorwaarde dat er hiertussen geen nieuwe schrijfoperaties gebeurd zijn [52].



(a) Een leesoperatie van de drie processen na de schrijfoperatie (sterke consistentie).

(b) Een leesoperatie van de drie processen na de schrijfoperatie (zwakke consistentie).

Figuur 3.2: Onderscheid tussen sterke en zwakke consistentie.

We bestuderen hierna twee concrete vormen van zwakke consistentie, namelijk, uiteindelijke consistentie en causale consistentie.

3.3 Uiteindelijke Consistentie

Uiteindelijke consistentie of in het Engels *eventual consistency* is een specifieker model dat onder de zwakke consistentie valt. Dit model garandeert, als er geen nieuwe updates (schrijfoperaties) gebeuren dat het systeem dan uiteindelijk in alle nodes de nieuwste data ter beschikking heeft. Alle leesoperaties geven dus uiteindelijk dezelfde data terug. Het inconsistentievenster kan benaderend berekend worden door rekening te houden met bijvoorbeeld eventuele vertragingen en werklast. Hier moeten we wel rekening houden met het feit dat deze berekeningen enkel gelden als er geen partities of andere fouten gebeuren in het systeem.

In feite is uiteindelijke consistentie een zwak model. De gebruiker weet in principe niet of hij met consistente data werkt, aangezien het systeem eender welke data kan teruggeven en toch uiteindelijk consistent kan zijn. Het positieve punt is dat we er zeker van zijn dat ergens in de toekomst de data consistent zal zijn. Willen applicaties dit soort consistentie hebben? Het lijkt een zwak model, maar toch worden er veel applicaties gebouwd die steunen op het uiteindelijk consistentiemodel [9].

Om het nut van uiteindelijke consistentie te benadrukken geven we een voorbeeld. We merken op dat dit voorbeeld zich opnieuw in het kader van de CAP stelling bevindt. Stel dat iemand een nieuwe tweet plaatst op Twitter. We veronderstellen dat er zeer veel servers zijn die deze tweet doorgeven om de andere mensen die op Twitter actief zijn, te kunnen bedienen. Stel dat er enkele communicatielijnen tussen de servers uitvallen, met als gevolg dat de nieuwe tweet niet kan worden doorgestuurd naar de andere servers. De mensen die aan de "goede kant" van de servers Twitter gebruiken, kunnen zonder problemen de nieuwe tweet lezen, maar wat zeggen we tegen de mensen die zich aan de "slechte kant" van de servers bevinden? We bedoelen hiermee dat deze servers de update van de tweet momenteel niet kunnen ontvangen door het falen van de communicatielijnen. Wat kunnen we nu doen? Enerzijds kunnen we gaan voor consistentie: Wacht tot het probleem is opgelost en geef ondertussen bijvoorbeeld een foutmelding. Dit gaat natuurlijk ten koste van de gebruiksvriendelijkheid (beschikbaarheid). Anderzijds kunnen we ook doen alsof er niets aan de hand is en de beschikbare tweets laten zien. De gebruiker zal niet de nieuwste tweets zien, maar hij kan toch gebruik blijven maken van het systeem. Door onze uiteindelijke consistentie zijn we er zeker van dat er ergens in de toekomst toch de juiste tweet zal worden weergegeven. Als deze gebruiker bijvoorbeeld binnen een minuut terugkijkt naar de pagina, zal hij misschien wel de nieuwste tweet zien. Deze gebruiker zal waarschijnlijk nooit weten dat hij deze tweet "te laat" heeft gezien.

3.3.1 Liveness en Safety

Uiteindelijke consistentie garandeert dat we uiteindelijk, consistente data lezen uit alle nodes. Deze eigenschap van uiteindelijke consistentie noemen we de *liveness* eigenschap. Met andere woorden wil deze eigenschap zeggen dat er uiteindelijk *iets goed* zal gebeuren.

Op zich is dit een vrij sterke eigenschap, die ons een bepaalde garantie van zekerheid geeft. Er is echter wel een probleem dat kan opduiken. Er is namelijk een tweede belangrijke eigenschap, namelijk *safety* die standaard niet wordt voorzien door uiteindelijke consistentie. Deze safety eigenschap garandeert dat er niets fout loopt. Dit wil zeggen dat we enkel waardes uit de database kunnen lezen die voordien waren geschreven. Merk op dat een systeem dus uiteindelijk consistent kan zijn, zelfs als er waardes worden teruggegeven die nooit naar de database zijn geschreven. Dit moeten we natuurlijk vermijden aangezien we op deze manier een totaal fout beeld van de data kunnen krijgen. Een voorbeeld kan zijn dat we bij het opvragen van de examencijfers van een student "-1324" terugkrijgen.

Als we dit samenvatten, is het probleem dat uiteindelijke consistentie rekening houdt met de eerste eigenschap van liveness, maar niet met de tweede eigenschap van safety. Uiteindelijk zal er dus iets goed gebeuren (in elke node wordt bijvoorbeeld dezelfde waarde gelezen), maar in de tussentijd hebben we geen controle over wat er precies gebeurt.

Veel modellen die krachtiger zijn dan uiteindelijke consistentie bieden op één of andere manier een vorm van safety aan. Uiteindelijke consistentie kan dus gezien worden als het absolute minimum wat betreft data consistentie [9].

We hebben uiteindelijke consistentie besproken in theorie, het volgende wat we bekijken, is hoe we de prestaties van uiteindelijke consistentie kunnen meten.

3.3.2 Hoe Goed is Uiteindelijke Consistentie?

Zoals we reeds besproken hebben, biedt uiteindelijke consistentie geen safety eigenschap en toch wordt dit model in veel systemen gebruikt. We kunnen ons de vraag stellen waarom dit zo is. In de praktijk werkt uiteindelijke consistentie goed wegens de hoge beschikbaarheid en lage vertraging die het biedt.

We kunnen de prestaties van uiteindelijke consistentie binnen een specifieke toepassing of systeem meten. We bespreken drie technieken die we kunnen gebruiken om de prestaties van uiteindelijke consistentie te berekenen of te voorspellen:

- maatstaven
- mechanismen
- probabilistically bounded staleness

Maatstaven

We bespreken in deze Paragraaf twee maatstaven, namelijk:

- tijd
- versie

De eerste maatstaf is tijd. Hier berekenen we hoelang het duurt vooraleer de geschreven data beschikbaar wordt voor iedereen. Dit komt overeen met het inconsistentievenster uit Paragraaf 3.3.

De tweede maatstaf is het gebruik van versies [9]. We illustreren dit kort met een voorbeeld. Stel dat ons systeem één getal kan bijhouden en dat de volgende getallen geschreven worden door gebruikers (Tabel 3.1):

Versie 1	Versie 2	Versie 3	Versie 4	Versie 5
1	5	2	14	9

Tabel 3.1: Voorbeeld van het gebruik van versies: schrijven van data.

Stel dat er nu drie personen zijn die de waarde uit het systeem proberen te lezen, dan kunnen we de volgende situatie verkrijgen (Tabel 3.2):

Persoon	Leest waarde:	Hoeveel versies achter?
Rob	9	0
Tom	1	4
Gert	14	1

Tabel 3.2: Voorbeeld van het gebruik van versies: lezen van data.

Aan de hand van dit voorbeeld zien we duidelijk dat we van elke persoon kunnen zien hoeveel versies hij achter staat op de huidige, nieuwste versie. Deze informatie kan ook gebruikt worden om ervoor te zorgen dat gebruikers zeker geen oudere versies lezen: de volgende versie die Gert bijvoorbeeld leest, zal versie 5 zijn of hoger, maar zeker niet lager.

Mechanismen

We bespreken in deze Paragraaf twee mechanismen, namelijk:

- meten
- voorspellen

Voor mechanismen is de eerste methode: *meten*. Dit is een eenvoudig begrip, het beantwoordt de vraag: "Hoe consistent *is* mijn systeem, gegeven een bepaalde werklast?". Deze methode is handig om bijvoorbeeld te berekenen hoeveel procent van de nodes op een bepaald ogenblik consistent zijn.

De tweede methode is: *voorspellen*. Dit begrip beantwoordt net zoals de eerste methode een vraag: "Hoe consistent *zal* mijn systeem zijn, gegeven een configuratie en een werklast?" Deze methode kan handig zijn voor het berekenen van de kans dat het systeem onder een bepaalde werklast blijft werken. We kunnen bijvoorbeeld het aantal nodes specificeren, samen met het aantal operaties die worden gestuurd naar een node. We kunnen dan voorspellen of een node niet overbelast raakt en of deze node nog voldoende zijn updates kan doorsturen naar de rest van het netwerk [9].

Probabilistically Bounded Staleness

Als laatste bespreken we een recentere techniek die gebruikt wordt om voorspellingen te doen, namelijk *Probabilistically Bounded Staleness* (*PBS*). Probabilistically Bounded Staleness zijn modellen die helpen om de consistentie van uiteindelijk consistente databases te voorspellen. Deze modellen beantwoorden vragen als:

- Hoe uiteindelijk is uiteindelijke consistentie?
- Hoe consistent is uiteindelijke consistentie?

Zoals we ondertussen weten, biedt uiteindelijke consistentie geen garantie over de "nieuwheid" van de data. Toch wordt uiteindelijke consistentie veel gebruikt en we kunnen met behulp van Probabilistically Bounded Staleness aantonen waarom. Met dit model kunnen we de consistentie van een uiteindelijk consistent systeem voorspellen. We kunnen ook verklaren waarom data consistent of net inconsistent is. Zoals eerder vermeld, zal in de praktijk blijken dat uiteindelijk consistente systemen in de meeste gevallen consistente data afleveren. Met behulp van PBS kunnen we ook trade-offs tussen consistentie en vertraging beter inschatten [9].

Voor specifieke details van PBS verwijzen we naar [10].

3.3.3 Compensaties

We hebben in vorige Paragrafen voorbeelden gezien over hoe we de consistentie en andere prestaties van een uiteindelijk consistent systeem kunnen voorspellen en berekenen. We moeten er rekening mee houden dat dit onze safety eis niet oplost aangezien deze niet wordt gegarandeerd door uiteindelijke consistentie.

We kunnen deze afwijkingen van de consistentie omzeilen door dit te programmeren. Het probleem is hierbij dat we dikwijls met *speculaties* bezig zijn.

We kunnen in principe niet weten of een bepaalde waarde effectief de laatst geschreven waarde is. We kunnen er echter wel van uit gaan dat we met de laatst geschreven data werken. Als later blijkt dat we gelijk hadden, moet er in principe niets meer gebeuren. Jammer genoeg zal in de meeste gevallen blijken dat het niet de laatst geschreven waarde was en dan moeten we onze fout *compenseren*. Deze compensatie zorgt ervoor dat er alsnog safety kan worden behaald. Het garandeert dat fouten eventueel worden hersteld, maar niet dat er geen fouten worden gemaakt [9].

3.3.4 CALM Stelling

De *CALM stelling* zegt iets over welke programma's al dan niet veilig uitgevoerd kunnen worden onder uiteindelijke consistentie. CALM staat voor *consistentie als logische monotoniciteit* of in het Engels *consistency as logical monotonicity*. Met *logische monotoniciteit* wordt simpelweg bedoeld dat als we programma's meer input geven, de gegenereerde output alleen maar kan toenemen. *Niet-monotone* programma's kunnen we zien als programma's die reeds gegenereerde output kunnen terugnemen [5]. Voorbeelden van monotone operaties zijn de gekende selectie, projectie en join operaties. Typische voorbeelden van niet-monotone operaties zijn de aggregatie en negatie operatoren. We merken op dat bij aggregatie operaties, bijvoorbeeld een *group by*, eerst de volledige output ter beschikking moet zijn alvorens de *group by* operatie zijn werk kan doen [6].

De CALM stelling zegt dat:

- delen van de programmacode die voldoen aan de eigenschap van logische monotoniciteit de uiteindelijke consistentie eigenschap naleven.
- delen van de programmacode die niet voldoen aan de eigenschap van logische monotoniciteit ook kunnen voldoen aan de eigenschap van uiteindelijke consistentie, maar dat we deze code moeten beschermen door middel van een coördinatieprotocol.

We kunnen met andere woorden proberen om zo veel mogelijk monotone code te schrijven omdat deze gegarandeerd uiteindelijk consistent is en als we dan toch niet anders kunnen dan niet-monotone code te schrijven, moeten we dit oplossen met een coördinatieprotocol om toch de uiteindelijke consistentie te behalen.

Onder CALM vallen een aantal *design patronen*, zo hebben we *ACID 2.0* [9]. Dit acroniem staat voor:

1. associativiteit (Engels: associativity)
2. commutativiteit (Engels: commutativity)
3. idempotentie (Engels: idempotence)
4. gedistribueerd (Engels: distributed)

Met de *associatieve eigenschap* wordt bedoeld dat we bij operaties de haakjes van plaats kunnen veranderen, zonder de uitkomst van de operatie te veranderen. We kunnen dit best illustreren met enkele voorbeelden:

$$(a + b) - c = a + (b - c)$$
$$f(a, f(b, c)) = f(f(a, b), c)$$
$$(a + b) \times c \neq a + (b \times c)$$

De *commutatieve eigenschap* zegt dat we argumenten van plaats mogen veranderen, zonder dat de uitkomst verandert.

$$a + b + c = c + a + b$$
$$f(a, b) = f(b, a)$$
$$a - b - c \neq c - b - a$$

De volgende eigenschap die we bespreken is de *idempotentie eigenschap*. Deze eigenschap wil simpelweg zeggen dat we een functie verschillende keren op dezelfde input kunnen toepassen, zonder dat de output hierdoor verandert.

$$f(f(f(f(x)))) = f(x)$$
$$MIN(MIN(MIN(2, 5))) = 5$$

De laatste eigenschap *gedistribueerd* staat enkel voor het feit dat ACID 2.0 wordt gebruikt voor gedistribueerde systemen.

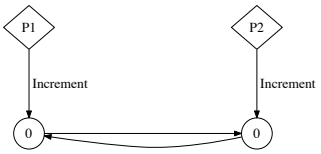
Als we deze design patronen volgen, kunnen we monotoniciteit bereiken en zo ook uiteindelijke consistentie [9].

3.3.5 Commutative, Replicated Data Types

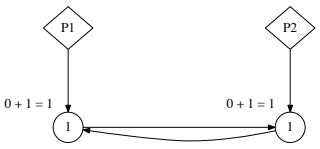
Een andere manier om uiteindelijke consistentie te bereiken, is door gebruik te maken van de *commutative, replicated data types* (*CRDTs*). Hiermee bedoelen we data types die ervoor zorgen dat gelijktijdige operaties toch mooi worden afgehandeld, zelfs als updates in een verkeerde volgorde worden doorgestuurd. Door het gebruik van deze methode is het gebruik van ingewikkelde *concurrency control* ook overbodig. Concurrency control is een mechanisme dat ervoor zorgt dat gelijktijdige operaties correct worden uitgevoerd [46]. Ook CRDTs nemen de principes van CALM en ACID 2.0 mee in hun design voor data types, zoals bijvoorbeeld sets en grafen. Programma's die met andere woorden gebruik maken van CRDTs moeten zich geen zorgen maken over safety en krijgen zonder extra moeite uiteindelijke consistentie.

We geven een voorbeeld waar het duidelijk is dat er problemen ontstaan als we geen gebruik maken van CRDTs: stel dat we een systeem ter beschikking hebben met twee nodes die een initiële waarde hebben van 0. Het systeem kan alleen maar incrementeren (dus enkel +1). Als een gebruiker of een proces zo een increment uitvoert, zal de huidige waarde eerst worden uitgelezen. Deze waarde wordt verhoogd met één en wordt vervolgens terug weggeschreven naar de huidige node om vervolgens ook een kopie van de nieuwe waarde te zenden naar de rest van het systeem. Stel dat in dit systeem twee gebruikers op hetzelfde moment een increment uitvoeren op twee verschillende nodes (zie Figuur 3.3a). Wat zal er dan gebeuren? De eerste uitvoering zal de huidige waarde in de eerste node lezen en zien dat deze een waarde van 0 heeft. Bij deze waarde wordt 1 bij opgeteld en wordt vervolgens terug opgeslagen en doorgestuurd naar de

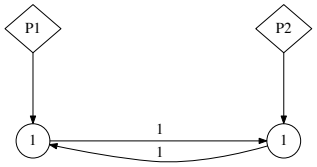
andere node. Ondertussen wordt de tweede increment echter ook uitgevoerd en hier zal net hetzelfde plaatsvinden. Het gevolg hiervan is dat het systeem zal eindigen met de waarde 1 en dus niet de waarde 2. Deze situatie is voorgesteld in Figuur 3.3b en Figuur 3.3c. Het systeem zal deze waarde blijven behouden tot er opnieuw een increment zal plaatsvinden [9].



(a) Beide processen voeren gelijktijdig een increment operatie uit.



(b) In beide nodes wordt gelijktijdig de waarde 0 gelezen en hierbij wordt 1 opgeteld.



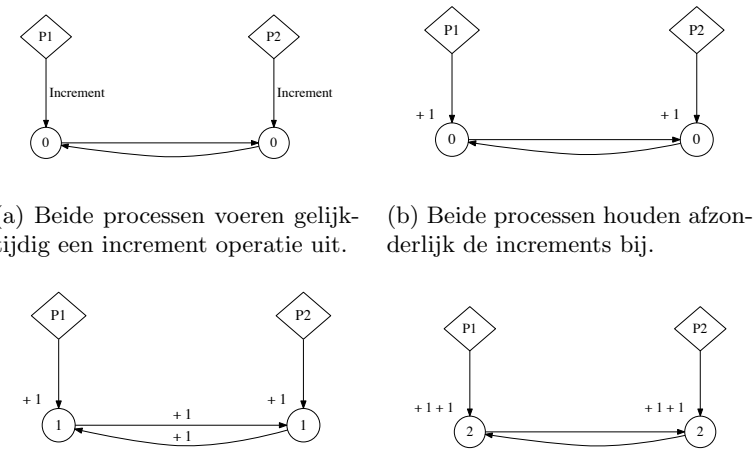
(c) Elke node geeft zijn nieuwe waarde door aan de andere node.

Figuur 3.3: Schematische voorstelling van een increment operatie.

We merken op dat het ook mogelijk is dat beide nodes naar de (correcte) waarde van 2 kunnen evolueren. Dit kan als er voldoende tijd zit tussen de increment van persoon 1 en persoon 2. Het systeem moet met andere woorden tijd hebben om de juiste waarde eerst te kopiëren naar de andere node. Vervolgens kan de andere node incrementeren op de nieuwe waarde en zal er dus geen probleem zijn.

Een betere manier om dit soort systeem te implementeren, is door gebruik te maken van een CRDT, namelijk een *G-counter*. Zoals we in Paragraaf 3.3.4 gezien hebben, zijn er heel wat eigenschappen die uitgebuit kunnen worden. In dit systeem is dat de eigenschap van commutativiteit, het maakt immers geen verschil in welke volgorde de incrementen gebeuren. De enige vereiste is dat ze allemaal moeten gebeuren. Met deze techniek lezen we niet eerst de waarde van de node, maar we zeggen enkel dat er een increment moet gebeuren. De nodes houden met andere woorden bij hoeveel incrementen er moeten gebeuren. Het juiste resultaat wordt vervolgens bekomen door ze uit te voeren. Er kan nu ook nog een foute waarde worden gelezen, maar met dit systeem garanderen we uiteindelijke consistentie. Alle incrementen worden uiteindelijke uitgevoerd zodat er uiteindelijk een juist resultaat bekomen wordt. We passen dit opnieuw toe op een voorbeeld. We beginnen met de beginopstelling zoals weergegeven in Figuur 3.4a. Hier zien we dat beide processen een increment operatie uitvoeren. De volgende stap die zal gebeuren, is dat de nodes bijhouden hoeveel incrementen er moeten gebeuren. Ze kijken dus niet meer wat de huidige waarde is van de node. Dit zien we gebeuren in Figuur 3.4b. Het volgende wat de nodes doen,

is het doorsturen van deze increment operaties zodat de andere node ook weet wat hij moet doen. Dit is geïllustreerd in Figuur 3.4c. Als laatste moeten de nodes enkel de increments uitvoeren om het juiste resultaat te bekomen. Een voorstelling van deze laatste stap is te vinden in Figuur 3.4d. [9].



(a) Beide processen voeren gelijktijdig een increment operatie uit.

(b) Beide processen houden afzonderlijk de increments bij.

(c) De increments worden doorgestuurd naar de nodes.

(d) De increments worden uitgevoerd door de nodes.

Figuur 3.4: Schematische voorstelling van een G-counter.

3.3.6 Separation of Concerns

Een goede techniek om systemen te ontwikkelen, is het toepassen van *separation of concerns*. Separation of concerns wil zeggen dat elk element in een systeem een bepaalde taak krijgt. Als we denken aan systemen die gebruik maken van data, dan hebben we typisch te maken met een database en één of meerdere applicaties. De database krijgt de verantwoordelijkheid over het beheren van de data en de applicaties kunnen interageren met de database. Door gebruik te maken van deze techniek is het relatief eenvoudig om bijvoorbeeld andere applicaties toe te voegen.

We kunnen deze techniek ook doortrekken naar technieken zoals CALM, ACID 2.0 en CRDTs. We onderscheiden twee taken: enerzijds de consistentie in de database, en anderzijds de consistentie van de applicatie. Zoals we weten, is het mogelijk dat de database op sommige momenten inconsistent is. CALM, ACID 2.0 en CRDTs zorgen ervoor dat we consistentie bereiken op een "hoger level", namelijk op het applicatie niveau. De applicatie moet er immers enkel voor zorgen dat de semantiek van de operaties wordt nageleefd, in het voorbeeld in Paragraaf 3.3.5 moet de applicatie ervoor zorgen dat er enkel een increment van één kan gebeuren. Aangezien we weten dat de incrementen commutatief zijn, weten we dat de volgorde waarin deze operaties worden uitgevoerd geen belang heeft [9].

3.3.7 Bloom

Aan de Universiteit van Californië - Berkeley is men bezig met het ontwikkelen van een programmeertaal *Bloom* [4], die steunt op de principes van CALM. We sommen de eigenschappen van deze programmeertaal op:

- *disorderly programming*
- *a collected approach*
- *CALM consistency*
- *concise, familiar code*

We geven dit mee om aan te tonen dat een techniek zoals CALM effectief wordt gebruikt in de praktijk. Voor meer informatie verwijzen we naar [4].

3.3.8 BASE

Zoals we in Paragraaf 1.4 reeds gezien hebben, worden relationele databases gekenmerkt door het ACID principe. NoSQL databases worden niet door het ACID principe gekenmerkt, maar door het "minder strenge" principe van *BASE* [44]. BASE staat voor:

1. eenvoudige beschikbaarheid (Engels: basic availability)
2. zachte toestand (Engels: soft-state)
3. uiteindelijke consistentie (Engels: eventual consistency)

Eenvoudige beschikbaarheid wil zeggen dat het systeem onder elke omstandigheid beschikbaar blijft. Het systeem blijft dus ook beschikbaar als er verschillende nodes falen. NoSQL databases slagen hierin door hun data te kopiëren en deze te verdelen over verschillende nodes of door hun data te partitioneren over verschillende nodes. Dit is een groot verschil ten opzicht van traditionele modellen die één grote server foutvrij en beschikbaar proberen te houden. Technieken om data te verdelen over verschillende nodes komen terug bij Voldemort in Hoofdstuk 6 en bij MongoDB in Hoofdstuk 7.

Zachte toestand wil zeggen dat de data die nodes hebben, mogelijk verouderd is. Met andere woorden de data "vervalt" na een bepaalde tijd. Een goed voorbeeld hiervan is het gebruik van (browser-)caches. Als we een pagina openen, wordt er een kopie van die pagina opgeslagen in deze cache om er voor te zorgen dat als we later opnieuw deze pagina openen, dit sneller gaat. Het probleem hier is dat als de pagina verandert, we een oude versie van de pagina in onze cache hebben. We moeten bijgevolg opnieuw de cache aanpassen zodat deze terug up-to-date is.

Uiteindelijke consistentie is de enige consistentie-eis in NoSQL databases. Het zegt dat er consistentie wordt bereikt in een bepaald punt van de toekomst. We weten niet wanneer, maar we zijn er zeker van dat het ooit gebeurt [15, 44]. Voor meer informatie verwijzen we terug naar Paragraaf 3.3.

Het grote verschil tussen ACID en BASE is dat er in ACID veel belang wordt gehecht aan consistentie, terwijl BASE zijn pijlen op beschikbaarheid richt. Daarnaast focust BASE zich ook sterk op de schaalbaarheid.

Naast uiteindelijke consistentie zijn er nog verschillende andere consistentiemodellen. In de volgende Paragraaf bespreken we causale consistentie.

3.4 Causale Consistentie

Causale consistentie, in het Engels *causal consistency*, steunt op afhankelijkheden tussen verschillende operaties en wordt gecategoriseerd onder de zwakke consistenties. Het eerste voordeel heeft te maken met de aanwezigheid van partities. Dit consistentiemodel biedt namelijk de best mogelijke consistentie in de aanwezigheid van partities en is daarom zeer belangrijk in gedistribueerde systemen. Het tweede belangrijke punt is dat causale consistentie een zekere vorm van *semantiek* verzorgt. Het zorgt er voor dat zaken die voor mensen niet logisch zijn, niet kunnen voorkomen [8]. Een voorbeeld uit ons eigen leven zou kunnen zijn dat we onze auto eerst moeten starten alvorens de auto kan rijden. Het zou niet logisch zijn als we eerst rijden en dan pas de auto zouden starten. Als we dit doortrekken naar voorbeelden die te maken hebben met deze thesis, kunnen we stellen dat de data pas zichtbaar wordt gemaakt als ook de afhankelijkheden van deze data aanwezig zijn. We komen hier later in deze Paragraaf nog op terug.

3.4.1 Twee Soorten Causale Consistentie

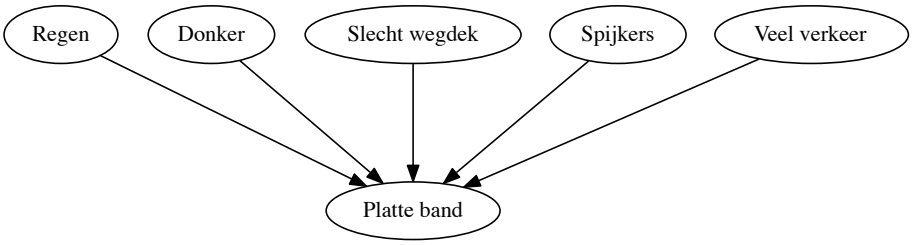
Om de twee soorten consistenties uit te leggen, introduceren we eerst het begrip *causale graaf* of in het Engels *causality graph*. Een causale graaf geeft afhankelijkheden weer tussen verschillende schrijfoperaties. Als er dus een boog is van schrijfoperatie A naar schrijfoperatie B, dan is schrijfoperatie B afhankelijk van schrijfoperatie A. Deze grafen worden gebruikt bij causale consistentie om de afhankelijkheden weer te geven.

Als we over causale consistentie praten, kunnen we een onderscheid maken tussen enerzijds *potentiële consistentie*, in het Engels *potential consistency*, en anderzijds *expliciete consistentie*, in het Engels *explicit consistency*. Bij potentiële consistentie kan elke nieuwe schrijfoperatie afhangen van alle andere schrijfoperaties. Als we denken aan voorbeelden in de sociale media, kan een reactie van een bepaalde persoon beïnvloed zijn door honderden andere statussen, maar in de praktijk zal de oorspronkelijke afhankelijkheid van de reactie bij één status van een andere persoon liggen (de reactie hoort namelijk bij een status). Dit soort van consistentie is daarom te algemeen om gebruikt te worden in dergelijke grote systemen omdat er simpelweg te veel afhankelijkheden zijn, waardoor het controleren van deze afhankelijkheden te intensief zal zijn. Toch kan potentiële consistentie dienst doen om bijvoorbeeld te debuggen aangezien we alle afhankelijkheden ter beschikking hebben en zo dus eenvoudig een eventuele fout kunnen opsporen.

De andere vorm van consistentie is de expliciete consistentie. Deze consistentie zal rekening houden met de effectieve semantische afhankelijkheden en zal daarom maar een subset van de potentiële consistentie bevatten. Dit heeft als gevolg dat de diepte en de graad van de causale graaf sterk wordt gereduceerd. Het is niet altijd even duidelijk wat precies semantisch van elkaar afhankelijk is, daarom kan de ontwerper van het systeem kiezen wat afhankelijk is van elkaar [8].

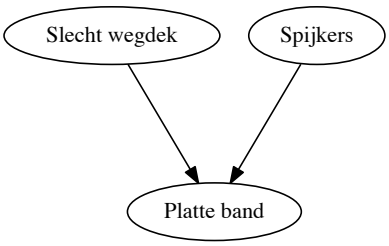
We geven een voorbeeld uit het dagelijkse leven om dit te illustreren. In Figuur 3.5 geven we een voorbeeld van een situatie op de weg terwijl we met de fiets aan het rijden zijn. Tijdens het rijden, krijgen we een platte band. Als we met potentiële consistentie te maken krijgen, dan zou de platte band *mogelijk*

het gevolg kunnen zijn van: de regen, de spijkers op de baan, het drukke verkeer, het slechte wegdek en het feit dat het donker is.



Figuur 3.5: Een causale graaf (potentiële consistentie).

In Figuur 3.6 zien we daarentegen dat we expliciete consistentie gebruiken. We voegen met andere woorden semantiek toe en daarom weten we dat het feit dat het donker is, er veel verkeer is en dat het regent niets te maken kan hebben met het feit dat we een platte band krijgen. Zoals te zien in Figuur 3.6 is de graaf kleiner geworden.



Figuur 3.6: Een causale graaf (expliciete consistentie).

Om dit voorbeeld concreter te maken geeft Bailis et al. [8] een voorbeeld over Twitter. Een Twitter conversatie is gemiddeld zo'n elf tweets lang. Als we dit proberen te meten door gebruik te maken van potentiële consistentie van een jaar lang tweets, krijgen we een causale graaf die een orde van negen groter is. Er is dus een groot verschil tussen potentiële en expliciete consistentie.

3.4.2 Terminologie

Bij causale consistentie wordt de uitkomst van een operatie pas getoond na hun oorzaken. We kunnen het belang van causale consistentie uitleggen aan de hand van een voorbeeld op Facebook [8]. Stel dat de volgende situatie zich voordoet:

1. Gert post een status X waarin hij schrijft: "Ik ben gebuisd!"

2. Even later update Gert zijn post X naar X': "Oef! Het was een fout van de universiteit. Ik ben geslaagd!"

3. Hierop reageert Bram met status Y: "Proficiat, ik ben blij voor jou!"

Als we deze conversatie lezen zoals hij hier is weergegeven, is dit een logische conversatie. De volgorde van de conversatie is in dit geval: $X \rightarrow X' \rightarrow Y$. Stel nu dat een derde persoon Jonas de conversatie leest en de causale afhankelijkheden zijn niet gerespecteerd. Jonas leest het volgende:

1. Gert post een status X waarin hij schrijft: "Ik ben gebuisd!"

2. Hierop reageert Bram met status Y: "Proficiat, ik ben blij voor jou!"

Jonas zou nu kunnen denken dat Bram blij is dat Gert gebuisd is. Bij het respecteren van de afhankelijkheden zou het nooit mogelijk geweest zijn om Y weer te geven zonder X'. Het kan dus zeer belangrijk zijn om hiermee rekening te houden.

Deze verschillende versies (van de staat van het systeem) die gebruikers lezen, hangen dus af van een bepaalde volgorde, namelijk de *gebeurt-voor relatie* of in het Engels de *happens-before relation*. De gebeurt-voor relatie stellen we voor door de volgende notatie:

$$X \rightarrow Y$$

We lezen X gebeurt voor Y, dit wil simpelweg zeggen dat eerst X uitgevoerd werd en daarna Y (we stellen hier X en Y voor als schrijfoperaties).

We geven een voorbeeld hoe deze gebeurt-voor relatie in de praktijk wordt gebruikt: we veronderstellen dat we drie versies hebben van een bepaald data object: v_1, v_2, v_3, met de relaties $v_1 \rightarrow v_2 \rightarrow v_3$.

Als we nu twee keer achter elkaar een leesoperatie uitvoeren, dan kunnen we de eerste maal v_1 lezen en de tweede maal v_3. Wat niet kan, is dat we bijvoorbeeld eerst v_3 lezen en vervolgens v_2. In een causaal systeem is dit niet mogelijk, omdat we ons aan de volgorde van de gebeurt-voor relatie moeten houden. We kunnen wel nieuwere versies lezen, maar we kunnen niet "terug in de tijd gaan" [8].

De gebeurt-voor relatie heeft drie eigenschappen [8]:

- Thread-of-execution: Als een gebruiker eerst a en dan b uitvoert, dan $a \rightarrow b$
- Reads-from: als a een schrijfoperatie is en b een leesoperatie, die de waarde van de schrijfoperatie teruggeeft, dan $a \rightarrow b$.
- Transitivity: als $a \rightarrow b$ en $b \rightarrow c$, dan $a \rightarrow c$

3.4.3 Causale$^+$ Consistentie

In deze Paragraaf bespreken we een lichte variant op de gewone causale consistentie, namelijk de *causale$^+$ consistentie* of *convergente causale consistentie*. Als we in het vervolg van dit Hoofdstuk over causale consistentie hebben, hebben we het specifiek over deze convergente causale consistentie. Deze vorm van

consistentie is de causale consistentie die we in het begin van Paragraaf 3.4 besproken hebben samen met een *convergentie* eis. Met convergentie bedoelen we dat alle nodes naar dezelfde staat zullen evolueren. We merken op dat deze convergentie mogelijk wordt gemaakt door het feit dat updates naar alle nodes in het netwerk worden gestuurd. Voor meer informatie hierover verwijzen we naar Paragraaf 3.4.4. Als updates stoppen, zien alle nodes (net zoals bij uiteindelijke consistentie) dezelfde data. Deze eigenschap is nodig aangezien causale consistentie geen liveness garantie heeft. De safety garantie heeft causale consistentie daarentegen wel. Herinner dat dit het tegenovergestelde is van uiteindelijke consistentie. Uiteindelijke consistentie heeft namelijk wel een liveness garantie, maar geen safety garantie. Zonder de convergentie kan een systeem perfect voldoen aan causale consistentie, zonder ook maar één schrijfoperatie te delen met andere nodes. Dit leidt tot safety, maar verwaarloost (zoals eerder vermeld) de liveness garantie [8].

3.4.4 Implementatie

We geven eerst het algemeen idee bij potentiële causale consistentie. In Paragraaf 3.4.6 bespreken we de hogere efficiëntie ten gevolge van expliciete causaliteit. De normale vorm van werken, is dat elke node van een systeem een deel van de schrijfoperaties onderhoudt en deze schrijfoperaties vervolgens veilig en lokaal kan uitvoeren en lezen. Als er nu een nieuwe schrijfoperatie arriveert, controleert de node de *metadata* van de nieuwe schrijfoperatie om te kijken of alle afhankelijkheden voor deze operatie in orde zijn. We geven een eenvoudig voorbeeld van een afhankelijkheid: als een bepaalde gebruiker een waarde $A = 20$ leest en vervolgens een waarde $B = 30$ schrijft, dan kan een andere gebruiker niet $B = 30$ lezen en vervolgens een oudere versie dan $A = 20$ lezen. De achterliggende werking gaat dus als volgt: als de schrijfoperatie $B = 30$ aankomt bij een node, dan moet deze node wachten tot hij de schrijfoperatie $A = 20$ heeft gehad. Dit wordt allemaal geregistreerd in de metadata van de schrijfoperatie $B = 30$ [9]. Als de afhankelijkheden in orde zijn, is er geen probleem en kan de node de schrijfoperatie lokaal uitvoeren. In het andere geval, als er afhankelijkheden niet aanwezig zijn, wacht de node met de schrijfoperatie totdat al de bijhorende afhankelijkheden aanwezig zijn. Dergelijke schrijfoperaties worden zichtbaar voor de gebruikers vanaf het moment dat de schrijfoperatie lokaal is uitgevoerd [8].

Causale consistentie kunnen we zien als het bewaren van de data integriteit door gebruik te maken van constraints die steunen op de gebeurt-voor relatie. Voor een goed voorbeeld verwijzen we terug naar het begin van Paragraaf 3.4.2 waar we het hebben gehad over statussen bij Facebook. Hier hebben we gezien dat elk bericht een aparte schrijfoperatie is en dat het belangrijk is dat berichten in een juiste volgorde worden getoond aan de gebruiker. Een bericht is dus afhankelijk van een vorig bericht en deze volgorde moet dus worden nageleefd.

Het controleren van afhankelijkheden kan enerzijds worden gedaan door de bovenliggende applicatie en anderzijds door de database zelf. Voor de applicatie is dit een moeilijk en duur proces. Dit komt omdat bij elke leesoperatie dan alle transitieve afhankelijkheden moeten gecontroleerd worden. Merk op dat als een opvolgende afhankelijkheid binnenkomt bij een node, dat dit niet altijd wil zeggen dat de voorgangers van deze afhankelijkheid al zijn binnengekomen.

Nadat een schrijfoperatie is binnengekomen bij een node, zal deze node (na

het controleren van de afhankelijkheden) lokaal de schrijfoperatie uitvoeren. Daarna stuurt deze node deze schrijfoperatie door naar alle andere nodes in het netwerk via een *reliable broadcast* [8]. Merk op dat dit proces wordt uitgevoerd door de database. Het reliable broadcast protocol zorgt ervoor dat een sequentie van berichten of operaties wordt doorgestuurd naar een set van computers in het systeem. Een garantie is dat deze berichten uiteindelijk aankomen (net zoals bij uiteindelijke consistentie). Uiteraard willen we dat deze berichten zo snel mogelijk aankomen [26]. Deze operaties worden vervolgens door deze nodes lokaal gebufferd. De gebufferde schrijfoperaties worden pas uitgevoerd vanaf het moment dat alle afhankelijkheden voor deze schrijfoperatie voldaan zijn. De convergentie eis zorgt ervoor dat alle schrijfoperaties van de broadcast uiteindelijk hun bestemming bereiken en dus ook uiteindelijk worden uitgevoerd door de desbetreffende node. Elke node houdt de causale geschiedenis bij van zijn eigen schrijfoperaties en voegt een samenvatting hiervan toe aan de schrijfoperaties. Op deze manier krijgen andere nodes ook de nodige informatie over de de rest van het systeem. Merk op dat deze geschiedenis heel groot kan worden, maar dat er technieken zijn om deze te reduceren. Hier wordt verder op ingegaan in Paragraaf 3.4.5 [8].

3.4.5 Mogelijke Gevaren

Zoals werd geïllustreerd, is causale consistentie een mogelijk interessante vorm van consistentie in gedistribueerde systemen en dus ook in NoSQL systemen. Ondanks de verschillende voordelen, moeten we rekening houden met het feit dat deze vorm van consistentie moeilijk te schalen is. Het waarom hiervan bespreken we in het vervolg van deze Paragraaf. Dit is een niet te onderschatten probleem aangezien we dit soort consistentie ook zouden willen toepassen op NoSQL databases die zeer schaalbaar moeten zijn.

Om de rest van deze Paragraaf goed te kunnen volgen, beginnen we met het uitleggen van enkele belangrijke begrippen. Het eerste begrip is *throughput*. Dit is de snelheid waarmee nodes nieuwe schrijfoperaties kunnen genereren. We kunnen dit bijvoorbeeld uitdrukken als X schrijfoperaties per seconde.

Een tweede begrip dat we definiëren is *visibility latency*. Als een node een bepaalde schrijfoperatie niet kan uitvoeren, moet deze node wachten op de bijhorende afhankelijkheden. De tijd die deze node moet wachten, noemen we de visibility latency.

Het volgende begrip is *apply capacity*. Dit is de snelheid waarmee een node afhankelijkheden kan controleren. Ook hier kunnen we dit bijvoorbeeld uitdrukken als X afhankelijkheden controleren per seconde.

We merken op dat de visibility latency afhankelijk is van enerzijds de vertragingen die op het netwerk zitten en anderzijds de apply capacity van een node. Als we ons in een situatie bevinden waar de nodes op een sneller tempo nieuwe schrijfoperaties aanmaken dan dat ze verwerkt kunnen worden, dan zal de visibility latency enorm stijgen door de *onstabiele queues*. Met onstabiele queues bedoelen we dat schrijfoperaties vastzitten in de buffer omdat ze moeten wachten op hun afhankelijkheden.

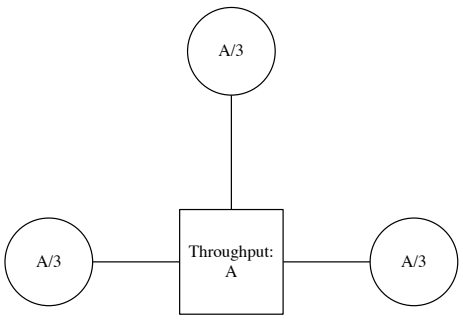
Er moet een goede balans gevonden worden tussen de throughput en de visibility latency [8]. Dit hangt onder meer af van:

- het aantal nodes.

- de snelheid waarmee elke node afhankelijkheden kan controleren en schrijfoperaties kan uitvoeren.

Problemen met Schalen

Een nadeel van causale consistentie is het feit dat de globale schrijf throughput wordt gelimiteerd door de broadcast die moet gebeuren om alle schrijfoperaties naar iedereen te propageren. We geven een voorbeeld om dit te verduidelijken. Stel dat we twee nodes hebben die elk een apply-capacity hebben van A. Herinner dat apply capacity de snelheid is waarmee een node afhankelijkheden kan controleren. Om te voorkomen dat deze nodes hun werk niet gedaan krijgen, limiteren we de write throughput ook tot A. Op deze manier vermijden we de onstabiele queus waardoor verschillende schrijfoperaties moeten wachten op hun afhankelijkheden. Elke node kan bijgevolg ook maar $\frac{A}{2}$ nieuwe schrijfoperaties broadcasten aangezien de throughput gelimiteerd is tot A. Als we nu een even sterke node toevoegen, zal de situatie niet verbeteren, elke node zal in dit geval gelimiteerd worden tot $\frac{A}{3}$ nieuwe schrijfoperaties. Een schematische voorstelling van deze situatie zien we in Figuur 3.7. We zien dat elke node een verwerkingscapaciteit van $\frac{A}{3}$ schrijfoperaties heeft en dat in het centraal vierkant de totale throughput wordt weergegeven. Als we dit verder veralgemenen tot N nodes krijgen we een situatie waarin elke node slechts $\frac{A}{N}$ nieuwe schrijfoperaties kan verwerken. Het is duidelijk dat de sterkte van het netwerk afhangt van apply capacity van de zwakste node.



Figuur 3.7: Drie nodes die elke $\frac{A}{3}$ schrijfoperaties kunnen verwerken.

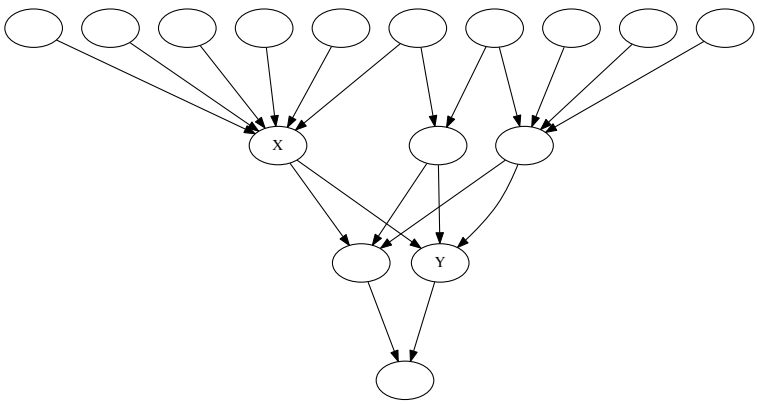
Als we willen bekomen dat elke node een constant aantal schrijfoperaties mag blijven genereren, dan zitten we met een sterke groei wat betreft de apply capacity van de nodes. We illustreren dit met een eenvoudig voorbeeldje. Stel dat we twee nodes hebben die elk 100 nieuwe schrijfoperaties per seconde genereren. Volgens onze vorige discussie zou elke node dan een apply capacity moeten hebben van 200 schrijfoperaties per seconde. Als we vervolgens een derde node willen toevoegen en we willen dat elke node 100 nieuwe schrijfoperaties per seconde kan blijven genereren, dan zou elke node in plaats van 200 schrijfoperaties per seconde, moeten upgraden naar 300 schrijfoperaties per seconde. Als we even een rekensom maken, zien we dat:

- elke node een upgrade van 50% maakt (200 schrijfoperaties per seconde $\rightarrow$ 300 schrijfoperaties per seconde).
- de toegevoegde node voor een toevoeging van 75% apply capacity zorgt (oorspronkelijk 2$\times$ 200 schrijfoperaties per seconde en we voegen een node van 300 schrijfoperaties per seconde toe).
- de totale server capaciteit een stijging van 125% kent (2 $\times$ 200 schrijfoperaties per seconde $\rightarrow$ 3 $\times$ 300 schrijfoperaties per seconde).

Vaak wordt er gedacht dat het toevoegen van extra nodes het probleem van de throughput zal oplossen. Zoals we net hebben aangetoond, is dit niet het geval en zullen we om convergente causale consistentie na te streven alle nodes mee moeten upgraden, aangezien het netwerk maar zo snel en goed is als zijn zwakste schakel. Daarnaast hebben we gezien dat het effectief mee upgraden van alle nodes zorgt voor een quadratische stijging wat betreft de server capaciteit. We zouden ook datacenters met dezelfde capaciteit of met minder capaciteit kunnen toevoegen, maar dit zal leiden tot situaties die niet zo gunstig zijn voor de werking van het systeem. We creëren in dergelijke situaties een hoge visibility latency, met de eerder genoemde problemen tot gevolg [8].

Problemen met de Potentiële Causale Graaf

Zoals hierboven duidelijk is geworden, hangt de throughput van het systeem af van de traagste node in het netwerk. De snelheid waarmee deze node vervolgens zijn schrijfoperaties kan verwerken, hangt dan weer af van de grootte van de causale graaf. De graad van elke node in de causale graaf bepaalt hoeveel afhankelijkheidscontroles er moeten gebeuren om te kijken of alle afhankelijkheden van deze node in orde zijn. De diepte en de connectiviteit van de causale graaf bepaalt hoeveel controles er tegelijk uitgevoerd kunnen worden. In Figuur 3.8 zien we een voorbeeld van een causale graaf. We merken op dat dit een kleine graaf is ten opzichte van de realiteit, maar Figuur 3.8 volstaat om het idee duidelijk te maken.



Figuur 3.8: Een voorbeeld van een potentiële causale graaf.

In Figuur 3.8 zien we een schrijfoperaties X en een schrijfoperatie Y. Schrijfoperatie X moet zes afhankelijkheden controleren en schrijfoperatie Y slechts drie.

Zoals reeds opgemerkt is dit een zeer klein voorbeeld in vergelijking met de realiteit. In de praktijk kunnen potentiële causale grafen miljoenen schrijfoperaties bevatten aangezien elke operatie die een gebruiker uitvoert potentieel zijn schrijfoperatie kan beïnvloeden. Als we bijvoorbeeld op Twitter of Facebook eerst 30 reacties van vrienden lezen en we posten vervolgens zelf een reactie, dan zou onze causale graaf 30 (potentiële) afhankelijkheden hebben voor deze schrijfoperatie. Merk op dat elk van deze 30 afhankelijkheden misschien ook 30 afhankelijkheden hebben en die afhankelijkheden ook weer 30 afhankelijkheden hebben. Dit proces kan nog even doorgaan, wat leidt tot een immens grote causale graaf.

Het is dus duidelijk dat de graad en diepte van de causale graaf een sterke invloed hebben op de apply capacity van een node, aangezien elke schrijfoperatie enorm veel (potentiële) afhankelijkheden heeft. Deze apply capacity beperkt vervolgens ook de throughput.

Een manier om de grootte van de causale graaf kleiner te maken, is het reduceren van de afhankelijkheden. Als we ons in een situatie bevinden waarin elke node dezelfde versie (stabiele versie) van bepaalde data ter beschikking heeft, moeten we deze niet meer in de metadata plaatsen voor nieuwe schrijfoperaties. Inderdaad, elke afhankelijkheid van die data is dan reeds voldaan op elke node [8].

3.4.6 Van Potentiële naar Expliciete Causale Consistentie

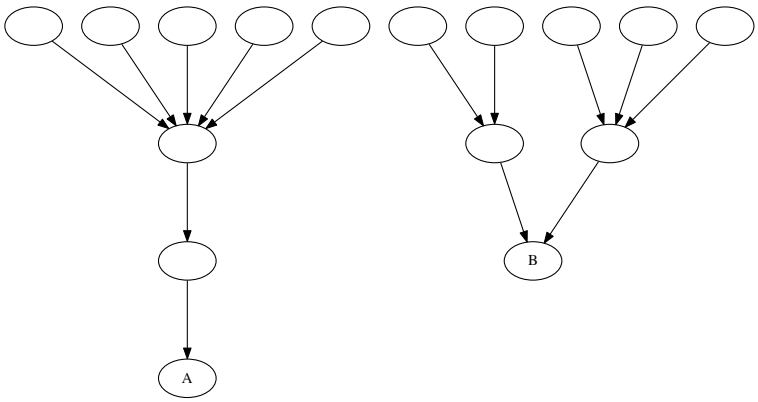
Potentiële causale consistentie is niet altijd even ideaal om te gebruiken. Het grootste nadeel is dat de causale graaf al snel heel groot wordt. In Paragraaf 3.4.1 hebben we het reeds gehad over een andere vorm van causale consistentie, namelijk expliciete consistentie. Bij dit soort van consistentie gaan we niet alle operaties beschouwen als een mogelijk afhankelijkheid, maar gaan we effectief kijken welke afhankelijkheden een bepaalde schrijfoperatie heeft. We maken hier gebruik van de semantiek van de operaties, in Paragraaf 3.4.1 hebben we een voorbeeld gegeven hierover. Dit biedt als voordeel dat de causale graaf immens kleiner wordt, maar biedt nog steeds geen oplossing voor het feit dat we voor elke schrijfoperatie een broadcast moeten uitsturen [8].

Een kleinere graad van een schrijfoperatie in de causale graaf wil zeggen dat elke schrijfoperatie minder afhankelijkheden heeft. Het gevolg hiervan is dat de afhankelijkheidscontrole veel sneller kan gaan. Als we moeten kiezen tussen een potentiële causale graaf waar er duizend afhankelijkheden moeten worden gecontroleerd of een expliciete causale graaf waar maar vier afhankelijkheden gecontroleerd moeten worden, is de keuze snel gemaakt. Dit leidt tot een snellere afhankelijkheidscontrole, dus bijgevolg ook een grotere apply capacity en een verminderde visibility latency.

Naast de snelheidswinst is er ook een voordeel wat betreft de overhead van de metadata die telkens opnieuw moet worden meegestuurd. Minder afhankelijkheden zorgen voor minder metadata en dus ook minder overhead.

Als laatste beschikken we ook over een voordeel omdat bij een kleinere graaf er minder schrijfoperaties aanwezig zijn. In een expliciete causale graaf zien we ook een duidelijk verschil tussen de connectiviteit. In een potentiële causale

graaf is de kans groot dat elke schrijfoperatie geconnecteerd is met elke andere schrijfoperatie. In een expliciete causale graaf is de kans groter dat we te maken krijgen met disjuncte subgrafen. Dit heeft als voordeel dat we met onafhankelijke grafen te maken hebben die we ook tegelijk kunnen afhandelen [8]. In Figuur 3.9 zien we een expliciete variant van Figuur 3.8.



Figuur 3.9: Een voorbeeld van een expliciete variant van Figuur 3.8.

We zien duidelijk dat er veel minder connecties zijn en dat we te maken hebben met twee disjuncte grafen. Als we bijvoorbeeld de afhankelijkheden controleren van A en we moeten wachten op een bepaalde andere afhankelijkheid kunnen we tegelijkertijd de afhankelijkheden van B controleren. Als deze in orde zijn kunnen we deze schrijfoperatie verwerken.

3.5 Conclusie

In dit Hoofdstuk hebben we twee soorten zwakke consistenties besproken, namelijk uiteindelijke consistentie en causale consistentie.

Bij uiteindelijke consistentie is het belangrijk dat we weten dat er een liveness garantie is, maar geen safety garantie. Deze safety garantie moeten we zelf proberen op te lossen. Verder zijn er nog enkele belangrijke principes uitgelegd die helpen om uiteindelijke consistentie te bereiken. Zo is er de CALM stelling die zegt dat als een operatie logisch monotoon is, deze operatie dan ook uiteindelijk consistent is. Een ander belangrijke techniek is het gebruik van CRDTs. Dit soort operaties zorgen er immers voor dat gelijktijdige operaties automatisch goed worden afgehandeld. Daarnaast hebben we het acroniem BASE besproken. Dit is in feite de tegenhanger van ACID, dat gebruikt wordt bij relationele databases.

Ten slotte hebben we causale consistentie besproken. Hierbij hebben we twee soorten gezien. Enerzijds de potentiële causale consistentie die rekening houdt met alle schrijfoperaties om afhankelijkheden te controleren en anderzijds de expliciete causale consistentie die rekening houdt met de semantiek om afhankelijkheden te controleren. Deze afhankelijkheden worden geleid door de

gebeurt-voor relatie en worden voorgesteld als een causale graaf. Een belangrijk nadeel dat we hebben gezien bij dit soort consistentie is het feit dat de efficiëntie van het systeem afhangt van de zwakste schakel in het netwerk.

Hoofdstuk 4

NoSQL

4.1 Inleiding

Tegenwoordig moeten we in staat zijn om telkens meer en meer data te kunnen verwerken. In onze huidige samenleving worden er allerlei soorten data bijgehouden, dit kan gaan van koopgedrag van mensen tot foto's van het heelal. Het feit is dat deze data heel groot wordt en dat het moeilijker wordt om deze data te kunnen verwerken. Een mogelijke oplossing die tegenwoordig veel naar voren wordt geschoven, is het gebruik van NoSQL databases.

In dit Hoofdstuk geven we algemene informatie over NoSQL. Hierbij bespreken we enkele eigenschappen van NoSQL databases en geven we een uitleg waarom ze tegenwoordig zo populair zijn.

Een ander interessant feit is dat er twee kampen ontstaan, namelijk de strekking die pro NoSQL is enerzijds en anderzijds de strekking die contra NoSQL is.

4.2 Wat is NoSQL

De term *NoSQL* dook voor het eerst op in 1998 toen Carlo Strozzi zijn RDBMS uitbracht. Hij categoriseerde zijn RDBMS als NoSQL om zijn RDBMS te onderscheiden van de rest van de databases. Hij gebruikte nog steeds het relationeel model, maar hij stelde zijn SQL interface niet ten toon. Merk op dat deze betekenis van NoSQL niets te maken heeft met de betekenis zoals we deze tegenwoordig kennen [25].

Als we de laatste jaren over *NoSQL* spreken, dan hebben we het over databases die anders zijn dan het relationeel model. Zo gebruiken ze andere manieren om hun data op te slaan en gebruiken ze minder vaak SQL om hun data te queryen. NoSQL kan op twee manieren vertaald worden. De eerste betekenis is "No SQL". Hiermee bedoelen we dat er geen SQL wordt gebruikt om de data te queryen. De andere betekenis is "Not only SQL", waarmee wordt bedoeld dat het relationeel model niet altijd de beste oplossing is voor problemen. In de literatuur [11, 21] vinden we beide betekenissen terug, hoewel er meer wordt geneigd naar de "Not only SQL" betekenis [25].

We keren nog even terug naar de geschiedenis van de databases zoals besproken in Paragraaf 1.2. Na de echte lancering van de relationele databases

werd er nog gewerkt aan andere database types, zoals de XML databases en de object databases. Deze databases zijn echter nooit succesvol geworden, aangezien er opnieuw geen theoretische onderbouwing was en omdat er maar weinig voordelen waren ten opzichte van de relationele databases [32].

Tegenwoordig zijn de grootste spelers op de markt:

- IBM's DB2
- Microsoft SQL Server
- Oracle Database

Deze databases volgen de laatste trends. Neem nu het voorbeeld van de XML databases. Echte "native" XML databases zijn nooit doorgebroken, maar de grote ontwikkelaars zorgen voor een kleine plug-in waardoor ze ook met XML data kunnen werken in hun databases. Een reden om dan daadwerkelijk over te stappen naar een "native" XML database moet dan al zeer overtuigend zijn. We komen hier later nog op terug als we het hebben over de pro's en contra's van NoSQL databases in Paragraaf 4.4. Grote ontwikkelaars doen net hetzelfde bij NoSQL databases, ze voegen namelijk eigenschappen van NoSQL databases toe aan hun relationele databases.

Het is tegenwoordig echt een "hype" om over NoSQL databases te spreken. Er zijn veel verschillende soorten NoSQL databases, maar slechts enkele zijn echt commercieel gemaakt. Deze NoSQL databases komen grotendeels uit de industrie, denk maar aan de BigTable van Google of de DynamoDB van Amazon [32].

De keuze tussen een relationele database of een NoSQL database is natuurlijk afhankelijk van verschillende zaken. We bespreken de voor- en nadelen in Paragraaf 4.4.

Zoals we in Paragraaf 1.2 hebben gezien, waren de relationele databases zeer populair in de jaren '80. De reden hiervoor is dat de data die verwerkt moest worden onder de categorie van commerciële data viel. Deze data kennen we onder de naam *Online Transaction Processing* (*OLTP*). De focus van deze data ligt bij het wegschrijven naar een database.

Dit is in tegenstelling tot de *data warehouses*, deze databases hechten veel belang aan complexere queries en hebben een focus op het lezen van data.

De NoSQL databases hebben daarentegen van beide varianten kenmerken. Toepassingen waar NoSQL wordt gebruikt, zijn bijvoorbeeld sociale netwerken zoals Facebook en Twitter. Hier ligt de focus ergens tussen OLTP en data warehouses. Aangezien er miljoenen posts worden geplaatst op deze sociale netwerksites is zowel het lezen als het schrijven van de data van zeer groot belang [32].

4.3 Mogelijke Oorzaken NoSQL

In deze Paragraaf geven we enkele redenen waarom NoSQL zo populair is. We bespreken de volgende oorzaken:

- big Data
- structuur van de data
- drang naar parallellisatie

4.3.1 Big Data

Het woord *big data* wil in feite zeggen dat we met enorm veel data te maken hebben. Deze data kan overal vandaan komen, denk bijvoorbeeld aan de foto's die gebruikt worden in Google Streetview, de foto's die gemaakt worden van de ruimte of data van sociale netwerken zoals Facebook.

Big data is belangrijk voor zowel de academische wereld als voor de professionele wereld. De academische wereld vergaart enorm veel data die geanalyseerd moet worden. Denk maar aan de informatie over genen die biologen willen onderzoeken. In de professionele wereld gebruikt men big data om winst te maken, bijvoorbeeld door patronen te zoeken in het koopgedrag van klanten bij grote winkelketens [32].

Naast het vergaren van data speelt het *Web 2.0* ook een belangrijke rol in het creëren van veel informatie. Web 2.0 refereert naar het *read-write web*. Dit web is een interactief web waar gebruikers informatie kunnen aanpassen en kunnen lezen. Denk maar aan de grote voorbeelden zoals Wikipedia en Facebook. Het is vanzelfsprekend dat dit tot enorm veel data leidt [3].

Al deze data moet op een efficiënte manier kunnen worden opgeslagen en worden gequeried. Stel dat we petabytes aan informatie hebben en dat we dit met een relationele database moeten onderzoeken. Stel dat deze informatie is verdeeld over duizenden relaties. Een join van al deze relaties zou een heel zware taak zijn om uit te voeren. Men zal hier alternatieve methodes moeten gebruiken [32].

Big data wordt ook vaak voorgesteld aan de hand van de "drie v's". Deze eigenschappen karakteriseren verschillende aspecten van big data [45]:

1. volume (Engels: volume)
2. snelheid (Engels: velocity)
3. variëteit (Engels: variety)

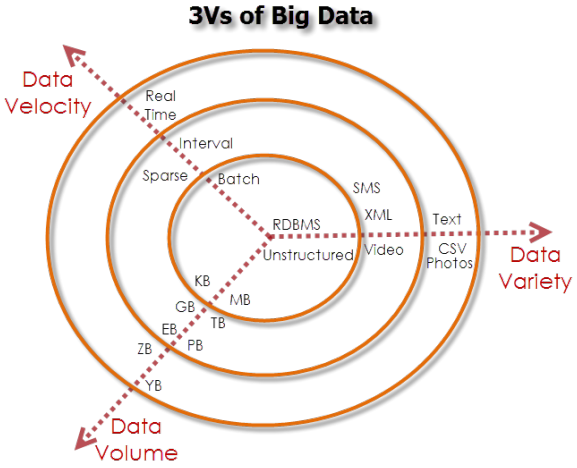
De eerste eigenschap is deze van *volume*. Met volume bedoelen we de hoeveelheid data. Tegenwoordig zien we een exponentiële groei wat betreft de hoeveelheid van data. Data kan zoals eerder vermeld allerlei soorten vormen aannemen: het kan gaan van tekst tot muziek en video's. Er moet dus een oplossing gezocht worden om dit allemaal efficiënt te kunnen opslaan en om de data ook nog efficiënt te queryen.

De tweede eigenschap is *snelheid*. Dit heeft niets te maken met de snelheid waarmee data wordt verwerkt, maar wel met de snelheid waarmee data wordt aangemaakt. Vroeger was data die een dag oud was nog recent. Tegenwoordig kunnen we bij sommige applicaties al over oude data spreken na bijvoorbeeld een uur. Denk maar aan de sociale media waar statussen of tweets om de minuut kunnen veranderen. We hebben bijna altijd te maken met real time updates.

De derde en laatste eigenschap is *variëteit*. Deze eigenschap heeft te maken met het feit dat we tegenwoordig data kunnen vinden in allerlei formaten. Als we denken aan NoSQL databases, moeten we dus in principe zo algemeen mogelijk zijn. Er moet een mogelijkheid zijn om enerzijds eenvoudige zaken zoals tekst op te slaan en anderzijds ook ingewikkeldere zaken zoals muziek of filmpjes. We kunnen dit bijvoorbeeld doen door een BLOB type te gebruiken, waar we eender welke byte string in kunnen opslaan. Het is dus belangrijk dat we al deze data

eerst moeten omzetten naar een afgesproken formaat om er gemakkelijk mee te kunnen werken [45].

In Figuur 4.1 zien we een schematische voorstelling van de drie v's met nog extra informatie.



Figuur 4.1: De drie karakteristieken van big data [45].

4.3.2 Structuur van de Data

Tegenwoordig slaan we echt alles op wat we tegenkomen. Het probleem hierbij is dat deze data verschillende structuren kan hebben. Deze verschillende structuren kunnen we kaderen in de variëteit eigenschap van big data, zoals besproken in Paragraaf 4.3.1. Bij relationele databases zijn we gebonden aan een vast schema. We kunnen met andere woorden niet zomaar attributen gaan toevoegen zonder het schema en de bestaande data aan te passen. Een ander probleem ontstaat wanneer we met een *sparse database* werken. In dergelijke databases zitten we met het probleem dat er verschillende velden niet zijn ingevuld en bijgevolg er dus nodeloos velden worden gereserveerd. We kunnen dit best illustreren met een klein voorbeeldje. Stel dat we ons op een website kunnen registreren en stel dat de registratie bestaat uit de velden zoals weergegeven in Tabel 4.1.

Informatie	**Type**
Email	Verplicht
Naam	Optioneel
Leeftijd	Optioneel
Woonplaats	Optioneel

Tabel 4.1: Voorbeeld van een registratie op een website.

Bij registratie wordt enkel het emailadres verplicht ingegeven. Over de rest van de informatie hebben we in feite geen controle. Het kan dus voorkomen dat er verschillende velden leeg blijven. In de database zijn deze velden gereserveerd

en bijgevolg wordt deze geheugenruimte niet efficiënt gebruikt. Een voorbeeld van een registratiedatabase zien we in Tabel 4.2.

Email	Naam	Leeftijd	Woonplaats
John@gmail.com	John	32	Zonhoven
Pieter@hotmail.com	NULL	23	NULL
Hanne@icloud.com	Hanne	NULL	NULL
Ansel@gmail.com	NULL	NULL	NULL

Tabel 4.2: Voorbeeld van een registratie database.

Dit probleem kunnen we verhelpen door gebruik te maken van dynamische schema's. We kunnen dan een attribuut bij één tuple toevoegen zonder dit attribuut toe te voegen aan de volledige database. We kunnen bijvoorbeeld enkel het emailadres "Ansel@gmail.com" toevoegen in de database zonder de drie andere attributen, terwijl we bij "John@gmail.com" alle attributen kunnen toevoegen [32].

4.3.3 Drang naar Parallellisatie

We zijn op een punt gekomen in de technologie waarop de kloksnelheden van de chips niet meer zo snel verhoogd worden als vroeger [39]. Om computers sneller te maken steken we bijvoorbeeld vier processors in onze computer zodat de computer deze zo efficiënt mogelijk kan gebruiken. Met efficiënt bedoelen we dat we berekeningen in parallel kunnen laten lopen. We laten met andere woorden elke processor een stukje berekenen. Dit principe wordt ook doorgetrokken naar databases.

In plaats van telkens verticaal te schalen en dus telkens betere databases aan te kopen, schakelen we over naar horizontaal schalen. Met andere woorden kopen we verschillende kleinere databases aan en we laten deze databases in parallel werken. In tegenstelling tot de klassieke relationele databases, zijn NoSQL databases speciaal ontworpen voor dit soort parallellisatie [32].

4.4 Argumenten Relationele Databases vs. NoSQL Databases

Als we informatie over NoSQL opzoeken, vinden we twee strekkingen: de ene groep is pro relationele databases en contra NoSQL databases, de andere groep is dan weer pro NoSQL databases en contra relationele databases. Elke groep komt met zijn argumenten. In de volgende Paragrafen proberen we een overzicht te geven van de verschillende argumenten die de groepen gebruiken om hun database aan te prijzen. We zeggen niet dat dit allemaal gegronde redenen zijn. We trachten hier enkel een zo breed mogelijk beeld te vormen van de discussie die er gaande is.

4.4.1 Pro Relationele Databases

SQL en het Relationeel Model

We beginnen met de taal waarmee een traditionele database gequeried kan worden. SQL is al een dertigtal jaar de standaard wat betreft relationele databases. Het is een *declaratieve taal*, hiermee bedoelen we dat de programmeur aangeeft wat hij wil doen, maar zich niets moet aantrekken over hoe het systeem zijn aanvraag zal afhandelen. In SQL is het vrij eenvoudig om queries te stellen, we kunnen bijna letterlijk zeggen wat we willen doen: "Geef alle studenten die ouder zijn dan 25 jaar" of "geef alle studenten die Informatica volgen". Het voordeel hierbij is dat de programmeur zich nergens zorgen over hoeft te maken. Het RDBMS bepaalt zelf hoe de data wordt opgeslagen, welke indexen er gemaakt worden of welke algoritmes er gebruikt worden. Een belangrijke schakel hierbij is de *query optimizer*, deze kiest een zo goed mogelijk *query plan* om de query uit te voeren. Dit alles is echter enkel mogelijk gemaakt door alle theoretische onderbouwing waarover het relationeel model beschikt. Het weggooien van al deze kennis zou zonde zijn [38].

Als we dit alles samenvatten, kunnen we besluiten dat SQL en het relationeel model al een lange tijd meegaan. Hier is heel wat onderzoek aan vooraf gegaan en relationele databases werken nog steeds heel betrouwbaar. Waarom zouden we dit allemaal overboord gooien om over te schakelen naar iets nieuws?

Uitbreiding van Relationele Databases

Vele commerciële databases worden uitgebreid met aspecten van NoSQL. Om dit duidelijk te maken gebruiken we het voorbeeld van IBM's DB2.

Zoals we in Hoofdstuk 5 gaan zien, bestaan er verschillende soorten NoSQL databases. Een voorbeeld van dergelijke NoSQL databases, is de XML database. In deze database kunnen we XML documenten opslaan en ophalen. IBM heeft een extra laag over hun huidige database geschreven om ook XML documenten toe te kunnen voegen [58]. Voorstanders van het relationeel model doen vervolgens uitspraken als: "Waarom overstappen naar NoSQL als het ook kan met de bestaande relationele databases?".

Een ander voorbeeld is dat IBM's DB2 tegenwoordig ook kan werken met met JSON bestanden. Dit valt net zoals de XML databases onder de NoSQL document stores zoals we later zullen bespreken in Paragraaf 5.4. Zoals we zien past IBM zich aan, aan de noden die er zijn en proberen ze de gebruikers een zo groot mogelijk aanbod te geven van functionaliteiten [17].

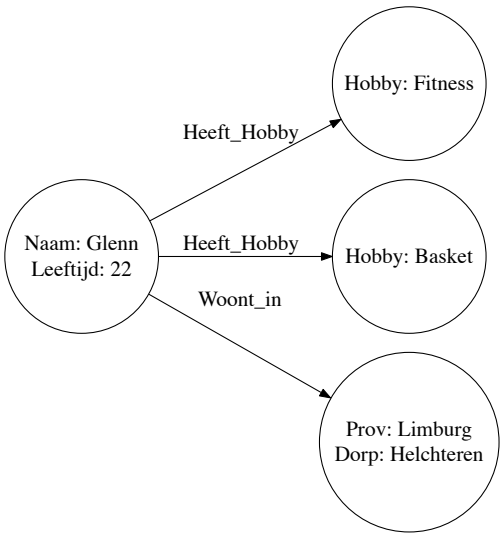
Andere commerciële databases doen hetzelfde als IBM en op die manier blijven ze up-to-date wat betreft de huidige technologie.

Samenvattend kunnen we stellen dat commerciële databases er alles aan doen om niet te moeten onder doen voor andere (NoSQL) databases en op deze manier verkleinen ze de kloof tussen relationele en NoSQL databases.

Voorstelling van de Data

NoSQL databases bieden een hele waaier aan mogelijkheden om hun data op te slaan, gaande van documenten tot graafstructuren. De realiteit is dat alle NoSQL vormen gemapt kunnen worden op het traditionele relationeel model. Merk op dat het niet altijd de mooiste oplossing is, maar het is wel mogelijk.

Waarom zouden we dan een andere structuur gebruiken [19]? We tonen dit aan met een klein fictief voorbeeld van een graafdatabase. We geven eerst de graafdatabase en transformeren deze vervolgens naar een relationele database. In Figuur 4.2 zien we een graafdatabase waar we zien dat Glenn twee hobby's heeft en in Helchteren woont.



Figuur 4.2: Een eenvoudig voorbeeld van een graafdatabase.

Het omzetten van deze graafdatabase naar een relationele database is vrij eenvoudig. We zien onmiddellijk drie tabellen, namelijk "Persoon", "Hobby" en "Woonplaats". In Tabel 4.3, Tabel 4.4 en Tabel 4.5 zien we een mogelijke voorstelling van een relationele database.

Persoon		
ID	**Naam**	**Leeftijd**
1	Glenn	22

Tabel 4.3: Relationele tabel voor "Persoon" uit de graafdatabase in Figuur 4.2

Hobby	
Pers-ID	**Hobby**
1	Fitness
1	Basket

Tabel 4.4: Relationele tabel voor "Hobby" uit de graafdatabase in Figuur 4.2

We merken op dat we een unieke ID hebben gegeven aan "Persoon", op deze manier kunnen we de informatie van de verschillende relaties joinen aan de hand

Woonplaats		
Pers-ID	**Prov**	**Dorp**
1	Limburg	Helchteren

Tabel 4.5: Relationele tabel voor "Woonplaats" uit de graafdatabase in Figuur 4.2

van deze key. De relatie "Heeft_Hobby" van de graafdatabase in Figuur 4.2 bekomen we door Tabel 4.3 en Tabel 4.5 te joinen met elkaar. De relatie "Woont_In" kan worden gesimuleerd door Tabel 4.3 en Tabel 4.4 met elkaar te joinen.

Als we dit samenvatten, kunnen we tot het besluit komen dat de relationele en de NoSQL databases dezelfde data kunnen modelleren. Overstappen naar een NoSQL database is bijgevolg niet nodig. Merk op dat we het hier enkel hebben over het modelleren, het verhaal over efficiëntie is hier niet in beschouwing genomen.

4.4.2 Pro NoSQL Databases

Hoeveelheid Data

NoSQL databases zijn bekend geworden door grote bedrijven. Deze bedrijven moeten zoveel data verwerken dat een gewone relationele database niet meer volstaat. Het punt wat we hier proberen duidelijk te maken is dat we het hier hebben over *enorm veel data*. Zo zijn er de voorbeelden van Google's BigTable, Facebook's Cassandra en Amazon's DynamoDB. Merk op dat de meeste bedrijven niet zoveel data bezitten en niet zo veel data moeten queryen als de gegeven voorbeelden [19].

Relationele databases zijn voor de *gewone* bedrijven met een normale hoeveelheid data sterk genoeg waardoor overstappen naar een NoSQL database niet nodig is. De echte NoSQL databases zijn immers in het leven geroepen om de problemen van bijvoorbeeld Google, Amazon en Facebook op te lossen. Samenvattend kunnen we stellen dat NoSQL wordt gebruikt door de grote bedrijven zoals Google, Amazon en Facebook omdat ze nu eenmaal te maken hebben met big data.

Schalen

Als we in een discussie verzeild geraken over relationele databases en NoSQL databases, dan is de kans heel groot dat we het woord schalen tegen komen. Zoals besproken in Paragraaf 1.6 zijn relationele databases het meest geschikt om verticaal te schalen en NoSQL databases om horizontaal te schalen. Het voordeel van horizontaal schalen is dat het sneller, goedkoper en vrij eenvoudig kan, waardoor wat betreft schaling een groot voordeel wordt gehaald uit NoSQL. Een mooi voorbeeld hiervan is het promotiefilmpje van DynamoDB [7] dat is voorgesteld in Figuur 4.3:

1. We beginnen met een situatie waarin gebruikers communiceren met een applicatie en die applicatie communiceert vervolgens met de database. Dit zien we in Figuur 4.3a.

2. Naarmate de applicatie populairder wordt, gebruiken er meer mensen deze applicatie en krijgt de database langzaam maar zeker meer werk. Deze situatie is voorgesteld in Figuur 4.3b.

3. Als we met een relationele database zouden werken, dan zouden we een betere database kunnen aanschaffen. Dit wordt jammer genoeg snel een dure zaak. Een andere mogelijkheid is om een cluster van databases aan te leggen. Dit zal werken, maar we blijven hier met het probleem zitten dat databases kunnen uitvallen, dat ze altijd de nieuwste software moeten draaien en dat het een ingewikkelde zaak is om deze te blijven managen. Het gebruik van een cluster is te zien in Figuur 4.3c.

4. Vanaf dit moment komt NoSQL in het verhaal. NoSQL staat voor zeer schaalbare systemen en DynamoDB stelt hun systeem voor als een machine waar we zelf een bepaalde werklast kunnen instellen. In het begin van onze applicatie is de werkdruk nog laag aangezien de applicatie nog niet bekend is. We stellen het aantal nodes met andere woorden laag in. Dit zien we in Figuur 4.3d.

5. Als de werkdruk zeer hoog wordt, moeten we niet gaan experimenteren met clusters van databases. We moeten enkel de database naar een hoger aantal nodes instellen en we moeten ons verder geen zorgen maken. Deze situatie zien we in Figuur 4.3e.

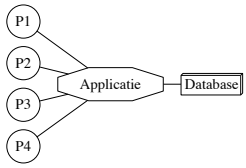
We merken nog op dat DynamoDB achterliggend nog heel wat andere functionaliteit verzorgt. Zo moeten de nodes worden geïnitialiseerd en zal de database ook een soort van recovery aanbieden zodat data niet verloren gaat bij een eventuele crash.

Verschillende Types

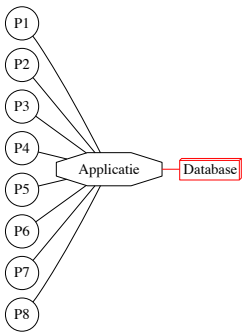
Een voordeel van de NoSQL stroming is het feit dat er zeer veel verschillende soorten NoSQL databases op de markt te verkrijgen zijn. In Hoofdstuk 5 bespreken we vijf bekende soorten NoSQL databases:

- key value store
- document store
- graph databases
- multivalue databases
- wide column store

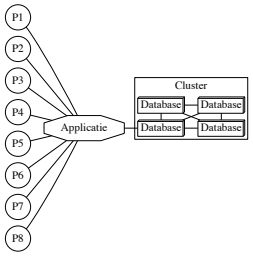
Als we beslissen om een NoSQL database te gebruiken, kunnen we eerst een grondige analyse uitvoeren en vervolgens beslissen welk soort we gebruiken. Dit geeft enorm veel vrijheid en op deze manier worden de noden van de gebruiker zo goed mogelijk ingevuld. Dit is in tegenstelling tot de relationele databases waar we altijd verplicht zijn om met dezelfde structuur te werken [23].



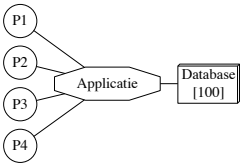
(a) Een standaard situatie: gebruikers communiceren met de applicatie en de applicatie communiceert met database.



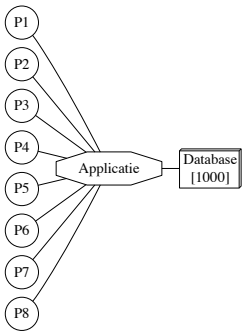
(b) Gebruikers nemen toe, waardoor de applicatie groeit en de database overbelast geraakt.



(c) Een mogelijk oplossing: gebruik een database cluster.



(d) Een NoSQL oplossing voor het probleem: stel de database op een laag niveau in.



(e) Een NoSQL oplossing voor het probleem: stel de database op een hoog niveau in.

Figuur 4.3: Schematische voorstelling van schalen door gebruik te maken van het promotiefilmpje van DynamoDB.

Data Model

Misschien wel één van de belangrijkste voordelen van NoSQL is het feit dat we veel losser kunnen omgaan met onze data. Herinner uit Paragraaf 1.3.1 dat we bij een relationele database een schema moeten definiëren waar onze data in alle tijden aan moet voldoen. Daarnaast moeten we ook keys en foreign keys definiëren. Naast de verplichtingen waar de data aan moet voldoen, zitten we ook met het probleem dat de data over verschillende tabellen is verdeeld. Als we gecombineerde data willen opvragen moeten we dus de verschillende tabellen joinen om op deze manier een gedetailleerder overzicht te krijgen.

NoSQL databases hebben geen nood aan een schema. We kunnen bijvoorbeeld eenvoudig een key samen met zijn value opslaan of anderzijds een volledig JSON document opslaan. Het voordeel van bijvoorbeeld een JSON document is het feit dat alle informatie reeds in dat document zit. Er moeten dus geen

ingewikkelde joins gebeuren om de volledige data op te vragen.

Door het feit dat we geen gebruik maken van schema's kunnen we op eender welk moment nieuwe data toevoegen zonder ook maar iets te moeten aanpassen. Bij een relationele database moet het schema worden aangepast en dit moet doorgevoerd worden op de bestaande data in de database [20].

4.5 Problemen met Specialisatie

In Paragraaf 4.4.1 hebben we het gehad over het feit dat relationele databases zich aanpassen aan NoSQL databases door zelf ook NoSQL functies aan te bieden. Dit is een mooi voorbeeld van een *algemene ontwikkeling* en een *gespecialiseerde ontwikkeling*. Relationele databases vallen onder de algemene ontwikkeling, ze blijven openstaan voor verschillende mogelijkheden, terwijl NoSQL zich focust op de gespecialiseerde ontwikkeling door zich maar op één bepaalde mogelijkheid te fixeren.

Kumar [33] heeft sterke bedenkingen bij het specialiseren van de databases. Het uitgangspunt van deze discussie is het feit dat gespecialiseerde systemen geoptimaliseerd zijn voor bepaalde situaties. Als er echter een kleine wijziging is in de situatie waarin het systeem moet gebruikt worden, zal de database veel minder efficiënt worden. Daarnaast hebben we de algemene systemen die zeker geen optimale prestaties garanderen in alle situaties, maar die wel degelijk presteren voor de meerderheid van de situaties.

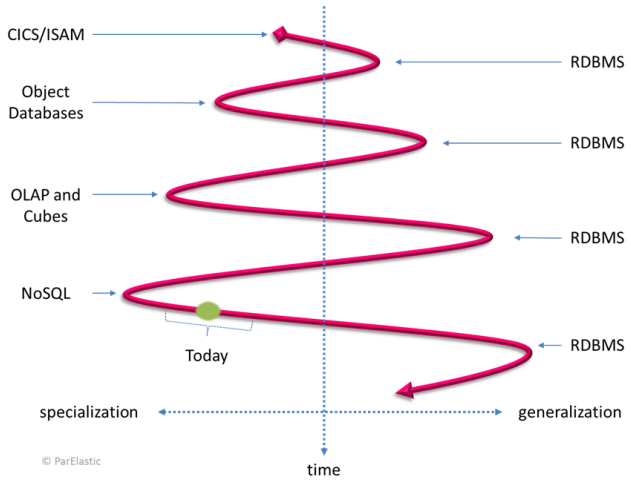
In het verleden zijn er momenten geweest waarin sommige situaties enkel opgelost konden worden met gespecialiseerde systemen. Hier bieden algemene systemen toch nog voordelen aangezien de technologie telkens sneller en beter wordt. Hierdoor kunnen de problemen die ooit enkel opgelost konden worden door een gespecialiseerd systeem, nu ook opgelost worden door een algemeen systeem. Een voorstelling hiervan is te vinden in Figuur 4.4. Hierin zien we dat we altijd naar een gespecialiseerd systeem grijpen, maar na verloop van tijd toch weer terugkeren naar het vertrouwde relationeel model. Als we naar Figuur 4.4 kijken zien we een mooi patroon en zien we ook dat we binnen X aantal jaar terug keren naar het relationeel model (als de geschiedenis zich zal herhalen).

Het grote probleem van specialisatie blijft het feit dat we het systeem aanpassen aan een bepaalde situatie. Als we op een bepaald moment het systeem queryen en de situatie verschilt van de oorspronkelijke situatie, dan kan het systeem al snel uit elkaar vallen. Hiermee bedoelen we dat als het systeem bijvoorbeeld te specifiek is en queries echt gebaseerd zijn op deze structuur dat het dan soms moeilijk is om queries op een efficiënte manier aan te passen als er een structuurwijziging plaatsvindt.

We kunnen concluderen dat algemene systemen over het algemeen beter presteren. Ze zijn namelijk eenvoudiger aan te passen aan de noden van de mensen en de evolutie van de data. Dit zien we al aan de commerciële databases die functies van beide werelden aanbieden.

4.6 Kritiek op NoSQL

Een grote tegenstander van NoSQL databases is Michael Stonebraker. In [48, 49] schrijft hij over het feit dat de NoSQL discussie in feite niets te maken heeft



Figuur 4.4: De vergelijking tussen algemene en gespecialiseerde systemen (Kumar [33]).

met SQL.

Het punt wat Stonebraker [49] hier dus wil duidelijk maken is dat alle overheads van traditionele databases, niets te maken hebben met SQL. Alles draait om de implementatie van de ACID transacties, multi-threading en het omgaan met de schijf. Om echt sneller te kunnen worden, moeten we de overheads oplossen. Dit kan met of zonder SQL context.

Stonebraker [48] geeft twee redenen waarom gebruikers mogelijk zouden kunnen overschakelen naar een NoSQL database. De twee redenen zijn:

- flexibiliteit
- prestaties

Merk op dat we deze redenen in Paragraaf 4.4.2 ook hebben gegeven om eventueel over te schakelen naar NoSQL databases. Het argument van flexibiliteit gaat over het feit dat gebruikers sommige data niet geschikt vinden om in een vast schema te gieten. Hier gaan we echter niet verder op in.

Stonebraker [48] bespreekt het prestatie argument als volgt: gebruikers die bijvoorbeeld MySQL gebruiken, kunnen na een tijdje niet meer tevreden zijn over de prestaties van hun database. Een oplossing hiervoor is om data te verdelen over verschillende databases, maar dat wordt voor de gebruiker al snel een moeilijke taak om te onderhouden. Daarnaast kan er ook voor de dure oplossing worden gekozen, namelijk een commerciële database aankopen. In [49] gaat Stonebraker dieper in op dit prestatie argument. Hij richt zich voor deze discussie op systemen waarvoor NoSQL databases het meest in aanmerking komen, namelijk “update en look-up” intensieve toepassingen. Er zijn twee manieren om prestaties van algemene databases te verhogen, namelijk enerzijds zorgen voor een automatische *sharding*. Met sharding bedoelen we het verdelen van de data over verschillende nodes. Anderzijds de snelheid van elke node afzonderlijk verhogen. De overhead die bij databases in het algemeen komen

kijken, hebben echter weinig te maken met SQL zelf. Daarom is de naam NoSQL volgens Stonebraker [49] slecht gekozen.

De standaard overhead van een applicatie met een databasesysteem is de communicatie tussen de applicatie en de DBMS. Het telkens over en weer communiceren van de applicatie met de DBMS (met bijvoorbeeld JDBC) kost heel veel tijd. Als we echte prestatiegerichte applicaties bekijken, zien we dat dergelijke applicaties altijd gebruik maken van procedures die ervoor zorgen dat de applicatielogica in de DBMS wordt opgenomen. Een andere mogelijke oplossing is de DBMS in dezelfde adresruimte te runnen als de applicatie, maar dit is wegens veiligheidsredenen niet aan te raden.

Daarbij komt nog eens de algemene overhead van de standaard operaties die een DBMS gebruikt, namelijk:

- logging: traditionele databases schrijven alles twee keer weg. Ze schrijven namelijk de data en de log weg naar de schijf.
- locking: alvorens iets kan worden weggeschreven, moet er eerst een lock worden aangevraagd.
- latching: we moeten opletten met gedeelde datastructuren. Ook hiervoor moet een soort van locking worden gebruikt, namelijk latching.
- buffer management: data wordt in vaste disk pages opgeslagen. Hierbij moet worden bepaald welke disk pages in het geheugen worden geladen.

Het oplossen van één van de bovenstaande overheads kan zorgen voor een verbetering in efficiëntie van maar liefst 25%. We kunnen verwachten dat het oplossen van al deze oorzaken van overhead een enorme prestatieverbetering zal opleveren [48].

4.7 Conclusie

In dit Hoofdstuk hebben we het gehad over NoSQL databases. Eerst hebben we gekeken wat de term NoSQL inhoudt.

Vervolgens zijn we dieper ingegaan op het ontstaan van NoSQL databases. Hier zijn we tot het besluit gekomen dat de mogelijke oorzaken te maken hebben met big data. Relationele databases zijn namelijk niet zo geschikt om te kunnen werken met enorm veel data. Daarnaast kunnen we ook oorzaken vinden bij de structuur van de data en de drang om telkens meer te parallelliseren.

Ook hebben we een aantal argumenten tegenover elkaar gezet over pro NoSQL en pro relationele databases. De strekking van de pro relationele databases argumenteert dat SQL al heel lang bestaat. Ze geeft ook aan dat alles in principe kan worden omgezet naar het relationeel model. Hier komt nog bij dat verschillende commerciële bedrijven features van NoSQL databases verwerken in hun eigen relationele databases.

Het belangrijkste argument pro NoSQL is het feit dat ze zeer schaalbaar zijn en relatief eenvoudig om kunnen gaan met big data.

Hoofdstuk 5

Soorten NoSQL

5.1 Inleiding

In dit Hoofdstuk bespreken we enkele vormen van NoSQL databases. Het is hier niet de bedoeling om tot in het diepste detail deze databases te bespreken. De bedoeling is om een overzicht te geven waardoor we ons een beter beeld kunnen vormen van hoe deze databases werken en welke technieken er worden toegepast.

We bespreken alle databases aan de hand van hetzelfde voorbeeld. Dit voorbeeld wordt eerst als het relationeel model voorgesteld en vervolgens passen we dit voorbeeld toe op de volgende NoSQL databases:

- key value store
- document store
- graph databases
- multivalue databases
- wide column stores

5.2 Voorbeeld in het Relationeel Model

In het vervolg van dit Hoofdstuk geven we voor elke NoSQL database een voorbeeld. Het is duidelijker als we daarvoor een algemeen voorbeeld gebruiken dat we telkens op de individuele databases toepassen. We gebruiken een database die informatie bijhoudt over studenten, hun loopbaan en hun opleidingsonderdelen. We merken op dat dit een louter illustratief voorbeeld is dat de verschillende punten van de NoSQL databases zo goed mogelijk probeert te belichten. Het kan daarom voorkomen dat we bewust sommige attributen leeg laten. De verschillende relaties bevatten de volgende attributen[1]:

- Student (Stud.): stelt een student voor.

[1] In de databases korten we de relatienamen af om de voorstellingen zo compact mogelijk te houden.

  - StudentenID (ID): een unieke waarde voor dit record.
  - Achternaam (ANaam): de achternaam van de student.
  - Voornaam (VNaam): de voornaam van de student.
  - Gemeente (Gem.): de gemeente waar de student woont.

- Loopbaan (Lpbn): stelt de volledige geschiedenis van een student voor.
  - StudentenID (ID): refereert naar een student.
  - Jaar: het academiejaar.
  - Inschrijving (Inschr.): voor welke opleiding de student in dat jaar ingeschreven is.
  - Gemiddelde (Gemid.): het gemiddelde dat de student behaald heeft.
- Opleidingsonderdeel (Opl.): stelt alle vakken voor die een student in een bepaald jaar volgt.
  - StudentenID (ID): refereert naar een student.
  - Jaar: het academiejaar.
  - Vak: het vak dat bij een opleiding hoort.
  - Studiepunten (Stdp.): hoeveel studiepunten een vak bevat.

De keys van elke relatie zijn onderstreept. Als we deze drie tabellen met andere woorden met elkaar joinen, krijgen we alle informatie die beschikbaar is over een student. In Tabel 5.1, Tabel 5.2 en Tabel 5.3 zien we de bovengenoemde relaties, opgevuld met voorbeelddata.

Student			
ID	**VNaam**	**ANaam**	**Gem.**
0928574	Gert	Peeters	Helchteren
1227833	Sarah	Maes	NULL
1335879	Peter	NULL	NULL

Tabel 5.1: Voorbeeld: relationele tabel voor “Student”.

Loopbaan			
ID	**Jaar**	**Inschr.**	**Gemid.**
0928574	2009-2010	Informatica	72
0928574	2010-2011	Informatica	67
0928574	2011-2012	Informatica	75
0928574	2012-2013	Informatica	65
0928574	2013-2014	Informatica	NULL
1227833	2012-2013	TEW	62
1227833	2013-2014	NULL	NULL
1335879	2013-2014	NULL	NULL

Tabel 5.2: Voorbeeld: relationele tabel voor “Loopbaan”.

We merken nogmaals op dat bijvoorbeeld het NULL maken van een achternaam niet heel realistisch is, maar dit zorgt voor genoeg vrijheid om verschillende aspecten van NoSQL databases te belichten.

Opleidingsonderdeel			
ID	**Jaar**	**Vak**	**Stdp.**
0928574	2009-2010	NULL	NULL
0928574	2010-2011	NULL	NULL
0928574	2011-2012	NULL	NULL
0928574	2012-2013	NULL	NULL
0928574	2013-2014	DBSA	6
0928574	2013-2014	InfVis	6
0928574	2013-2014	MPC	6
0928574	2013-2014	Complexiteit	6
0928574	2013-2014	Project DB	6
0928574	2013-2014	Masterproef	30
1227833	2012-2013	Micro Eco	12
1227833	2012-2013	Macro Eco	12
1227833	2012-2013	Boekhouden	12
1227833	2012-2013	Wiskunde	12
1227833	2012-2013	Marketing	12
1227833	2013-2014	NULL	NULL
1335879	2013-2014	NULL	NULL

Tabel 5.3: Voorbeeld: relationele tabel voor "Opleidingsonderdeel".

5.3 Key Value Store

De *key value* stores zijn misschien wel de eenvoudigste soort onder de NoSQL databases. Deze database slaat zijn data simpelweg op door kolommen te maken die enkel bestaan uit een "key" en een "value". Een key wordt geassocieerd aan een unieke value. Dit soort NoSQL database is enerzijds zo eenvoudig door enkel gebruik te maken van keys en values, maar anderzijds worden de operaties die we kunnen uitvoeren op de database ook heel eenvoudig gehouden. De basisoperaties van key-value stores zijn de volgende [30]:

- *put(key, value)*: zet *key* samen met *value* in de database.
- *get(key)*: geeft de waarde van *key* terug.
- *delete(key)*: verwijdert de key en de value van *key*.

Voor meer informatie hierover verwijzen we naar Hoofdstuk 6. In dit Hoofdstuk bespreken we de NoSQL database Voldemort, een key-value store die gebaseerd is op DynamoDB van Amazon.

Ten slotte passen we het voorbeeld uit Paragraaf 5.2 toe op de key-value store. Een mogelijk resultaat van een student is te zien in Tabel 5.4.

We zien dat de keys op een triviale manier worden aangemaakt, namelijk het studentenID samen met de attribuutnaam. Dit is *een* mogelijke hashfunctie. We kunnen nog honderden andere hashfuncties bedenken om de data te mappen.

Merk op dat in dit voorbeeld de value altijd slechts bestaat uit één waarde. We kunnen in principe zo veel waardes in een key stoppen als we willen.

Key	Value
0928574_Naam	Gert
0928574_Geslacht	man
0928574_Richting	TEW
0928574_Jaar	3
0928574_Studiepunten	60
1227833_Naam	Sarah
1227833_Geslacht	vrouw
1227833_Richting	fysica
1227833_Jaar	5
1335879_Naam	Peter

Tabel 5.4: Key-value store.

5.4 Document Store

Document stores bouwen verder op het principe van key-value stores. In dit geval bestaat de *key* zoals in alle andere gevallen uit een *unieke waarde* die verwijst naar een value. Deze *value* bestaat uit een *(semi-)gestructureerd document* zoals bijvoorbeeld JSON of XML. Deze documenten kunnen zeer complex zijn en kunnen bijvoorbeeld informatie van verschillende joins bevatten: bij een relationele database zouden we eerst deze tabellen moeten joinen om vervolgens informatie te kunnen queryen. In een document store kan het volledige document in één keer worden ingelezen door simpelweg gebruik te maken van de key [21]. Verschillende documenten die samen horen worden opgeslagen in *collections*. In termen van het relationeel model kunnen we dit als volgt beschrijven: een tupel kan vergeleken worden met een document en een tabel kan vergeleken worden met een collection [40].

Het feit dat we niet te maken hebben met een vast schema heeft zowel voor- als nadelen. Het voordeel is dat we eender welke data kunnen opslaan. We moeten ons (zoals eerder vermeld) niet aan een schema houden. Het nadeel hiervan is dat programmeurs voorzichtig moeten omgaan met dit feit. Met andere woorden moeten we onze applicaties op zo een manier ontwikkelen dat we geen foute informatie kunnen toevoegen aan de database. In een relationele database zou dit immers worden opgevangen door het schema [18].

We geven een voorbeeld om dit soort database voor te stellen. We baseren ons opnieuw op het voorbeeld zoals weergegeven in Paragraaf 5.2. In Listing 5.1 zien we een voorstelling van “Student” in een semi-gestructureerd formaat.

```
0928574
{
      VNaam: Gert,
      ANaam: Peeters,
      Gem.: Helchteren
}

1227833
{
      VNaam: Sarah,
```

```
        ANaam: Maes
}

1335879
{
        VNaam: Peter
}
```

Listing 5.1: Voorstelling van "Student" in een semi-structered formaat.

In Listing 5.2 zien we informatie in verband met de "Loopbaan" van de student.

```
Lpbn-0928574-0910
{
        Inschr: Informatica,
        Gemid.: 72
}

Lpbn-0928574-1011
{
        Inschr: Informatica,
        Gemid.: 67
}

Lpbn-0928574-1112
{
        Inschr: Informatica,
        Gemid.: 75
}

Lpbn-0928574-1213
{
        Inschr: Informatica,
        Gemid.: 65
}

Lpbn-0928574-1314
{
        Inschr: Informatica
}
```

Listing 5.2: Voorstelling van "Loopbaan" in een semi-structered formaat.

We merken nog op dat de geneste bomen ook een eigen unieke key hebben gekregen.

Door de flexibiliteit van de documenten, is het ook mogelijk om deze *embedded* op te slaan [40]. Dit wil zeggen dat we "Student" en "Loopbaan" bij

elkaar plaatsen. Een voorbeeld hiervan is te vinden in Listing 5.3. We hebben dit slechts uitgewerkt voor één student en enkel voor de tabellen "Student" en "Loopbaan". De rest van de uitwerking verloopt triviaal en zou duidelijk moeten zijn uit het voorbeeld. Merk op dat elk subdocument ook nog een aparte key heeft.

```
0928574
{
      VNaam: Gert,
      ANaam: Peeters,
      Gem.: Helchteren
      Lpbn.:
      {
            Lpbn-0928574-0910
            {
                  Inschr: Informatica,
                  Gemid.: 72
            }

            Lpbn-0928574-1011
            {
                  Inschr: Informatica,
                  Gemid.: 67
            }

            Lpbn-0928574-1112
            {
                  Inschr: Informatica,
                  Gemid.: 75
            }

            Lpbn-0928574-1213
            {
                  Inschr: Informatica,
                  Gemid.: 65
            }

            Lpbn-0928574-1314
            {
                  Inschr: Informatica,
            }
      }
}
```

Listing 5.3: Voorstelling van embedded data.

5.5 Graph Databases

De *graph databases* gebruiken een graafstructuur om data voor te stellen. Ze maken gebruik van *vertices*, *bogen* en *eigenschappen* van vertices. De vertices stellen een record voor uit een relationele tabel, de bogen stellen de relaties ("relationships") voor tussen de tabellen uit een relationele database en de eigenschappen zijn de waardes van een record. We merken op dat de bogen niet alleen relaties kunnen voorstellen, maar ook eigenschappen. We geven opnieuw een voorbeeld om dit duidelijk te maken. Om de graaf overzichtelijk te houden geven we enkel de key als eigenschap van een vertex. Om een volledig beeld te scheppen, moeten we dan enkel alle andere waardes van dat record erbij denken. In Figuur 5.1 zien we een voorstelling van het "Student"-"Loopbaan" voorbeeld uit Paragraaf 5.2.

In dit voorbeeld hebben we alleen de loopbanen van de student met ID 0928574 weergegeven. Als we de volledige graaf zouden tekenen, zouden we het overzicht verliezen. In uitgebreidere voorbeelden kunnen alle vertices verbonden zijn met nog honderden andere vertices waardoor we een zeer complex netwerk kunnen krijgen [50].

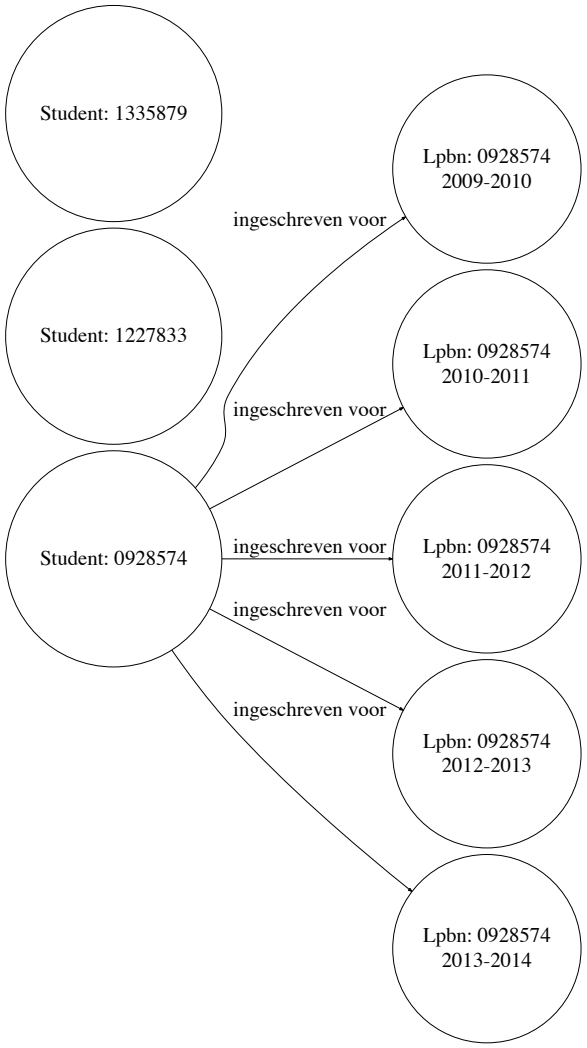
5.6 Multivalue Databases

Multivalue databases lijken heel sterk op de traditionele relationele databases. We weten dat relationele databases de data opslaan in twee dimensionale tabellen. Multivalue databases kunnen in dat opzicht iets meer aangezien ze de data kunnen opslaan in een driedimensionale tabel. Met andere woorden kunnen we meer dan één waarde per attribuut opslaan [47]. We kunnen dit goed demonstreren aan de hand van ons voorbeeld uit Paragraaf 5.2. We passen deze techniek toe op de relatie "Opleidingsonderdeel". Een uitgewerkte multivalue tabel zien we in Tabel 5.5.

ID	**Jaar**	**Vak**	*Stdp.*
0928574	2009-2010	NULL	NULL
0928574	2010-2011	NULL	NULL
0928574	2011-2012	NULL	NULL
0928574	2012-2013	NULL	NULL
0928574	2013-2014	DBSA, InfVis, MPC, Complexiteit, Project DB, Masterproef	6, 6, 6, 6, 6, 30
1227833	2012-2013	Micro Eco, Macro Eco, Boekhouden, Wiskunde, Marketing	12, 12, 12, 12, 12
1227833	2013-2014	NULL	NULL
1335879	2013-2014	NULL	NULL

Tabel 5.5: Een voorbeeld van een multivalue tabel.

Door gebruik te maken van multivalue krijgen we een logischer en compacter geheel. Merk op dat we ervoor moeten zorgen dat de volgorde tussen de vakken en de studiepunten gerespecteerd moet worden tijdens het toevoegen in



Figuur 5.1: Voorstelling van het "Student" -"Loopbaan" voorbeeld in een graafvorm.

de database. Het grote voordeel is dat we bijvoorbeeld alle vakken in één cel kunnen stoppen [31]. Bij het voorbeeld van het relationeel model, moesten we voor elk vak een rij definiëren.

5.7 Wide Column Store

Wide column stores kunnen we classificeren onder NoSQL databases. We mogen dit soort database niet verwarren met de reeds bekende column-oriented databases. Een column-oriented database is een relationeel implementatie techniek van het relationeel model, terwijl een wide column store een echte NoSQL database is. Een wide column store slaat zijn data op in cellen die gegroepeerd

worden in kolommen. Dus niet in rijen zoals het traditioneel relationeel model. Deze kolommen worden vervolgens logisch gegroepeerd in zogenaamde *kolom families*. Deze kolom families kunnen in principe een onbeperkt aantal kolommen bevatten.

Zoals eerder aangehaald slaan relationele databases hun data op in rijen. Het voordeel van data op te slaan in kolommen is het feit dat we op deze manier snelle opzoekingen kunnen doen en bijgevolg ook snel de data kunnen opvragen. Daarnaast biedt het ook voordelen wat betreft data aggregatie. Relationele databases slaan een rij als aangesloten blokken geheugen op, op bijvoorbeeld de harde schijf. Het kan dus voorkomen dat verschillende rijen op verschillende plaatsen op de harde schijf voorkomen. De column-oriented NoSQL databases slaan daarentegen alle cellen van een kolom op in aangesloten blokken op de harde schijf. Het ophalen van deze data kan bijgevolg veel efficiënter gebeuren [34].

In Tabel 5.6 hebben we een mogelijke omzetting gedaan van een relationeel model naar een wide column store. We hebben dit gedaan voor de student met studentenID 0928574 en voor het jaar 2013-2014. De rest van de omzetting kan via een analoge manier gebeuren. Merk op dat dit geen voorstelling is van hoe de data effectief op de harde schijf wordt opgeslagen. Dit geeft enkel een voorstelling hoe de data met elkaar in relatie staat.

We zien in Tabel 5.6 twee rij keys, namelijk “StudentID” en “Jaar”. Deze keys helpen de kolommen te identificeren. Naast de keys beschikken we ook over drie kolom families, namelijk “Student”, “Loopbaan” en “Opleidingsonderdeel”. Deze kolom families bevatten op hun beurt de kolommen:

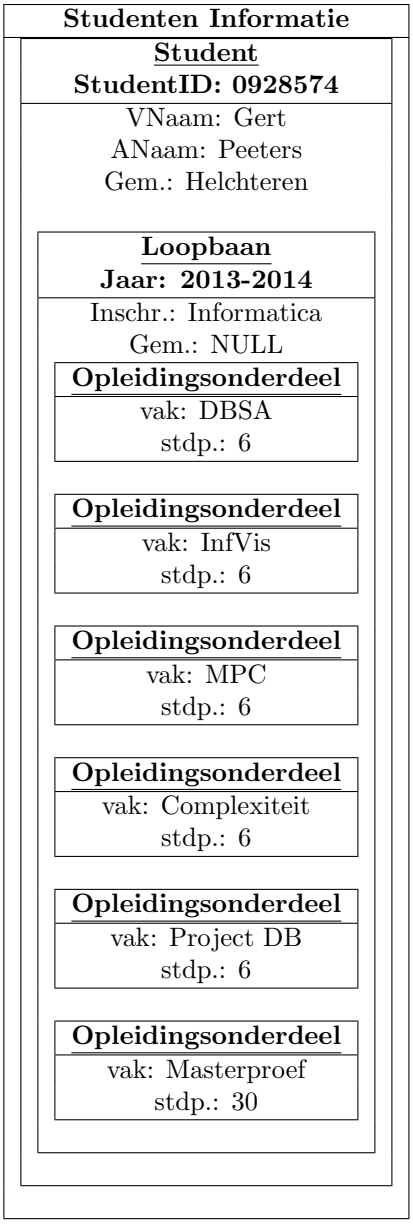
- Student: Vnaam, Anaam, Gem.
- Loopbaan: Inschr., Gem.
- Opleidingsonderdeel: Vak, Stdp.

Merk op dat dit een verdeling is. We kunnen in principe de data ook anders ordenen. Zo kunnen we bijvoorbeeld de opleidingsonderdelen enkel bij de student bijhouden, in plaats van ook bij de loopbaan.

5.8 Conclusie

In dit Hoofdstuk hebben we verschillende soorten NoSQL databases besproken. We hebben telkens een voorbeeld gegeven van een omzetting van het relationele model naar een soort van NoSQL database. Uit dit Hoofdstuk onthouden we best dat er verschillende soorten NoSQL databases zijn en dat deze allemaal andere eigenschappen hebben. Het al dan niet kiezen voor een NoSQL database hangt sterk af van de toepassing die we willen ontwikkelen.

Een geschikte NoSQL database vinden, zal echter geen probleem zijn aangezien er tegenwoordig zeer veel NoSQL databases ter beschikking zijn. We moeten dus enkel de database kiezen die het beste aan onze noden beantwoordt.



Tabel 5.6: Wide column store voor een student.

Hoofdstuk 6

Voldemort

6.1 Inleiding

In de vorige Hoofdstukken hebben we de theorie bekeken die wordt gebruikt bij het ontwikkelen van onder andere NoSQL databases. In het vervolg van deze thesis bekijken we enkele bestaande NoSQL databases. We bespreken welke technieken ze gebruiken en wat de verschillende eigenschappen zijn. Verder doen we ook enkele tests om te kijken hoe consistent en hoe beschikbaar deze databases zijn.

In dit Hoofdstuk bespreken we de NoSQL database Voldemort. Dit is een bestaande key-value store.

Om te beginnen maken we een afspraak in verband met terminologie:

- Netwerkpartitie: een communicatieprobleem tussen twee nodes.
- Data distributie: het verdelen van data over verschillende nodes.

6.2 Wat is Voldemort?

Voldemort is een open-source NoSQL database. Voldemort past in de categorie van de key-value stores, zoals we gezien hebben in Paragraaf 5.3. Deze database ondersteunt dus enkel de volgende operaties:

- put (key, value)
- get (key)
- delete (key)

Voldemort is in feite afgeleid van DynamoDB van Amazon. De achterliggende werking van beide databases is min of meer hetzelfde.

Voldemort wordt in de praktijk gebruikt door LinkedIn om grote schaalbaarheid te verwezenlijken [56].

6.3 Model

In deze Paragraaf bespreken we de achterliggende werking en technieken die worden gebruikt bij Voldemort. We hebben vooral aandacht voor de manier waarop Voldemort zijn data verdeelt over de verschillende nodes.

6.3.1 Naïeve Data Distributie

Het is belangrijk dat we data niet op één plaats opslaan, maar dat we de data verdelen en kopiëren naar verschillende nodes. Op die manier kunnen we een hogere beschikbaarheid bereiken, maar zijn we ook beschermd tegen het eventueel uitvallen van een node. Als we data distribueren over verschillende nodes, moeten we wel rekening houden dat bijvoorbeeld maar één node de leesoperaties kan afhandelen [54]. We geven een klein voorbeeld om dit te verduidelijken in Figuur 6.1. Stel dat we een array van acht getallen hebben.

$$\{2, 4, 6, 8, 10, 12, 14, 16\}$$

We kunnen deze array opsplitsen in bijvoorbeeld vier delen, namelijk $\{2, 4\}$, $\{6, 8\}$, $\{10, 12\}$ en $\{14, 16\}$. In ons voorbeeld beschikken we over een cluster die bestaat uit vier nodes. We verdelen de vier stukken gelijkmatig over de vier nodes. Dit zien we in Figuur 6.1a. Als een client nu een leesoperatie op node 1 uitvoert om het vijfde element van de array te weten, dan zal deze leesoperatie verder moeten worden gepropageerd naar node 3, die $\{10, 12\}$ bevat. In dit voorbeeld reist de leesoperatie van node 1 naar node 2 en van node 2 verder naar node 3. We zien dit in Figuur 6.1b en Figuur 6.1c. Merk op dat we ook een andere weg hadden kunnen kiezen. Uiteindelijk wordt in Figuur 6.1d de juiste waarde 10 teruggegeven. Zonder gebruik van een soort van index systeem is het hier moeilijk om terug te vinden waar 10 effectief is opgeslagen. We moeten daarom in feite alle nodes controleren tot we 10 tegenkomen. Dit is niet zo efficiënt.

In Paragraaf 6.3.2 bespreken we consistent hashen dat wordt gebruikt door Voldemort. Om het nut van consistent hashen te motiveren, bespreken we echter eerst een andere mogelijke oplossing. We introduceren hiervoor eerst enkele variabelen:

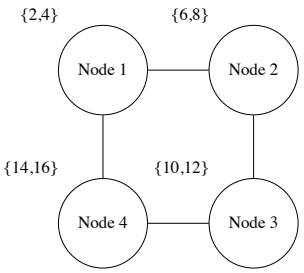
- D: het aantal data distributies
- K: een key van de data
- S: een server (node)

We doen het volgende: we splitsen de data op in D data distributies, waar D gelijk is aan het totaal aantal beschikbare nodes. We berekenen welke node verantwoordelijk is voor een bepaalde key K als volgt [54]:

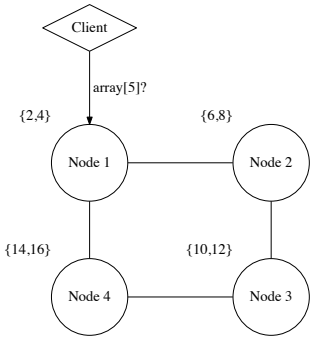
$$S_i = K \ mod \ D$$

Om dit duidelijk te maken geven we een voorbeeld. In Tabel 6.1 zien we een mogelijke inhoud van de Voldemort database.

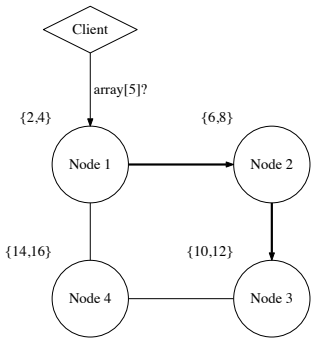
We zien telkens een key met zijn bijgevoegde value. We hebben daarnaast een systeem dat beschikt over vier nodes, dus $D = 4$. We moeten dus proberen om deze data zo efficiënt mogelijk te verdelen. We gebruiken hiervoor de



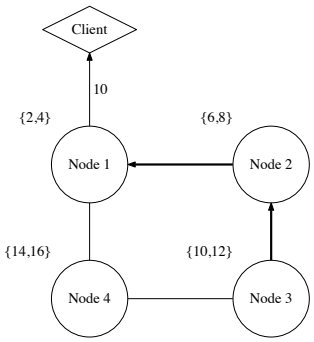
(a) Een systeem met vier nodes waarin een array is verdeeld in vier delen ($\{2,4\}$, $\{6,8\}$, $\{10,12\}$, $\{14,16\}$).



(b) Node 1 krijgt een leesoperatie voor het vijfde element uit de array.



(c) $array[5]$ wordt opgehaald.



(d) 10 wordt teruggegeven.

Figuur 6.1: Schematische voorstelling om het verdelen van data duidelijk te maken.

Key	**Value**
1	a
2	b
3	c
4	d
5	e
6	f
7	g
8	h
9	i
10	j

Tabel 6.1: Voorbeeld van de inhoud van de Voldemort Database.

bovenstaande formule $S_i = K \ mod \ D$. We beginnen met de eerste key 1. Als we deze in de formule invullen, krijgen we: $1 \ mod \ 4 = 1$. Met andere woorden

wordt de key met waarde 1, samen met zijn value a opgeslagen in de node met id 1. We geven nog één voorbeeld voor de key met waarde 4: $4 \; mod \; 4 = 0$. De key met waarde 4 en value d moet dus worden opgeslagen in de node met id 0. De andere berekeningen laten we over aan de lezer. In Tabel 6.2 geven we de volledige verdeling.

Node	**Key-Value**
0	(4,d), (8,h)
1	(1,a), (5,e), (9,i)
2	(2,b), (6,f), (10,j)
3	(3,c), (7,g)

Tabel 6.2: Het voorbeeld uit Tabel 6.1, verdeeld over de verschillende nodes.

Een voordeel van deze techniek is dat iedereen de exacte locatie van de keys kan berekenen door enkel D en K te weten. Verder is de berekening vrij eenvoudig.

We stoten echter snel op een probleem als we bedenken wat er moet gebeuren als er een node weg wordt gehaald omdat deze bijvoorbeeld stuk is of als we een node toevoegen om een groter systeem te krijgen. Het gevolg hiervan is dat key-values in feite op de verkeerde plaats staan en dat deze allemaal opnieuw moeten berekend worden. Stel dat we in ons eerder gegeven voorbeeld plotseling overschakelen op vijf nodes. Als we bijvoorbeeld de node voor key 5 berekenen, krijgen we: 5 mod 5 = 0, dus we moeten de key met waarde 5 verplaatsen van de node met id 1 naar de node met id 0 [54].

Voldemort voorkomt dit probleem door gebruik te maken van consistent hashen.

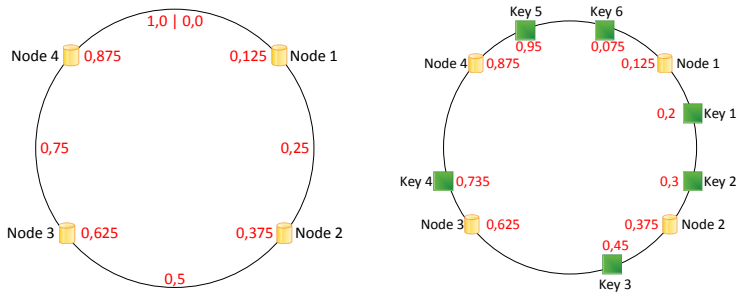
6.3.2 Consistent Hashen

Consistent hashen zorgt ervoor dat we opnieuw een plaats kunnen berekenen voor de keys, met als voordeel dat we bij het toevoegen of wegnemen van een node veel minder keys moeten verplaatsen. In feite moet enkel de data die in de nieuwe node hoort, verplaatst worden. De rest van de data kan dus gewoon op zijn plaats blijven [54].

Hoe werkt Consistent Hashen?

We stellen ons systeem voor als een ring die een interval $[0, 1]$ voorstelt. We kunnen zo een voorstelling zien in Figuur 6.2. Merk op dat we de ring doorlopen met de klok mee.

Het idee gaat als volgt: we hashen elke node naar een waarde in het interval $[0, 1]$. De nodes worden dus verdeeld over de ring, zoals we zien in Figuur 6.2a. Merk op dat de rode cijfers aan de binnenkant van de cirkel de hashwaardes voorstellen. Vervolgens hashen we de keys uit de Voldemort database ook naar een waarde van het interval $[0, 1]$. Dit zien we in Figuur 6.2b. Dit is alles wat we moeten doen, want op deze manier hebben we reeds beslist welke node verantwoordelijk is voor welke key. De verdeling van de keys gaat als volgt: de key hoort bij de node die we het eerste tegenkomen als we de ring in wijzerzin aflopen. Zo zien we bijvoorbeeld dat key 1 en key 2 bij node 2 horen [43].



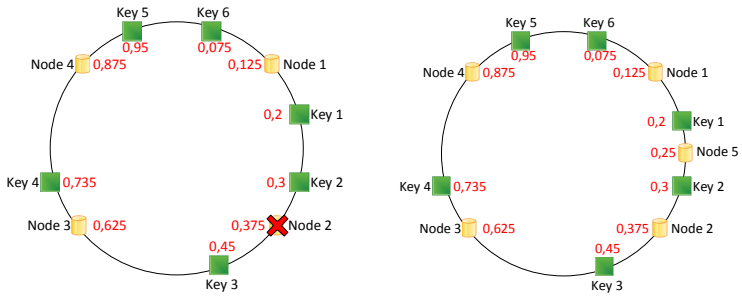
(a) Een verdeling van de nodes over de hash ring.

(b) De hash ring met de nodes en de data.

Figuur 6.2: Schematische voorstelling van consistent hashen.

Op deze manier is het ook eenvoudig in te zien wat er gebeurt wanneer er een node wegvalt of wordt toegevoegd. In Figuur 6.3a zien we dat node 2 wegvalt. Key 1 en key 2 die normaal bij node 2 hoorden, worden nu doorgehashed naar node 3.

In het andere geval, wanneer er een node wordt toegevoegd, gebeurt er iets gelijkaardigs. We zien dat tussen node 1 en node 2 een nieuwe node 5 wordt toegevoegd. Als we in Figuur 6.3b kijken zien we hoe key 1 nu naar node 5 wordt gehashed in plaats van naar node 2. Key 1 wordt dus verwijderd bij node 2 en wordt toegevoegd bij node 5.



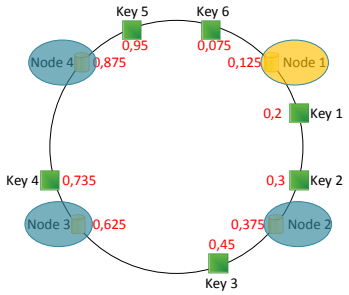
(a) Node 2 stopt met werken.

(b) We voegen node 5 toe.

Figuur 6.3: Schematische voorstelling van consistent hashen (verwijderen en toevoegen van een node).

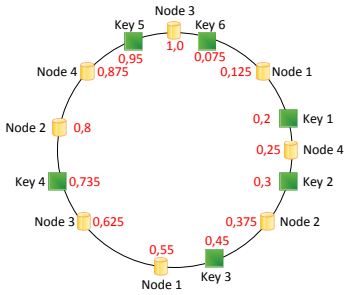
Een belangrijk punt dat we hier moeten vermelden, is het feit dat als een key bij een node wordt geplaatst, dat deze node dan de *master node* is voor deze specifieke key. In DynamoDB van Amazon wordt dit de *coordinator node* genoemd. Om de functie van deze node te kunnen uitleggen, introduceren we de variabele N. Deze variabele zegt hoeveel keer de data moet worden gekopieerd over de verschillende nodes. Voor een uitgebreidere uitleg verwijzen we naar Paragraaf 6.4.1. De master node is zoals eerder vermeld verantwoordelijk voor een specifieke key. De taak van deze node is om zijn keys te kopiëren naar zijn $N-1$ volgende nodes in de hash ring. Als N gelijk is aan het totaal aantal

nodes, moet de master node dus naar alle andere nodes een kopie sturen. We kunnen dit illustreren aan de hand van onze vorige voorbeelden. In Figuur 6.4 zien we dat node 1 bij $N = 4$ verantwoordelijk is voor de blauwe nodes.



Figuur 6.4: Bij $N = 4$ is node 1 verantwoordelijk voor de blauwe nodes.

Een probleem van deze hash ring is dat de verdeling van de keys over de verschillende nodes niet altijd even mooi verdeeld is. We kunnen dit oplossen door replica's van nodes (*virtuele nodes*) toe te voegen in de hash ring. Virtuele nodes zorgen ervoor dat fysieke nodes op verschillende plaatsen in de hash ring bereikt kunnen worden. Merk op dat als in dit geval een node uitvalt, ook al zijn virtuele nodes uit de hash ring verdwijnen. De virtuele nodes bieden enkele belangrijke voordelen ten opzichte van de methode zonder virtuele nodes. Het voordeel hierbij is dat bij het uitvallen van een node, de keys die bij deze nodes hoorden nu gelijkmatiger worden verdeeld over de rest van de nodes. Dit geldt ook in de omgekeerde situatie als er een nieuwe node in het netwerk komt. Op deze manier worden op een gelijkmatige manier keys van andere nodes overgenomen door deze nieuwe node. Voor de master node verandert de situatie een klein beetje. Hij moet namelijk de data repliceren naar de $N - 1$ unieke volgende nodes in de hash ring [22].



Figuur 6.5: Gebruik van replica's in een hash ring.

6.3.3 Consistentie

Het is moeilijk om in een gedistribueerd systeem consistentie te bewaren. Dit heeft onder andere te maken met de CAP stelling zoals gezien in Hoofdstuk 2. Als we kiezen voor beschikbaarheid en partitie tolerantie, dan is het moeilijk om

de consistentie onder controle te houden. In Paragraaf 2.5 hebben we gekeken hoe we moeten omgaan met netwerkpartities en wat de mogelijke oplossingen kunnen zijn om problemen in verband met consistentie op te lossen. We hebben onder andere gezien dat we inconsistenties kunnen vermijden gedurende een netwerkpartitie of dat we consistentie proberen te herstellen na een netwerkpartitie. Voldemort verkiest de mogelijkheid om inconsistenties in het systeem toe te laten, maar om deze op te lossen vanaf het moment dat er een leesoperatie gebeurt. We herinneren uit Paragraaf 3.2 dat we een systeem als consistent beschouwen vanaf het moment dat elke node dezelfde data ziet. Als we te maken hebben met een systeem dat enkel leesoperaties toelaat, is er geen probleem. De data wordt immers eenmaal aangemaakt en vanaf dat moment enkel nog gelezen. De problemen ontstaan echter als we beginnen te schrijven en data beginnen te kopiëren. Om dit allemaal in goede banen te leiden, gebruikt Voldemort versies (Engels: versioning) en leesherstelling (Engels: read-repair) [54].

Versies

Om problemen met conflicterende data op te kunnen lossen, gebruikt Voldemort versies door gebruik te maken van *vectorklokken*. We geven eerst een algemene uitleg en passen deze vervolgens toe op Voldemort.

Een vectorklok geeft aan elke operatie een *timestamp*. Op deze manier kunnen we vectorklokken onderling vergelijken om causale afhankelijkheden tussen operaties te achterhalen. Merk op dat dit te maken heeft met de causale consistentie uit Paragraaf 3.4. Elk proces krijgt dus een vectorklok toegewezen die de geschiedenis kan reconstrueren. We zeggen dat een proces P_i een vectorklok $VC_i[0 \dots n]$ heeft met een initiële waarde van $[0, 0, 0, \dots, 0]$ met n het aantal processen. We vergelijken vectorklokken als volgt [37]:

$$VC_1 \leq VC_2 \leftrightarrow \forall k : VC_1[k] \leq VC_2[k]$$

Om de vergelijking in goede banen te leiden, gebruiken vectorklokken de volgende regels [37]:

1. Als een proces P_i voor een intern event zorgt, verhoogt deze zijn vectorklok als volgt: $VC_i[i] = VC_i[i] + d$ met $d > 0$
2. Als een proces P_i voor een zend event zorgt, verhoogt deze zijn vectorklok als volgt: $VC_i[i] = VC_i[i] + d$ met $d > 0$. Deze vectorklok wordt meegestuurd met de zend event.
3. Als een proces P_j een bericht met timestamp T ontvangt van P_i, gebeurt het volgende: $VC_j[k] = max(VC_j[k], T[k])$. Vervolgens verhoogt P_j zijn vectorklok als volgt: $VC_j[j] = VC_j[j] + d$ met $d > 0$

In Voldemort bestaat een versie uit een vectorklok en een master node. Als een key-value wordt geschreven in de database, krijgt dit paar een vector klok toegewezen. De node die in deze vectorklok staat, is de master node van deze key-value. We merken op dat er een vectorklok per versie wordt bijgehouden [22]. We geven een voorbeeld:

Deze versie geeft aan dat node 5 de master node is voor een gegeven key en dat deze key reeds 123 keer is aangepast. Deze master node wordt gekozen aan de hand van welke node eerst wordt bereikt door de hash ring zoals we reeds gezien hebben in Paragraaf 6.3.2. We kunnen deze methode vergelijken met de master-slave replicatie zoals we besproken hebben in Paragraaf 2.4.4.

Deze informatie volstaat om de uitleg over Voldemort te kunnen volgen. We komen in Paragraaf 6.6.4 nog terug op de versies. Voor een uitgebreidere uitleg over klokken verwijzen we naar [35].

Gebruik van Versies om te herstellen

Aangezien er hierover weinig informatie te vinden is in de Voldemort handleidingen, baseren we ons op de Amazon Dynamo paper [22]. Om de situatie te schetsen geven we eerst een voorbeeld dat ook in [22] wordt gebruikt. In dit voorbeeld bespreken we de winkelkar die we op praktisch elke webshop vinden. Het is belangrijk dat als een gebruiker een voorwerp in zijn winkelkar plaatst, deze verandering ook effectief wordt doorgevoerd, ook al is de meest recente winkelkar van de gebruiker niet beschikbaar. Deze verandering moet dan geschreven worden in een oudere, beschikbare winkelkar. Dit is nodig omdat deze verandering zinvol is voor de gebruiker. Daarnaast is het ook belangrijk dat deze winkelkar alle andere winkelkarren niet overschrijft. Samengevat kunnen we zeggen dat een verandering van de winkelkar altijd moet worden opgeslagen. Als dit niet kan in de nieuwste versie, zoeken we een oudere, beschikbare versie om dit te doen. Conflicten in verband met verschillende versies worden dan later opgelost.

Dit wordt in Amazon DB gedaan door, net zoals eerder besproken, elke verandering een vaste versie te geven. Op deze manier kunnen we dus verschillende versies van data ter beschikking hebben in onze database. In de meeste gevallen volgen de verschillende versies elkaar gewoon op. Een voorbeeld hiervan is het telkens opnieuw veranderen van een data element als er geen netwerkpartitie aanwezig is. Op deze manier is het voor het systeem heel duidelijke welke versie de nieuwste versie is. Amazon noemt dit *syntactic reconciliation*.

Problemen ontstaan echter als er netwerkpartities ontstaan en er conflicterende versies ontstaan. Een voorbeeld van conflicterende versies ontstaat wanneer er twee delen van de netwerkpartitie tegelijk dezelfde data aanpassen. Als dit gebeurt, kan het systeem niet meer zo eenvoudig weten welke versie de nieuwste is. In dit geval zal de client zelf moeten beslissen welke versie zal overleven. Hoe dit mergen gebeurt, hangt sterk af van de toepassing. Bij Amazon spreken ze dan over *semantic reconciliation*.

Als we het voorbeeld van de winkelkar er terug bijnemen, kunnen we eenvoudig inzien dat bij het mergen van twee winkelkarren nooit informatie verloren zal gaan. Het nadeel is echter wel dat eerder verwijderde producten, soms toch terug te voorschijn kunnen komen [22].

6.4 Praktische Configuratie van Voldemort

In het begin van dit Hoofdstuk hebben we de theorie achter Voldemort besproken. In het vervolg van dit Hoofdstuk testen we Voldemort. We bekijken de

achterliggende technieken en bespreken hoe Voldemort omgaat met consistentie en beschikbaarheid.

6.4.1 Configuratie

In deze Paragraaf bespreken we de verschillende mogelijke configuraties die we kunnen gebruiken bij Voldemort. We merken op dat we deze configuratie in grote lijnen bespreken. De essentie hier is te weten wat er kan, niet hoe het effectief uitgevoerd moet worden. Voor verdere uitleg in verband met deze praktische zaken verwijzen we naar de website van Voldemort [53].

Voor de configuratie worden drie bestanden gebruikt:

- cluster.xml
- stores.xml
- server.properties

In elk van deze bestanden kunnen er bepaalde instellingen gebeuren die voor ons interessant kunnen zijn.

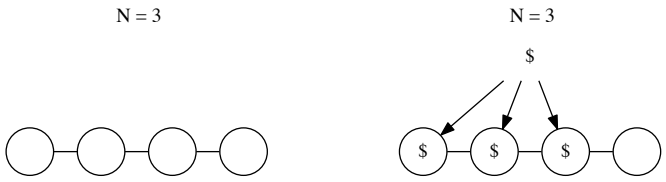
Het eerste wat we kunnen instellen, is de architectuur van ons systeem. Hiermee bedoelen we dat we het aantal nodes dat in ons systeem aanwezig is, kunnen bepalen. We kunnen hierbij ook instellen welk IP adres deze nodes hebben en op welke poort ze luisteren.

Eén van de belangrijkste instellingen voor ons experiment zijn de volgende drie variabelen [51]:

- Replication-factor (N): het aantal keren dat de data wordt opgeslagen.
- Required-reads (R): het aantal leesoperaties dat moet slagen om een leesoperatie als geldig te classificeren.
- Required-writes (W): het aantal schrijfoperaties dat moet slagen om een schrijfoperatie als geldig te classificeren.

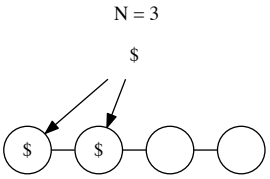
We merken op dat de N, R en W nooit groter kunnen zijn dan het aantal nodes in het netwerk. We kunnen immers de data niet over meer nodes distribueren dan we er ter beschikking hebben. Het kan immers wel nuttig zijn dat $W < N$: als $W < N$, dan we willen uiteindelijk N kopies van de data bekomen over de nodes, maar de node die de schrijfoperatie van de client afhandelt, is al tevreden van zodra er reeds W kopies geschreven zijn. De overige $N - W$ kopies kunnen achterliggend worden verzorgd nadat de client een bevestiging heeft gekregen van de eerste W schrijfoperaties (en de verbinding mogelijk al verbroken heeft). Om N beter te begrijpen, geven we een eenvoudig voorbeeld in Figuur 6.6.

Stel dat we over een systeem beschikken dat bestaat uit vier nodes. We beslissen dat de replication-factor N gelijk is aan drie. We zien dit systeem in Figuur 6.6a. Stel nu dat we een data element \$ moeten distribueren over het netwerk. In Figuur 6.6b zien we een situatie waarin de eis van $N = 3$ wordt ingevuld, aangezien \$ naar drie nodes is gekopieerd. In Figuur 6.6c daarentegen zien we een situatie waar de eis $N = 3$ niet voldaan is.



(a) Een systeem dat bestaat uit vier nodes.

(b) $ wordt gedistribueerd naar drie nodes en is bijgevolg correct.



(c) $ wordt gedistribueerd naar twee nodes en is bijgevolg niet correct.

Figuur 6.6: Schematische voorstelling van de replication-factor.

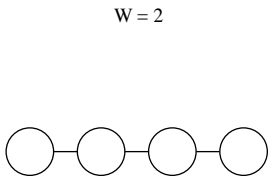
We leggen W en R uit aan de hand van hetzelfde voorbeeld met vier nodes. We beginnen met W. Deze variabele beslist hoeveel schrijfoperaties er moeten lukken vooraleer de schrijfoperatie als succesvol wordt gezien. In Figuur 6.7 geven we een voorbeeld. We gebruiken een required-writes die gelijk is aan twee. Er moeten met andere woorden minstens twee schrijfoperaties lukken alvorens het systeem deze schrijfoperatie als geldig gaat beschouwen. In Figuur 6.7b zien we dat $ wordt geschreven naar twee nodes en bijgevolg is deze schrijfoperatie correct. In Figuur 6.7c zien we dat er maar één schrijfoperatie lukt en bijgevolg dus niet voldoet aan W. Merk op dat de stippellijnen mislukte lees- of schrijfoperaties voorstellen.

Voor de required-reads gelijk aan twee is dit volledig triviaal. We tonen hier enkel de voorstellingen in Figuur 6.8.

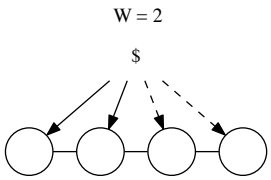
We kunnen aan de hand van deze variabelen verschillende soorten systemen creëren. Hier komen we later op terug als we het experiment effectief uitvoeren.

6.5 Situering in de CAP Theorie

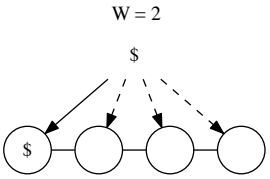
In deze Paragraaf bekijken we waar we Voldemort kunnen classificeren in de CAP theorie. Volgens [36] valt Voldemort onder de AP databases. Met andere woorden is het een beschikbare en partitie tolerante database. We kunnen echter wel de configuratie van Voldemort aanpassen waardoor we ook een CP systeem kunnen creëren. Dit is dus een consistent en partitie tolerant systeem.



(a) Een systeem dat bestaat uit vier nodes.



(b) $ wordt geschreven naar twee nodes en is bijgevolg correct.



(c) $ wordt geschreven naar één node en is bijgevolg niet correct.

Figuur 6.7: Schematische voorstelling van de required-writes.

Voldemort kan dus worden aangepast naar eigen noden. Een schematische voorstelling vinden we in Figuur 6.9.

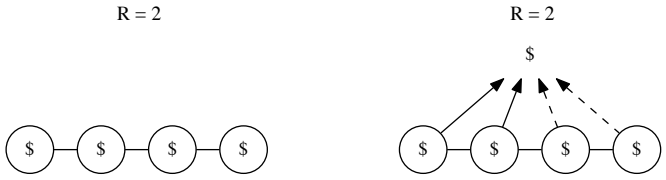
Herinner uit Paragraaf 2.4 dat we in principe maar twee soorten systemen kunnen creëren, namelijk de CA/CP systemen en de AP systemen. We komen hier later in Paragraaf 6.6.4 nog op terug.

6.6 Experiment

In deze Paragraaf beschrijven we de experimenten die we hebben uitgevoerd met Voldemort.

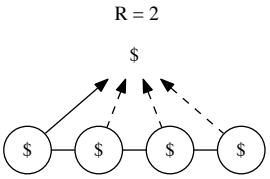
6.6.1 Doel

Een belangrijk onderdeel dat we in deze thesis bekeken hebben, is de CAP theorie uit Hoofdstuk 2. We hebben gezien dat NoSQL databases hier rekening mee moeten houden en dat in een partitie tolerant systeem (zoals Voldemort) regelmatig keuzes moeten gemaakt worden tussen enerzijds consistentie en anderzijds beschikbaarheid. In deze Paragraaf proberen we deze keuzes duidelijk zichtbaar te maken en we kijken hoe Voldemort hiermee omgaat.



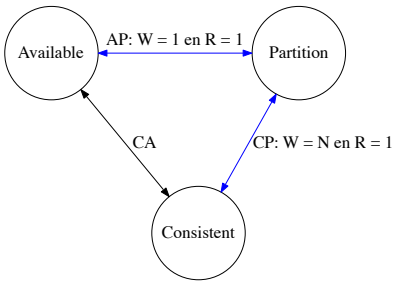
(a) Een systeem dat bestaat uit vier nodes.

(b) $ wordt gelezen uit twee nodes en is bijgevolg correct.



(c) $ wordt gelezen uit één node en is bijgevolg niet correct.

Figuur 6.8: Schematische voorstelling van de required-reads.



Figuur 6.9: Schematische voorstelling van Voldemort in de CAP theorie.

6.6.2 Idee

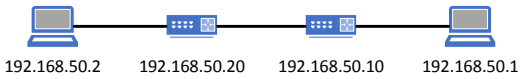
Zoals in Paragraaf 6.4.1 aangehaald, is het mogelijk om met Voldemort de parameters N, R en W aan te passen. Met deze parameters kunnen we een volledig consistent of een volledig beschikbaar systeem simuleren.

Wat we gaan doen is het volgende: we zetten een systeem op dat bestaat uit twee laptops, waarop we elk twee nodes starten. We kunnen in elke node afzonderlijk schrijven of lezen. We voegen data toe en we lezen deze uit. Vervolgens verbreken we de communicatie tussen de twee laptops en we kijken wat het resultaat is. Het resultaat zal uiteraard sterk afhangen van de configuraties

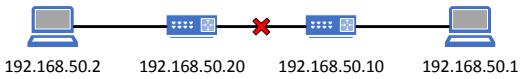
die we gebruiken.

6.6.3 Opstelling

We hebben een opstelling gemaakt die bestaat uit twee laptops en twee routers. De twee laptops blijven altijd verbonden met één router waardoor beide laptops steeds een IP adres blijven behouden. De communicatie kan vervolgens verbroken worden door de communicatie tussen de twee routers te verbreken. In Figuur 6.10a zien we een systeem zonder netwerkpartities, terwijl we in Figuur 6.10b een systeem zien waar er een netwerkpartitie ontstaat.



(a) Twee laptops zijn met elkaar verbonden via twee routers.



(b) Verbinding tussen twee routers wordt verbroken.

Figuur 6.10: Schematische voorstelling van de connectie tussen twee laptops via twee routers.

Deze opstelling geeft wel wat overhead, maar zorgt wel voor een vrij realistische opstelling.

Voor de praktische zaken in verband met het effectief opzetten van de verschillende servers, verwijzen we naar de website van Voldemort [55].

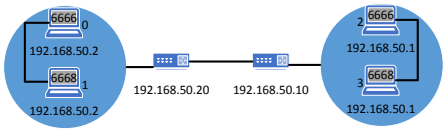
We maken dus gebruik van vier nodes die verdeeld zijn over twee laptops. In Figuur 6.11 zien we een schematisch voorstelling. We zien dat we over twee laptops beschikken die elk een ander IP adres hebben:

- Laptop 1:
  - IP: 192.168.50.2, poort: 6666 (node 0)
  - IP: 192.168.50.2, poort: 6668 (node 1)
- Laptop 2:
  - IP: 192.168.50.1, poort: 6666 (node 2)
  - IP: 192.168.50.1, poort: 6668 (node 3)

Met andere woorden kunnen we de communicatie tussen laptop 1 en laptop 2 verbreken, maar de connectie tussen de twee nodes die zich op dezelfde laptop bevinden, blijft *altijd* behouden.

6.6.4 Uitvoering

Concreet gezien, gaan we de drie situaties uit de CAP theorie testen. We bekijken eerst hoe het systeem werkt zonder netwerkpartities. Vervolgens maken we



Figuur 6.11: Schematische voorstelling van de opstelling van het experiment.

een netwerkpartitie door de ethernetkabel tussen de twee routers uit te trekken. We testen dan twee situaties:

- We verkiezen consistentie boven beschikbaarheid.
- We verkiezen beschikbaarheid boven consistentie.

Herinner dat we met vier nodes (node 0, node 1, node 2 en node 3) werken. Hierbij zijn node 0 en node 1 altijd verbonden en node 2 en node 3 altijd verbonden. We kunnen dus slechts twee situaties creëren: enerzijds een systeem waarin alle nodes verbonden zijn en anderzijds een systeem waar enkel node 0 met node 1 verbonden is en waar enkel node 2 en node 3 met elkaar verbonden zijn.

In elk experiment gaan we er van uit dat we een replicatie willen naar alle nodes, dus $N = 4$ zal altijd gelden. De waarde van W en R verschilt naarmate hetgeen we willen testen.

Merk op dat we in de volgende Paragrafen voorbeeld outputs geven van Voldemort. We geven hier niet de exacte output van Voldemort, maar we geven een intuïtieve output zodat we het verloop van de acties goed kunnen volgen.

Ten slotte geven we nog mee dat we een hash ring gebruiken die bestaat uit vier data distributies. Elke node bevat dus één data distributie.

Geen Netwerkpartitie (CA)

We beginnen met een systeem zonder netwerkpartities. Herinner uit Paragraaf 2.4 dat Abadi [1] suggereert dat CA = CP. We kunnen dit in ons achterhoofd houden tijdens de experimenten en we komen hier later nog op terug. We kiezen voor $W = 4$ en $R = 4$. Op deze manier creëren we een soort coördinatie waarbij elke *get* en *put* alle nodes zal aanspreken om tot een resultaat te komen. Als er dus slechts één node een request niet kan beantwoorden, faalt ons volledig systeem. We creëren met andere woorden een consistent en beschikbaar systeem dat niet partitie tolerant is.

We beginnen met een database die volledig leeg is. Elke query zal bijgevolg *null* als resultaat teruggeven. We gebruiken hiervoor de functie "*get* key", waarbij in ons geval de key "Masterproef" is. We zien een voorbeeld van deze queries in Listing 6.1.

```
Node 0: > get "Masterproef"
        null
Node 1: > get "Masterproef"
        null
Node 3: > get "Masterproef"
```

```
        null
Node 4: > get "Masterproef"
        null
```

Listing 6.1: Een willekeurige query op een lege database.

Verder kunnen we in één node schrijven en in een andere node lezen. Er zijn nog geen netwerkpartities dus normaal gezien zou de andere node na een zeer kleine vertraging het geschreven resultaat moeten zien. We hebben dan met andere woorden te maken met sterke consistentie zoals we gezien hebben in Paragraaf 3.2. In Listing 6.2 zien we hoe node 0 een waarde schrijft en dat node 2 dezelfde waarde probeert te lezen. Het schrijven doen we door de functie "*put* key value", waarbij in ons geval de key gelijk is aan "Masterproef" en de value gelijk is aan "NoSQL".

```
Node 0: > put "Masterproef" "NoSQL"
Node 2: > get "Masterproef"
        version(3:1): "NoSQL"
```

Listing 6.2: Schrijven en lezen van een waarde.

Merk op dat de output van Voldemort ook weergeeft met welke versie we te maken hebben. De versie van "Masterproef" staat momenteel op één. Het andere getal in de versie is belangrijk voor het vervolg van dit Hoofdstuk. Dit getal geeft weer welke node verantwoordelijk is voor deze key. De verantwoordelijke node is de master node voor een bepaalde key, zoals we besproken hebben in Paragraaf 6.3.2. In dit voorbeeld is node 3 verantwoordelijk voor de key "Masterproef". Als we nu opnieuw een waarde in "Masterproef" schrijven zien we dat het versienummer wordt verhoogd, maar dat de verantwoordelijkheid bij node 3 blijft. Een voorbeeld hiervan zien we in Listing 6.3.

```
Node 1: > put "Masterproef" "Partitie"
Node 3: > get "Masterproef"
        version(3:2): "Partitie"
```

Listing 6.3: Opnieuw een waarde schrijven om het versie nummer te verhogen.

Ten slotte kunnen we met het commando *preflist* kijken waar een gegeven key voorkomt in de nodes. Een voorbeeld hiervan zien we in Listing 6.4.

```
Node 1: > preflist "Masterproef"
        Node 3
        host:  192.168.50.1
        port: 6668
        available: yes
        last checked: 1395131880170 ms ago
```

```
Node 0
host:  192.168.50.2
port: 6666
available: yes
last checked: 1395131880169 ms ago

Node 1
host:  192.168.50.2
port: 6668
available: yes
last checked: 1395131880170 ms ago

Node 2
host:  192.168.50.1
port: 6666
available: yes
last checked: 1395131880170 ms ago
```

Listing 6.4: Output van het preflist commando.

We zien dat alle nodes beschikbaar zijn en dat ze de key "Masterproef" ter beschikking hebben. Dit is dankzij de variabele N, die we hebben ingesteld zodat alle schrijfoperaties worden doorgestuurd naar alle nodes.

Dit was een vrij eenvoudige test die de basis van Voldemort weergeeft. In de volgende Paragrafen bespreken we de problemen die opduiken als we netwerkpartities introduceren.

Netwerkpartitie (AP)

De volgende configuratie die we testen, zorgt voor een beschikbaar en partitie tolerant systeem. We kiezen $W = 1$ en $R = 1$. Op deze manier moet er telkens maar één lees- en schrijfoperatie lukken om de operaties te laten slagen. Merk op dat we op deze manier een grote kans hebben om consistentie te verliezen aangezien slechts één schrijfoperatie moet lukken. Het kan dus voorkomen dat er bijvoorbeeld een waarde in node 1 wordt geschreven en een andere waarde uit node 2 wordt gelezen.

We bouwen verder op ons vorig voorbeeld. De key "Masterproef" heeft als waarde "Partitie" met versie (3:2). We simuleren niet opnieuw het schrijven en lezen van het systeem zonder netwerkpartities, aangezien dit hetzelfde resultaat zou geven. We gaan deze keer dieper in op de netwerkpartities. We beginnen bijgevolg met het creëren van een netwerkpartitie. We verbreken de communicatie tussen de twee fysieke routers zoals reeds aangegeven in Figuur 6.10b.

We herinneren dat door het verbreken van de communicatie tussen de twee routers, er twee systemen gecreëerd worden. We hebben enerzijds node 0 en node 1 die nog met elkaar kunnen communiceren en anderzijds node 2 en node 3 die nog steeds met elkaar kunnen communiceren.

Wat we gaan doen is het volgende: een client schrijft een waarde in de key "Masterproef" in node 0 en de andere clients lezen deze key daarna van de andere nodes. Vervolgens schrijven we in node 0 en node 2 een andere waarde en kijken hoe dit conflict wordt opgelost.

We beginnen met het schrijven van een waarde in node 0. In Listing 6.5 zien we dat er een andere waarde in "Masterproef" wordt geschreven.

```
[Verbinding tussen laptop 1 en laptop 2 verbroken]
Node 0 > put "Masterproef" "Beschikbaarheid"
         Warning: kan masternode (node 3) niet bereiken
Node 0 > get "Masterproef"
         version(0:1, 3:2): "Beschikbaarheid"
Node 1 > get "Masterproef"
         version(0:1, 3:2): "Beschikbaarheid"
Node 2 > get "Masterproef"
         version(3:2): "Partitie"
Node 3 > get "Masterproef"
         version(3:2): "Partitie"
```

Listing 6.5: Schrijven van waardes terwijl er een netwerkpartitie aanwezig is.

Na het *put* commando wordt er onmiddellijk een warning getoond dat de master node niet kan worden bereikt. De master node is nodig aangezien we met een master-slave architectuur aan het werken zijn. Daarnaast moeten we ons eraan herinneren dat we nog steeds met $N = 4$ aan het werken zijn. De data moet dus uiteindelijk naar de vier nodes worden gekopieerd. Verder doet Voldemort wat we verwachten. Er is momenteel een netwerkpartitie aanwezig en bij het schrijven in node 0, kan enkel node 1 nog synchronizeren. Dit zien we door de twee gets op node 0 en node 1. Merk op dat dit voldoende is om de eis van $W = 1$ te respecteren. Wat er opvalt is dat er een andere versie wordt gemaakt. De key "Masterproef" bevat nu een algemene versie en een versie die eigen is voor de netwerkpartitie die er nu heerst. Voor deze netwerkpartitie wordt eveneens ook een andere master node voorzien, aangezien de oorspronkelijke master node 3 niet meer bereikbaar is. Node 2 en node 3 hebben nog geen idee dat er een netwerkpartitie aanwezig is. Als node 2 een *get* doet, is één leesoperatie voldoende en zal bijgevolg de oude waarde nog worden teruggegeven. Node 2 ligt ook in het bereik van de master node dus deze node kan hier altijd mee synchroniseren. In dit voorbeeld behouden we beschikbaarheid, maar we geven consistentie op.

Node 2 en node 3 gaan een netwerkpartitie opmerken vanaf het moment dat we opnieuw gaan schrijven. Ook hier volstaat in principe één voltooide schrijfoperatie, maar door het feit dat Voldemort vier replicaties van deze data wil maken, zal de database proberen om de andere nodes te bereiken. Vanaf dat moment zal er opgemerkt worden dat er een netwerkpartitie aanwezig is.

In Listing 6.6 laten we node 2 (het andere deel van de netwerkpartitie) een waarde schrijven. We zien opnieuw dat er een foutmelding wordt gegeven, maar dit keer wordt de versie geüpdatet. De reden hiervoor is dat node 3, de master node is voor de key "Masterproef". Bijgevolg moet hij dus geen rekening houden met een tweede versie. Bij de leesoperatie zien we dat node 0 en node 1 dezelfde waarde teruggeven als in Listing 6.5, maar dat node 2 en node 3 nu hun eigen nieuwe waarde hebben.

```
Node 2 > put "Masterproef" "Herstellen"
         Warning: kan laptop 1 niet bereiken
Node 0 > get "Masterproef"
         version(0:1, 3:2): "Beschikbaarheid"
Node 1 > get "Masterproef"
         version(0:1, 3:2): "Beschikbaarheid"
Node 2 > get "Masterproef"
         version(3:3): "Herstellen"
Node 3 > get "Masterproef"
         version(3:3): "Herstellen"
```

Listing 6.6: Schrijven van waardes terwijl er een netwerkpartitie aanwezig is.

Om te kijken hoe Voldemort omgaat met het herstellen van een netwerkpartitie schrijven we nog een laatste waarde in node 0, we herstellen de communicatie en we kijken wat de waarde is van de key "Masterproef". Dit zien we in Listing 6.7.

```
Node 0 > put "Masterproef" "Nieuwste"
         Warning: kan masternode (node 3) niet bereiken
Node 0 > get "Masterproef"
         version(0:2, 3:2): "Nieuwste"
[Verbinding tussen laptop 1 en laptop 2 hersteld]
Node 0 > get "Masterproef"
         version(3:3) "Herstellen"
Node 1 > get "Masterproef"
         version(3:3) "Herstellen"
Node 2 > get "Masterproef"
         version(3:3) "Herstellen"
Node 3 > get "Masterproef"
         version(3:3) "Herstellen"
```

Listing 6.7: Schrijven van waardes terwijl er een netwerkpartitie aanwezig is en deze later herstellen.

We zien dat Voldemort de waarde van de master node heeft gepropageerd en op deze manier wordt de waarde die in de netwerkpartitie geschreven is, ongedaan gemaakt. Er is hier geen syntactic reconciliation mogelijk aangezien de vectorklokken niet te vergelijken zijn.

Als we opnieuw de verbinding verbreken en we lezen opnieuw de waardes van de key "Masterproef" dan worden opnieuw de waardes gebruikt zoals ze voorkwamen in de twee netwerkpartities. Dit is weer een indicatie dat er geen syntactic reconciliation gebeurd is. We zien dit in Listing 6.8.

```
[Verbinding tussen Laptop 1 en Laptop 2 verbroken]
Node 0 > get "Masterproef"
         version(0:2, 3:2): "Nieuwste"
Node 1 > get "Masterproef"
```

```
        version(0:2, 3:2): "Nieuwste"
Node 2 > get "Masterproef"
        version(3:3): "Herstellen"
Node 3 > get "Masterproef"
        version(3:3): "Herstellen"
```

Listing 6.8: Gets bij het opnieuw verbreken van de communicatie.

In de volgende Paragraaf bespreken we het andere principe, waar we consistentie boven beschikbaarheid verkiezen.

Netwerkpartitie (CP)

De laatste test die we uitvoeren is deze met de configuratie voor een consistent en partitie tolerant systeem. We kiezen daarom voor $R = 1$ en $W = 4$. Op deze manier moeten de schrijfoperaties naar alle vier de nodes worden gepropageerd, alvorens de schrijfoperatie geslaagd is. Door $R = 1$ creëren we een systeem dat altijd beschikbaar is voor leesoperaties. Dit is toegestaan omdat we door de eis van $W = 4$ er toch voor zorgen dat elke node dezelfde waarde bevat. Het systeem zal dus enkel niet beschikbaar zijn voor schrijfoperaties tijdens een netwerkpartitie.

We bouwen opnieuw verder op ons voorbeeld en we beginnen met een systeem dat geen netwerkpartities vertoont en bijgevolg consistente data bevat. De key "Masterproef" heeft als waarde "Herstellen" met versie (3:3).

We gaan in deze test de verbinding verbreken en proberen om data te schrijven in een node. We gaan zien dat we in dit soort systeem veel beperkter zijn qua mogelijkheden.

In Listing 6.9 zien we hoe node 0 en node 2 een waarde proberen te schrijven.

```
[Verbinding tussen Laptop 1 en Laptop 2 verbroken]
Node 0 > put "Masterproef" "Nieuwste"
        Error: 4 writes nodig, 2 writes geslaagd
Node 0 > get "Masterproef"
        version(3:3): "Herstellen"
Node 1 > get "Masterproef"
        version(3:3): "Herstellen"
Node 2 > put "Masterproef" "Nieuwste"
        Error: 4 writes nodig, 2 writes geslaagd
Node 2 > get "Masterproef"
        version(3:3): "Herstellen"
Node 3 > get "Masterproef"
        version(3:3): "Herstellen"
[Verbinding tussen laptop 1 en laptop 2 hersteld]
```

Listing 6.9: Puts en gets bij een netwerkpartitie.

We zien dat de schrijfoperatie mislukt omdat deze niet gepropageerd kan worden naar alle nodes. De leesoperaties lukken wel en geven de oorspronkelijke data terug. Vervolgens zien we dat de de communicatie wordt hersteld. Daarna vragen we in Listing 6.10 de waardes op van de key "Masterproef".

```
Node 0 > get "Masterproef"
         version(3:3): "Herstellen"
Node 1 > get "Masterproef"
         version(3:3): "Herstellen"
Node 2 > get "Masterproef"
         version(3:3): "Herstellen"
Node 3 > get "Masterproef"
         version(3:3): "Herstellen"
```

Listing 6.10: Gets als er geen netwerkpartitie aanwezig is.

We zien dat de waarde van "Masterproef" dezelfde waardes teruggeeft als tijdens de netwerkpartitie. We krijgen met andere woorden totaal geen schrijftoegang tot de database als er een netwerkpartitie aanwezig is. Merk op dat we de veronderstelling van Abadi [1] uit Paragraaf 2.4, dat CP en CA ongeveer dezelfde beperkingen hebben, kunnen beamen. Het kleine verschil tussen CP en CA in deze situatie is dat we bij een netwerkpartitie totaal geen toegang krijgen tot de database in een CA systeem, terwijl we wel leestoegang krijgen in een CP systeem.

6.6.5 Invloed van W op de Uitvoeringstijd

Als laatste testen we wat de invloed is van de variabele W op de uitvoeringstijd. Herinner dat de variabele W bepaalt hoeveel schrijfoperaties er moeten slagen om als succesvol te worden geclassificeerd. We merken op dat dit geen realistisch beeld geeft van tijden aangezien dit experiment met lokale laptops is uitgevoerd. Het geeft ons daarentegen wel een goede onderlinge vergelijking. We testen twee situaties: enerzijds meten we de tijden van het toevoegen van tupels en anderzijds meten we de tijden van het verwijderen van de net toegevoegde tupels. Het verwijderen van een tupel komt conceptueel ook overeen met een schrijfoperatie, omdat het een aanpassing van de data inhoudt. We gebruiken voor dit experiment dezelfde opstelling als voorheen. We gebruiken dus vier nodes waarbij we twee laptops gebruiken waarop elk twee nodes werken.

Het experiment bestaat uit twee delen. We voeren het experiment eerst uit voor de variabele $W = 1$ en vervolgens voor de variabele $W = 4$. We herinneren dat bij $W = 1$ het voldoende is om een tupel te verwijderen of toe te voegen bij één node om als geslaagd te worden aanzien. Bij $W = 4$ hebben we hier vier geslaagde toevoegingen of verwijderingen nodig.

We beginnen met $W = 1$. In Tabel 6.3 zien we een samenvatting van het experiment. We hebben het aantal tupels laten variëren en telkens de tijden gemeten.

We zien duidelijk dat het langer duurt om een tupel toe te voegen dan een tupel te verwijderen. Bij het toevoegen van een tupel moet namelijk meer gebeuren dan bij het verwijderen van een tupel. Zo moet bij het toevoegen mogelijk een datastructuur worden opgebouwd en moet er geheugen gereserveerd worden. Bij het verwijderen moet er in het beste geval enkel een pointer worden verwijderd. De bestaande data kan vervolgens later bij het toevoegen van een nieuwe tupel overschreven worden.

We doen hetzelfde voor $W = 4$ en de resultaten zien we in Tabel 6.4.

Aantal tupels	Toevoegen	Verwijderen
100	1,658 sec	1,663 sec
1000	4,318 sec	3,694 sec
10000	21,883 sec	17,384 sec
100000	134,086 sec	96,965 sec
1000000	1280,519 sec	952,375 sec
2500000	3230,006 sec	2438,474 sec

Tabel 6.3: De resultaten van het experiment bij $W = 1$.

Aantal tupels	Toevoegen	Verwijderen
100	2,276 sec	1,808 sec
1000	7,499 sec	5,537 sec
10000	33,685 sec	24,231 sec
100000	228,491 sec	192,231 sec
1000000	2262,529 sec	1897,597 sec
2500000	5687,902 sec	4709,055 sec

Tabel 6.4: De resultaten van het experiment bij $W = 4$.

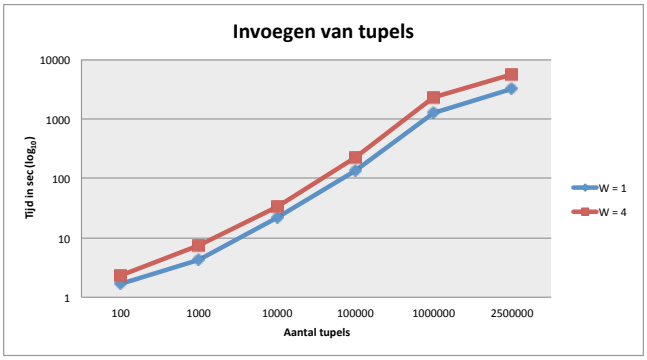
Ook hier zien we hetzelfde patroon als bij $W = 1$, namelijk dat het toevoegen langer duurt dan het verwijderen. Dit is echter niet zo belangrijk. Het belangrijkste wat we nu kunnen onderzoeken is het verband tussen $W = 1$ en $W = 4$. Om dit duidelijk weer te kunnen geven, gebruiken we een grafiek waarbij we de Y-assen een logaritmische schaal geven. Dit doen we omdat we op deze manier een duidelijk beeld van de situatie kunnen schetsen. We geven in Figuur 6.12a de onderlinge resultaten van het toevoegen en in Figuur 6.12b de onderlinge resultaten van het verwijderen.

We zien dat beide grafieken een vrij lineair verloop hebben. Wat sterk opvalt is dat de grafieken van $W = 1$ altijd onder de grafieken van $W = 4$ blijven. Dit wil dus zeggen dat ons vermoeden wordt bevestigd. Het garanderen dat alle operaties worden toegepast, heeft een kost qua uitvoeringstijd. Er moet namelijk altijd worden gewacht tot alle operaties zijn toegepast alvorens de client een nieuwe schrijfoperatie mag uitsturen. Dit is in tegenstelling tot $W = 1$ waarbij de operatie maar bij één node moet slagen. De rest van de nodes worden asynchroon behandeld.

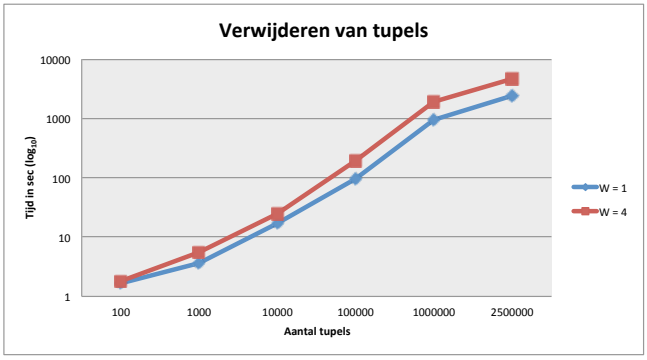
6.7 Conclusie

In dit Hoofdstuk hebben we het gehad over de NoSQL database Voldemort. We hebben gezien dat Voldemort een vrij flexibel systeem is met heel wat instellingen wat betreft replicaties, leesoperaties en schrijfoperaties.

Een belangrijke troef van Voldemort is de manier waarop de data wordt verspreid over de verschillende nodes. Voldemort gebruikt voor deze replicaties consistent hashen. Deze methode werkt met een hash ring die een interval $[0, 1]$ voorstelt. Alle nodes en keys worden vervolgens gehashed naar dit interval en aan de hand van de plaats in de hash ring krijgt elke key een master node. Verder werkt Voldemort met versies om te bepalen of we met oude of nieuwe



(a) Onderling verband tussen toevoegen van tupels bij $W = 1$ en $W = 4$.



(b) Onderling verband tussen toevoegen van tupels bij $W = 1$ en $W = 4$.

Figuur 6.12: Onderling verband tussen de uitvoeringstijden van $W = 1$ en $W = 4$.

data aan het werken zijn.

In de experimenten hebben we geprobeerd om alle theorie te controleren. We hebben hier vooral gebruik gemaakt van de drie variabelen N, W en R, die respectievelijk de replicatiefactor, de nodige schrijfoperaties en de nodige leesoperaties voorstellen. Aan de hand van deze variabelen hebben we verschillende soorten systemen gecreëerd. Bij deze systemen hebben we telkens data naar Voldemort geschreven en deze vervolgens weer uitgelezen (al dan niet in de aanwezigheid van netwerkpartities). Daarnaast hebben we nog gezien wat de invloed is van de variabele W op de uitvoeringstijd.

Hoofdstuk 7

MongoDB

7.1 Inleiding

In dit Hoofdstuk bespreken we de NoSQL database MongoDB. We doen dit op een analoge manier als in Hoofdstuk 6. We gaan dus opnieuw dieper in op de achterliggende werking en technieken die gebruikt worden bij MongoDB.

Als we de theorie onder de knie hebben, testen we MongoDB. We proberen enkele situaties te simuleren. Op deze manier tonen we aan dat de eerder besproken theorie effectief geïmplementeerd is door MongoDB.

7.2 Wat is MongoDB?

MongoDB is een open-source database die steunt op de document stores, zoals gezien in Paragraaf 5.4. MongoDB gebruikt een document formaat dat lijkt op JSON. Een voorbeeld over een persoon zou er in MongoDB uitzien zoals weergegeven in Listing 7.1.

```
{
        Achternaam: "Vandevelde",
        Voornaam: "Peter",
        Woonplaats: "Lummen",
        Hobby: ["Voetbal", "Lopen", "Koken"]
}
```

Listing 7.1: Een MongoDB document over een persoon.

We zien dat er telkens een veld en waarde wordt weergegeven. Zoals we ook in Paragraaf 5.4 gezien hebben, is het ook hier mogelijk om documenten te nesten. We zien hier bijvoorbeeld dat er drie hobby's worden toegekend aan het hobbyveld [42].

In de praktijk wordt MongoDB gebruikt door een heel aantal bedrijven. Zo gebruikt Ebay MongoDB voor zoeksuggesties en opslag van metadata. Een ander voorbeeld is Pearson dat MongoDB gebruikt als een onderliggende database voor het National Transcript Center [41]. Voor een volledige lijst van alle toepassingen verwijzen we naar [41].

7.3 Model

In deze Paragraaf bekijken we de achterliggende werking van MongoDB. We bekijken hier vooral hoe MongoDB de data afhandelt.

7.3.1 Replicaties

Replicaties is (net zoals in andere NoSQL databases) een belangrijk concept. Replicaties zorgen ervoor dat data over verschillende nodes wordt gekopieerd. Dit bevordert onder andere de beschikbaarheid van het systeem.

In MongoDB spreekt men over een *replica set*, dit is een groep van *mongod* processen die zorgen voor de *redundantie* (verschillende kopies van de data in verschillende nodes) en dus ook voor een hoge beschikbaarheid [42]. Mongod is de primaire daemon voor het MongoDB systeem. Deze daemon regelt onder andere data requesten, data formaten en zorgt voor achtergrondoperaties [42].

De replica set bestaat uit drie soorten componenten:

- een primair component
- één of meer secundaire componenten
- één of meer arbiter componenten (zie Paragraaf 7.3.2)

Merk op dat elk component wordt geleid door een mongod proces.

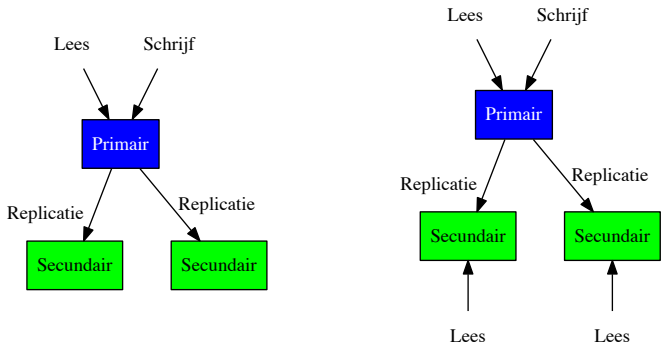
Het Primaire Component

Het *primaire component* is een component uit de replica set dat als enige schrijfoperaties van clients mag ontvangen. Alle schrijfoperaties van de clients worden dus eerst naar het primaire component gestuurd, die vervolgens dan deze schrijfoperaties verder propageert naar de secundaire componenten van de replica set. Na het ontvangen van deze schrijfoperaties kunnen de secundaire componenten deze vervolgens lokaal uitvoeren. In principe kunnen alle componenten in de replica set leesoperaties afhandelen, maar bij voorkeur worden ook de leesoperaties enkel afgehandeld door het primaire component. Verder kan er ook maar één primair component aanwezig zijn in een replica set.

Bij het eventueel uitvallen van het primaire component, wordt er door een *verkiezing* een nieuw primair component gekozen. Merk op dat een verkiezing ook altijd plaatsvindt bij het initialiseren van een nieuwe replica set [42].

We zien hier een duidelijk verband met het kopiëren van data uit Paragraaf 2.4.4. In deze Paragraaf hebben we besproken hoe alle updates eerst naar de master node worden gestuurd zodat de master node vervolgens alle updates kan propageren naar zijn slave nodes. MongoDB gebruikt in feite een exact dezelfde methode als we het primaire component beschouwen als een master node en de secundaire componenten als slave nodes.

In Figuur 7.1 zien we een schematische voorstelling van een drie-replica set met in Figuur 7.1a de configuratie waar alleen het primaire component mag lezen en in Figuur 7.1b de configuratie waar ook secundaire componenten mogen lezen. Met een drie-replica set bedoelen we dat de replica set uit drie componenten bestaat. We zien in beide Figuren één primair component en twee secundaire componenten. We bespreken de secundaire componenten hieronder.



(a) Enkel het primaire component mag lezen.

(b) Het primaire component en de secundaire componenten mogen lezen.

Figuur 7.1: Schematische voorstelling van een replica set met verschillende configuraties.

De Secundaire Componenten

Zoals we reeds gezien hebben, controleert en regelt het primaire component alle schrijfoperaties en kan het tevens ook leesoperaties afhandelen. De *secundaire componenten* zorgen voor een kopie van het primaire component. Het kopiëren van deze data wordt gedaan aan de hand van de *oplog* van het primaire component. De oplog is simpelweg de log van alle operaties die er gebeurd zijn [42]. De secundaire componenten lezen vervolgens de oplog van het primaire component en passen de log lokaal op hun eigen dataset toe. Het aanpassen van de eigen dataset door het lezen van de oplog gebeurt standaard asynchroon. Dit soort replicatie hebben we reeds besproken in Paragraaf 2.4.4. Vanaf het moment dat er één replicatie gelukt is, zal het systeem verder werken en worden de rest van de replicaties op de achtergrond afgewerkt. Dit kan echter ook worden aangepast. We gaan hier in Sectie 7.3.4 dieper op in. Zoals eerder vermeld kunnen secundaire componenten geen schrijfoperaties afhandelen, maar het is wel mogelijk dat ze leesoperaties afhandelen. We moeten dit echter wel instellen. Het is in principe dus mogelijk dat een client leest van een secundair component. Merk op dat dit kan leiden tot inconsistente data [42]. We komen hier nog op terug in Paragraaf 7.3.4.

7.3.2 Verkiezing van een Nieuw Primair Component

Zoals we nu weten, gebeurt bijna alles via het primaire component. Een vraag die we vervolgens kunnen stellen, is wat er gebeurt als het primaire component uitvalt? Als het systeem merkt dat het primaire component niet meer reageert, wordt er onmiddellijk een *verkiezing* georganiseerd om een nieuw primair component te verkiezen. Een secundair component wordt dus verkozen tot het

primaire component. Het is belangrijk dat ons systeem altijd bestaat uit een oneven aantal componenten. Dit is in principe niet nodig bij een verkiezing tijdens een normale situatie (dus zonder netwerkpartitie). In dergelijke gevallen zal het component dat reeds het langste actief is, verkozen worden tot het primaire component.

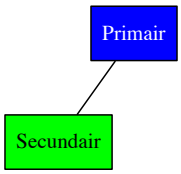
Als er daarentegen een netwerkpartitie aanwezig is, is een oneven aantal componenten wel belangrijk. Er mag namelijk maar één primair component in het systeem aanwezig zijn. Er moet dus gekozen worden in welk deel van de netwerkpartitie het primaire component aanwezig zal zijn. Als we met een oneven aantal componenten te maken hebben, zal er altijd een keuze gemaakt kunnen worden. Er zal namelijk altijd een meerderheid zijn in één van de twee delen van de netwerkpartitie. Als beide delen van de netwerkpartitie echter beschikken over hetzelfde aantal componenten, dan zal er geen consensus kunnen gemaakt worden. Dit heeft als gevolg dat er dan geen primair component gekozen kan worden. De oplossing voor dit probleem is het introduceren van een *arbiter*. Dit is een hulpcomponent om ervoor te zorgen dat een verkiezing niet op een gelijke stand kan uitdraaien. Dit component bevat dus ook geen kopies van de data en kan niet worden opgewaardeerd tot het primaire component. Dit component is "goedkoop" want het houdt geen kopies bij en kan dus ook niet voor vertragingen zorgen [42].

We illustreren dit aan de hand van een voorbeeld in Figuur 7.2. De ongerichte boog tussen de componenten stelt een continue communicatie voor. In Figuur 7.2a zien we een beginopstelling die meteen voor problemen kan zorgen als er een netwerkpartitie ontstaat. We hebben namelijk te maken met een even aantal componenten. Zoals eerder vermeld kan dit problemen veroorzaken in verband met een verkiezing. Daarom voegen we in Figuur 7.2b een arbiter component toe om een oneven aantal componenten te verkrijgen. Vervolgens zien we in Figuur 7.2c dat er een netwerkpartitie ontstaat waardoor er twee delen ontstaan. Door het oneven aantal componenten is het duidelijk dat het secundaire component van het onderste deel zal worden opgewaardeerd tot primair component. Het primaire component van het bovenste deel wordt gedegradeerd naar een secundair component. Dit zien we in Figuur 7.2d. Uitgebreidere voorbeelden van deze verkiezingen kunnen we terugvinden in Paragraaf 7.6.1.

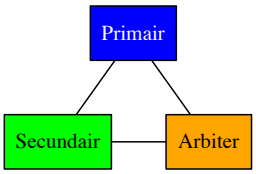
7.3.3 Hoge Beschikbaarheid

In Paragraaf 2.2 hebben we de eigenschappen van de CAP theorie gezien. Hierin hebben we gezien dat er drie eigenschappen belangrijk zijn bij het ontwikkelen van gedistribueerde systemen. Hoewel MongoDB een CP systeem is, wordt er toch veel aandacht geschonken aan beschikbaarheid. MongoDB wil dus een systeem zijn waar we in alle omstandigheden mee kunnen communiceren.

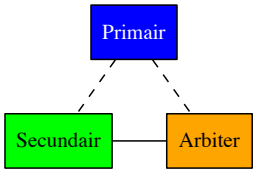
Om dit te realiseren gebruikt MongoDB een *failover*. Failover wil zeggen dat bij het uitvallen van het primaire component er een nieuw primair component wordt gekozen. We hebben deze techniek al uitgelegd in Paragraaf 7.3.1. Merk op dat het in sommige gevallen nodig is dat er een *rollback* van het systeem gebeurt na een failover. Met een rollback bedoelen we het proces waarin schrijfoperaties worden geannuleerd bij het vorige primaire component. Deze rollback moet enkel gebeuren wanneer secundaire componenten niet in staat zijn geweest om schrijfoperaties over te nemen van het primaire component omdat deze laatste bijvoorbeeld uitgevallen is. Deze schrijfoperaties zijn bijgevolg ook verloren.



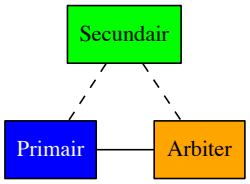
(a) Beginopstelling: een voorstelling van een twee replicatie set.



(b) We voegen een arbiter toe om een oneven aantal componenten te verkrijgen.



(c) Er ontstaat een netwerkpartitie (onderste deel heeft een meerderheid).



(d) Het secundaire component wordt het primaire component.

Figuur 7.2: Schematische voorstelling van een verkiezing met een arbiter tijdens een netwerkpartitie.

Na de verkiezing keert het primaire component terug als een secundair component omdat een ander component de taak van het primaire component heeft overgenomen. Het is daarom belangrijk dat de nog niet toegepaste schrijfoperaties een rollback ondergaan om op deze manier consistentie te laten terugkeren in het systeem. Een rollback is daarentegen niet nodig als deze schrijfoperatie vóór het uitvallen van het primaire component gekopieerd is naar een ander secundair component.

We merken op dat MongoDB in staat is om een rollback uit te voeren die minder dan 300 megabyte aan data bevat. Indien de rollback meer dan 300 megabyte data bevat, moet de gebruiker handmatig ervoor zorgen dat het systeem gesynchroniseerd wordt [42].

7.3.4 Lezen en Schrijven

Een belangrijk probleem van NoSQL databases is het al dan niet consistent zijn. MongoDB past zoals eerder vermeld een soort van master-slave replicatie toe. Alle schrijfoperaties worden dus afgehandeld door de master node of in dit geval het primaire component. Bij het schrijven, kunnen we net zoals bij Voldemort de variabele W instellen. Deze W bepaalt opnieuw hoeveel schrijfoperaties er moeten slagen alvorens de schrijfoperaties als succesvol wordt gezien. Met andere woorden worden dus W schrijfoperaties synchroon afgehandeld. Zoals we

in Paragraaf 2.4.4 over het synchroon kopiëren gezien hebben, wil dit zeggen dat het primaire component dus bevestigingen moet krijgen van W secundaire componenten. De overige schrijfoperaties ($N-W$ met N het totaal aantal componenten) worden vervolgens asynchroon afgehandeld (herinner Paragraaf 2.4.4 over het asynchroon kopiëren). In Paragraaf 7.4.2 gaan we dieper in op deze schrijfconfiguraties.

Nu kunnen we nog een keuze maken hoe de leesoperaties worden afgehandeld en dit heeft een invloed op de consistentie. We kunnen kiezen tussen:

- Enkel het primaire component handelt leesoperaties af.
- Het primaire component en de secundaire componenten mogen leesoperaties afhandelen.

Als enkel het primaire component leesoperaties uitvoert, zijn we zeker dat we altijd met consistente data te maken hebben. In Paragraaf 3.2 hebben we het gehad over sterke consistentie. De definitie die we daar gegeven hebben, zegt dat alle nodes in het systeem op eender welk moment dezelfde data moeten zien. Het kan gebeuren dat componenten van MongoDB op sommige momenten verschillende data bevatten. De gebruiker ondervindt echter een vorm van sterke consistentie aangezien voor hem de achterliggende architectuur niet zichtbaar is. Hij ervaart MongoDB als één database en niet als een aaneenschakeling van verschillende componenten. Merk op dat we in Paragraaf 2.4.4 een discussie hebben gevoerd over het feit dat we enkel één master node toelaten om leesoperaties te laten verwerken. Problemen kunnen al snel opduiken als de master node overbelast geraakt of niet meer bereikbaar is. In de documentatie wordt er weinig vermeld over deze nadelen. We hebben ze echter wel besproken in Paragraaf 2.4.4. MongoDB heeft voor de beschikbaarheid van het systeem wel het mechanisme van failover dat ervoor zorgt dat er telkens een nieuw primair component wordt verkozen als deze niet meer beschikbaar is.

De tweede keuze die we kunnen maken, is dat we primaire en secundaire componenten leesoperaties laten afhandelen. Net zoals we besproken hebben in Paragraaf 2.4.4, heeft deze methode ook een negatief punt. In dit geval gaan we weinig last hebben van beschikbaarheid bij leesoperaties aangezien er verschillende componenten leesoperaties kunnen afhandelen en op deze manier is de kans van overbelasting veel kleiner. Het nadeel is dat we een minder strikte vorm van consistentie bekomen, namelijk uiteindelijke consistentie. Deze vorm van consistentie is uitgebreid besproken in Paragraaf 3.3. We beschikken op deze manier over een heel beschikbaar systeem dat bij leesoperaties misschien inconsistente data kan teruggeven.

MongoDB geeft een mogelijke oplossing voor dit consistentieprobleem. We beschouwen het kopiëren van de data als voltooid vanaf het moment dat het primaire component alle secundaire componenten heeft voorzien van zijn data. We stellen met andere woorden W gelijk aan het totaal aantal componenten in het systeem. Dergelijke vorm van kopiëren komt overeen met het synchroon kopiëren, zoals gezien in Paragraaf 2.4.4. In de documentatie wordt opnieuw niets vermeld over het eventueel verlies van beschikbaarheid, maar we mogen stellen dat deze vorm van kopiëren toch nadelen heeft, zoals reeds besproken in Paragraaf 2.4.4 [42].

Een algemene opmerking die we geven, is dat de master-slave architectuur en de replication set niet helemaal hetzelfde zijn. Het principe komt grotendeels

overeen. Het verschil is dat master-slave minder redundantie verzorgt. MongoDB heeft redundantie door het gebruik van replicaties. Een ander verschil is dat master-slave geen automatische failover uitvoert [42].

7.4 Praktische Configuratie van MongoDB

7.4.1 Leesvoorkeuren

Standaard zorgt MongoDB ervoor dat alle leesoperaties naar het primaire component worden gestuurd om daar afgehandeld te worden. Dit is niet de ideale oplossing voor efficiëntie, maar garandeert wel dat we altijd de meest recente data lezen.

We kunnen ook alle nodes de leesoperaties laten afhandelen. Op deze manier kunnen we meer leesoperaties aan, maar zitten we met het probleem dat we oudere data kunnen lezen.

MongoDB ondersteunt echter vijf soorten voorkeuren om te lezen:

- primary
- primaryPreferred
- secondary
- secondaryPreferred
- nearest

De eerste mogelijkheid is *primary*. Alle leesoperaties gebeuren via het primaire component (master node). Dit is de standaardinstelling van MongoDB. Als er geen communicatie mogelijk is met het primaire component, zal MongoDB een foutmelding geven bij een leesoperatie.

PrimaryPreferred is de tweede mogelijkheid. Bij deze methode zal een leesoperatie eerst proberen om via het primaire component de data op te halen. In het geval dat dit niet lukt zal MongoDB lezen van een secundair component. Merk op dat er hier een risico is dat dit component niet over de recentste data beschikt.

De derde mogelijkheid is *secondary*. Deze voorkeur zorgt ervoor dat alle leesoperaties naar een secundair component worden gestuurd. Hier wordt er een foutmelding gegeven als geen enkel secundair component bereikt kan worden. Merk op dat omdat we hier van een secundair component lezen, er risico is op verouderde data.

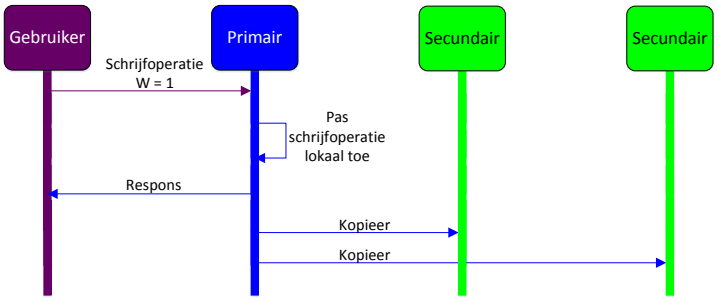
SecondairyPreferred is zeer gelijkend op de primairyPreferred. In dit geval zal er eerst geprobeerd worden om van secundaire componenten te lezen en als dit niet lukt, wordt er van het primaire component gelezen. Ook als de replica set maar uit één component (primaire component) bestaat, wordt er gelezen van het primaire component.

Als laatste mogelijkheid hebben we *nearest*. Hierbij wordt er geen rekening gehouden met de types van de componenten. Met andere woorden kan er van zowel primaire als secundaire componenten gelezen worden. MongoDB zal van het component lezen dat het dichtste bij ligt. Dit wordt bepaald door het *member selection proces* [42].

7.4.2 Schrijfvoorkeuren

Bij schrijfoperaties gebruikt MongoDB het zogenaamde *write concern* mechanisme. Write concern zorgt er in feite voor dat er aan verschillende componenten wordt gemeld of een schrijfoperatie gelukt is of niet. Deze write concern kan worden ingesteld op verschillende niveaus.

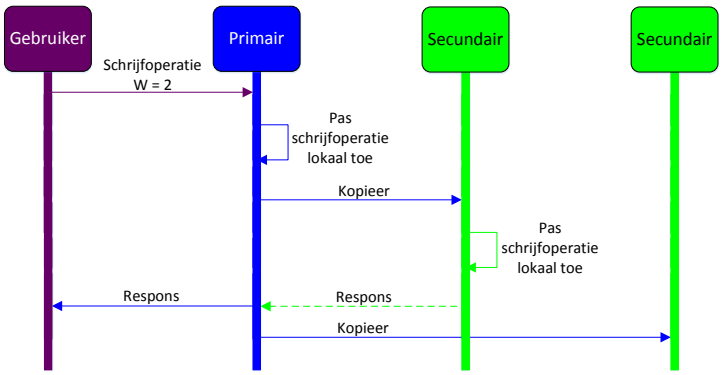
Standaard staat de write concern zo ingesteld dat er enkel een bevestiging wordt gestuurd naar het primaire component. Om dit duidelijk te maken, geven we in Figuur 7.3 een schematisch voorstelling over hoe dit in zijn werk gaat bij een drie-replica set. Merk op dat we opnieuw gebruik maken van de variabele W, zoals we die gebruikt hebben in Paragraaf 6.4.1. Deze variabele duidt aan hoeveel schrijfoperaties er moeten slagen alvorens de schrijfoperatie als geslaagd wordt geclassificeerd. We zien dat een gebruiker een schrijfoperatie stuurt naar het primaire component. Aangezien $W = 1$ moet hij deze schrijfoperatie enkel lokaal uitvoeren. De eis van $W = 1$ is bijgevolg voldaan. Daarom mag er een respons gestuurd worden naar de gebruiker dat alles goed verlopen is. Daarna kan het primaire component de schrijfoperatie asynchroon verder propageren.



Figuur 7.3: Schematische voorstelling van het schrijven met $W = 1$.

In het tweede voorbeeld hebben we te maken met $W = 2$. Een voorbeeld hiervan zien we in Figuur 7.4. Hier zien we dat de schrijfoperatie eerst weer lokaal wordt uitgevoerd. Maar omdat $W = 2$, moet er dus nog een bijkomende schrijfoperatie lukken alvorens een respons gestuurd kan worden naar de gebruiker. Het primaire component stuurt vervolgens een kopie van de geschreven data door naar een secundair component. Deze past de schrijfoperatie ook lokaal toe en stuurt vervolgens een respons naar het primaire component. Het primaire component kan nu ook een respons sturen naar de gebruiker omdat de eis van $W = 2$ voldaan is. De overige kopies worden opnieuw asynchroon doorgestuurd.

Zoals we in de voorbeelden hebben laten zien, kunnen we de parameter W specificeren. Deze W komt overeen met de required writes uit Paragraaf 6.4.1. Merk op dat als we $W > N$ instellen met N het aantal beschikbare componenten, dan zal MongoDB alle operaties blokkeren totdat er W componenten beschikbaar zijn. Hierbij moeten we voorzichtig zijn, want dit kan leiden tot een mogelijke deadlock als er geen nieuwe componenten (nodes) worden toegevoegd. Om dit tegen te gaan, kunnen we ook een bepaalde timeout instellen om ervoor te zorgen dat er toch gestopt wordt met blokkeren.



Figuur 7.4: Schematische voorstelling van het schrijven met $W = 2$.

7.5 Situering in de CAP Theorie

In deze Paragraaf bespreken we waar MongoDB wordt gesitueerd in de CAP theorie. MongoDB valt onder de CP databases. Deze database is dus consistent en partitie tolerant, maar niet altijd beschikbaar. In Figuur 7.5 zien we dit nog eens schematisch uitgelegd. De hoofdreden hiervan is dat alle schrijfoperaties moeten gebeuren via het primaire component (master node). Als dit component dus uitvalt, is er geen mogelijkheid meer om data naar de database te schrijven. Zoals we reeds gezien hebben in Paragraaf 7.3.4, zijn er twee mogelijkheden om leesoperaties af te handelen. We kunnen ervoor kiezen dat alles via het primaire component (master node) gebeurt of dat het ook via secundaire componenten (slave node) kan gaan [42]. Er zijn dus twee mogelijke vormen van beschikbaarheid als het primaire component niet bereikt kan worden. In het eerste geval, als de lees- en schrijfoperaties worden afgehandeld door het primaire component, is MongoDB totaal niet meer beschikbaar. In het andere geval als de leesoperaties kunnen worden afgehandeld door secundaire componenten, zal MongoDB niet beschikbaar zijn voor schrijfoperaties, maar wel voor leesoperaties.

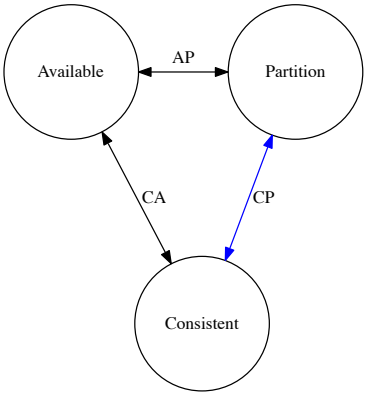
7.6 Experiment

In deze Paragraaf gaan we net zoals we voor Voldemort in Paragraaf 6.6 gedaan hebben, testen hoe MongoDB reageert op verschillende soorten situaties. We bespreken hier niet opnieuw de volledige opstelling. We gebruiken net zoals in Paragraaf 6.6.3 vier componenten[1]. Twee componenten werken op de ene laptop en de andere twee componenten werken op de andere laptop. Op deze manier kunnen we opnieuw de connectie verbreken tussen twee laptops waardoor er twee subsystemen ontstaan van telkens twee componenten.

7.6.1 De Uitvoering

Door het feit dat we met een CP systeem werken, zijn onze mogelijkheden wat betreft de testen minder vrij. We kunnen enkel de verschillende leesconfiguraties

[1] Als we deze configuratie veranderen in de loop van de experimenten wordt dit vermeld.



Figuur 7.5: Schematische voorstelling MongoDB in de CAP theorie.

testen, aangezien de schrijfconfiguratie vast ligt. We kunnen bij deze database wel de beschikbaarheid testen. Hiermee bedoelen we dat we kunnen testen hoe goed en snel het systeem reageert op een uitval van het primaire component.

Verkiezingen

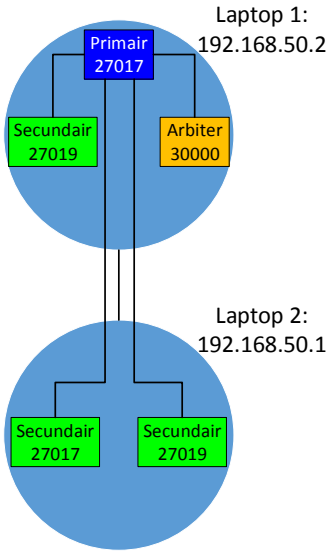
Om te beginnen gaan we de beschikbaarheid van het primaire component testen.

We beginnen met het opzetten van de replica set. We maken van node 0 het primaire component. Dit doen we door deze node te initialiseren zonder dat er nog andere nodes in de replica set zitten. Op deze manier ziet MongoDB nog maar één node (component) en zal deze dus de master node (primaire component) worden. Nu we het primaire component hebben geïnitialiseerd, kunnen we een aantal nodes (componenten) aan de replica set toevoegen. We voegen dus node 1, node 2 en node 3 toe aan de replica set. Daarnaast voegen we voor de algemeenheid nog een arbiter component toe. Dit volledig proces kunnen we vinden in Listing 7.2. Merk op dat MongoDB een replica set afkort met “rs”. Voor de praktische zaken, namelijk hoe de verschillende nodes effectief in replica set mode moeten worden opgestart, verwijzen we naar [42].

```
Node 0      > rs.initiate()
        Node 0 is primair component
Node 0 (P) > rs.add(node 1)
        Node 1 is secundair component
Node 0 (P) > rs.add(node 2)
        Node 2 is secundair component
Node 0 (P) > rs.add(node 3)
        Node 3 is secundair component
Node 0 (P) > rs.add(arbiter)
        Node 4 is arbiter component
```

Listing 7.2: Opzetten van een replica set.

We vinden deze voorstelling in Figuur 7.6. We zien dat laptop 1 uit een primair component, een secundair component en een arbiter component bestaat. Laptop 2 bestaat uit twee secundaire componenten. We zien ook op welke poort elk component luistert. Merk op dat alle componenten onderling ook met elkaar verbonden zijn, maar dit laten we achterwege om het overzicht te bewaren. Daarnaast is het ook belangrijk om op te merken dat er een connectie is tussen laptop 1 en laptop 2.



Figuur 7.6: Schematische voorstelling van de gebruikte replica set.

Op dit moment is het systeem beschikbaar voor zowel lees- als schrijfoperaties omdat er een primair component beschikbaar is. Zoals eerder vermeld, kunnen er problemen ontstaan wanneer het primaire component uitvalt of niet meer bereikbaar is. Vanaf dat moment moet er (zoals reeds uitgelegd in Paragraaf 7.3.2) een verkiezing worden gestart om zo een nieuw primair component te verkiezen.

In Listing 7.3 zien we dat we node 0 op een of andere manier stoppen. Vervolgens wordt er een verkiezing gestart en een nieuw primair component wordt verkozen. Merk op dat MongoDB zelf ziet dat er een component is weggevallen. Dit is in tegenstelling tot Voldemort die we besproken hebben in Hoofdstuk 6. Voldemort moet effectief proberen om een schrijfoperatie uit te voeren om vervolgens op te merken dat er een node niet meer beschikbaar is. MongoDB kan dit zelf heel snel opmerken door het feit dat er een soort van levenslijn tussen alle nodes (componenten) aanwezig is. Hierdoor wordt er elke tien seconden onderling een *ping* gestuurd en wordt er dus opgemerkt of er een component niet beschikbaar is. Als dit het geval is, kan er indien nodig een verkiezing worden gestart [42].

In het voorbeeld van Listing 7.3 wordt node 1 verkozen tot het nieuwe primaire component. We zien dat later node 0 hersteld wordt en dat deze wordt toegevoegd als een secundair component.

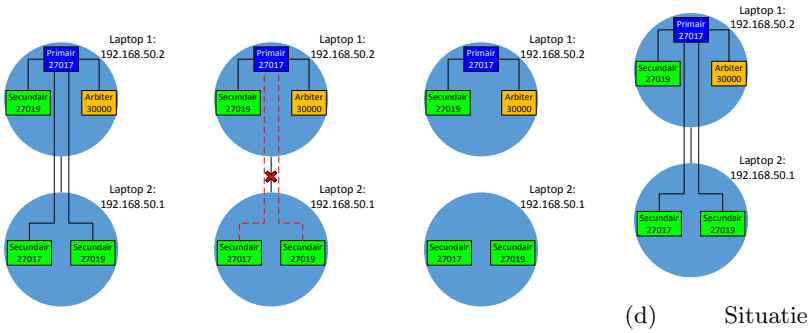
```
[Node 0 valt weg]
[Timout van 10 seconde]
[Verkiezing wordt gestart]
Node 1 (P) >
        Node 1 is primair component
[Node 0 is hersteld]
Node 1 (P) >
        Node 0 is secundair component
```

Listing 7.3: Verkiezen van een nieuw primair component.

In de vorige voorbeelden hebben we getest wat er gebeurt als het primaire component uitvalt. Een andere manier om de replica set te testen wat betreft beschikbaarheid is door een netwerkpartitie te introduceren. We kunnen een netwerkpartitie creëren door de verbinding tussen laptop 1 en laptop 2 te verbreken. Op deze manier wordt de replica set in twee stukken gekapt en zal er een beslissing genomen moeten worden in verband met wie het primaire component wordt. We testen drie situaties die we allemaal gaan voorstellen aan de hand van Figuren aangezien dit ons het duidelijkste beeld geeft. We geven eerst altijd de beginsituatie. Vervolgens zien we hoe de verbinding wordt verbroken. Dan zien we hoe de twee delen van het systeem reageren op de netwerkpartitie. Ten slotte geven we de situatie na het herstellen van de netwerkpartitie.

We beginnen met Figuur 7.7a. Hier zien we dezelfde beginsituatie als we reeds besproken hebben in Figuur 7.6, maar dit keer gaan we het primaire component niet laten uitvallen. We gaan dit keer, zoals reeds vermeld, de verbinding tussen de twee laptops laten uitvallen. In Figuur 7.7b zien we hoe de verbinding tussen de twee laptops verbroken wordt en welke gevolgen dit met zich meebrengt. We zien dat er twee delen worden gecreëerd met aan de bovenkant drie componenten en aan de onderkant twee componenten. Tijdens een netwerkpartitie probeert MongoDB altijd het primaire component te plaatsen bij de meerderheid van de componenten. In dit geval blijft het primaire component hetzelfde aangezien dit component in het deel van de meerderheid ligt. Merk op dat het systeem (indien het lezen van secundaire componenten is toegestaan) bij laptop 2 enkel leesoperaties kan ontvangen. We zien in Figuur 7.7c wat de uitkomst van de verkiezing is. In Figuur 7.7d zien we het volledige systeem als de netwerkpartitie hersteld is.

De situatie waar het primaire component zich bij de meerderheid bevindt, is de eenvoudigste situatie. De volgende situatie die we bespreken is wanneer het primaire component zich bij de minderheid bevindt tijdens een netwerkpartitie. We illustreren dit aan de hand van Figuur 7.8. In Figuur 7.8a zien we opnieuw een beginopstelling. Merk op dat het primaire component zich nu bij de andere laptop bevindt. In Figuur 7.8b zien we dat er een netwerkpartitie optreedt. Vervolgens moet er een nieuwe verkiezing gestart worden en zoals we in het vorige voorbeeld besproken hebben, wil MongoDB het primaire component altijd bij de meerderheid plaatsen. In Figuur 7.8c zien we dat het primaire component verschuift van laptop 2 naar laptop 1. Het oude primaire component wordt gedegradeerd naar een secundair component. Ook in dit geval mag er enkel gelezen worden van laptop 2. Ten slotte zien we in Figuur 7.8d de op-



(a) De beginopstelling van de replica set.

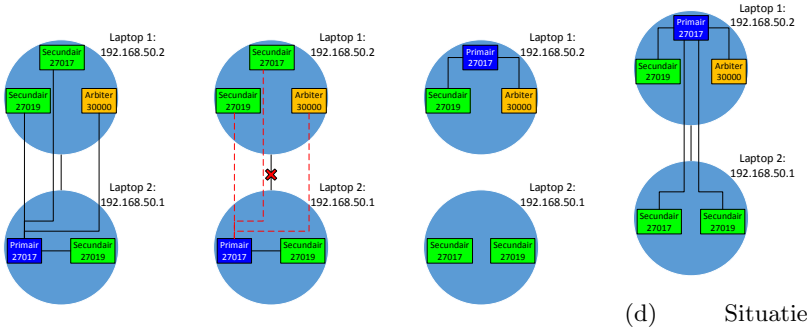
(b) Een netwerkpartitie wordt geïntroduceerd.

(c) Verkiezing tijdens de netwerkpartitie.

(d) Situatie na het herstellen van de netwerkpartitie.

Figuur 7.7: Schematische voorstelling van een verkiezing tijdens een netwerkpartitie (Primair component bevindt zich in de netwerkpartitie met de meerderheid).

stelling als de netwerkpartitie hersteld is. Merk op dat het ook mogelijk is dat er een ander primair component gekozen kan worden bij laptop 1. Dit hangt af van de verkiezingsprocedure. Tijdens het experiment werd echter deze situatie gecreëerd.



(a) De beginopstelling van de replica set.

(b) Een netwerkpartitie wordt geïntroduceerd.

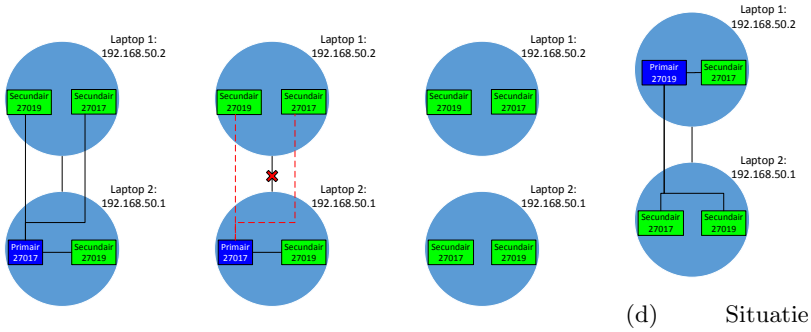
(c) Verkiezing tijdens de netwerkpartitie.

(d) Situatie na het herstellen van de netwerkpartitie.

Figuur 7.8: Schematische voorstelling van een verkiezing tijdens een netwerkpartitie (Primair component bevindt zich in de netwerkpartitie met de minderheid).

De laatste situatie die we getest hebben, is wanneer beide delen van de netwerkpartitie evenveel componenten tellen. We hebben dit gesimuleerd door het arbiter component dat reeds in vorige voorbeelden werd gebruikt, te laten vallen. We zien deze situatie in Figuur 7.9a. In Figuur 7.9b zien we net zoals in de eerdere voorbeelden hoe er een netwerkpartitie ontstaat. Als er nu een verkiezing start, kunnen we in geen enkel geval een meerderheid bereiken. In dergelijke gevallen beslist MongoDB om geen enkel component primair te maken. Hierdoor is het systeem enkel beschikbaar voor leesoperaties (als er mag gelezen worden van secundaire componenten) en niet meer voor schrijfoperaties. We zien

dit in Figuur 7.9c. Later als de netwerkpartitie hersteld wordt, kiest het systeem opnieuw een primair component door middel van een verkiezing. We zien in Figuur 7.9d dat er een nieuw primair component gekozen wordt in laptop 1.



(a) De beginopstelling van de replica set.

(b) Een netwerkpartitie wordt geïntroduceerd.

(c) Verkiezing tijdens de netwerkpartitie.

(d) Situatie na het herstellen van de netwerkpartitie.

Figuur 7.9: Schematische voorstelling van een verkiezing tijdens een netwerkpartitie (Beide delen van de netwerkpartities hebben evenveel componenten).

Schrijven

Zoals eerder vermeld, kan enkel een primair component schrijven naar MongoDB. Het schrijven naar MongoDB kan via het volgende commando:

$$db.collection.insert(document)$$

Hier specificeert *db* dat we naar de database willen schrijven. De *collection* geeft aan naar welke collection van de database we willen schrijven. Merk op dat we deze collections besproken hebben in Paragraaf 5.4 over de document stores. Vervolgens zegt *insert* dat we iets willen toevoegen aan de database. Ten slotte geven we het *document* mee dat we willen toevoegen.

Herinner dat we het experiment uitvoeren met de situatie waarmee we geëindigd zijn na Listing 7.3. Node 1 was in deze situatie het primaire component. In Listing 7.4 illustreren we dat het enkel mogelijk is om via een primair component te schrijven. Als we dit proberen via een secundair of arbiter component, geeft MongoDB een foutmelding en gebeurt er bijgevolg niets. We voegen een document in dat bestaat uit een zelfgekozen ID en een vak "Masterproef". Op deze manier kunnen we later naar dit document zoeken via het zelfgekozen ID.

```
Node 0 (S) > db.Masterproef.insert({ID: 2262,
                                    Vak : "Masterproef"})
        Error: is geen primair component
Node 1 (P) > db.Masterproef.insert({ID: 2262,
                                    Vak : "Masterproef"})
        OK
Node 2 (S) > db.Masterproef.insert({ID: 2262,
```

```
                        Vak : "Masterproef"})
        Error: is geen primair component
Node 3 (S) > db.Masterproef.insert({ID: 2262,
                        Vak : "Masterproef"})
        Error: is geen primair component
Node 4 (A) > db.Masterproef.insert({ID: 2262,
                        Vak : "Masterproef"})
        Error: is geen primair component
```

Listing 7.4: Schrijven in een replica set.

Lezen

Volgens de standaard instellingen van MongoDB is het enkel mogelijk om via het primaire component te lezen. Merk op dat we op die manier *altijd* met consistente data werken, maar dat we problemen kunnen krijgen met de werklast van het primaire component. Deze nadelen hebben we reeds besproken in Paragraaf 2.4.4. In Listing 7.5 zien we de problemen die opduiken als we proberen te lezen vanaf een secundair of arbiter component. Herinner hier opnieuw de situatie zoals weergegeven in Figuur 7.6 waarbij het primaire component nu node 1 is.

```
Node 0 (S) > db.Masterproef.find({ID: 2262})
        Error: is geen primair component
Node 1 (P) > db.Masterproef.find({ID: 2262})
        {"_id" : ObjectId("1"), "ID" : 2262,
                        "Vak" : "Masterproef"}
Node 2 (S) > db.Masterproef.find({ID: 2262})
        Error: is geen primair component
Node 3 (S) > db.Masterproef.find({ID: 2262})
        Error: is geen primair component
Node 4 (A) > db.Masterproef.find({ID: 2262})
        Error: is geen primair component
```

Listing 7.5: Lezen in een replica set.

We kunnen dit oplossen door expliciet tegen elk component te zeggen dat ze het recht krijgen om leesoperaties uit te voeren. Merk hier op dat de werklast verlaagd wordt, maar dat we niet altijd de meest consistente data lezen. In Listing 7.6 zien we het toekennen van de leesrechten.

```
Node 0 (S) > rs.slaveOk()
Node 2 (S) > rs.slaveOk()
Node 3 (S) > rs.slaveOk()
```

Listing 7.6: Secundaire componenten toelaten om ook te lezen in een replica set.

Als we Listing 7.5 opnieuw zouden uitvoeren zou iedereen (behalve het arbiter component) de juiste data moeten teruggeven.

Lezen en schrijven bij Netwerkpartities

Aangezien we voor het schrijven enkel gebruik kunnen maken van het primaire component, is het niet zo nuttig om dergelijke situaties te testen. In het begin van deze Paragraaf hebben we het gehad over de verkiezingen die plaatsvinden bij MongoDB. Dit volstaat om te kunnen inzien wat er gebeurt tijdens een netwerkpartitie. Enkel het primaire component kan nog schrijven en afhankelijk van de instellingen kunnen de secundaire componenten lezen. We gaan hier dus niet verder op in.

7.6.2 Invloed van W op de Uitvoeringstijd

Net zoals we bij Voldemort in Hoofdstuk 6, bekijken we ook hier wat de invloed is van W op de uitvoeringstijden. Herinner dat W het aantal schrijfoperaties voorstelt dat moet slagen alvorens de schrijfoperatie als geslaagd wordt geclassificeerd. We maken ook nu opnieuw gebruik van de situatie waarbij twee nodes op één laptop werken en waarbij de laptops onderling verbonden zijn. We hebben dus te maken met vier nodes. We testen het toevoegen en verwijderen van documenten bij $W = 1$ en $W = 4$. In Tabel 7.1 zien we de resultaten van het toevoegen en verwijderen van documenten bij $W = 1$.

Aantal tupels	Toevoegen	Verwijderen
100	0,088 sec	0,01 sec
1000	0,663 sec	0,089 sec
10000	6,216 sec	0,41 sec
100000	65,226 sec	3,96 sec
1000000	551,53 sec	40,348 sec
2500000	1381,718 sec	100,967 sec

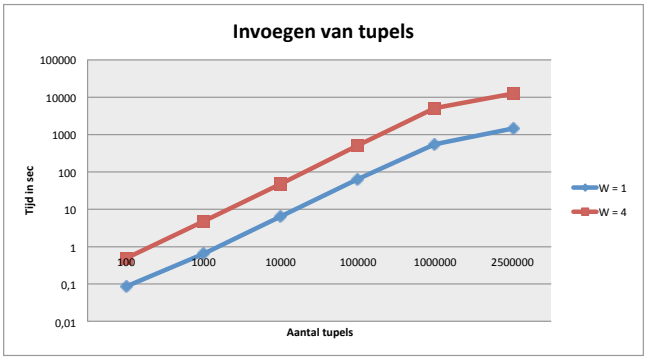
Tabel 7.1: De resultaten van het experiment bij $W = 1$.

Opnieuw valt op dat het toevoegen langer duurt dan het verwijderen van documenten. Ook hier kunnen we dit verklaren door het feit dat bij het toevoegen meer moet gebeuren dan bij het verwijderen (zie Paragraaf 6.6.5). In Tabel 7.2 zien we de resultaten van $W = 4$.

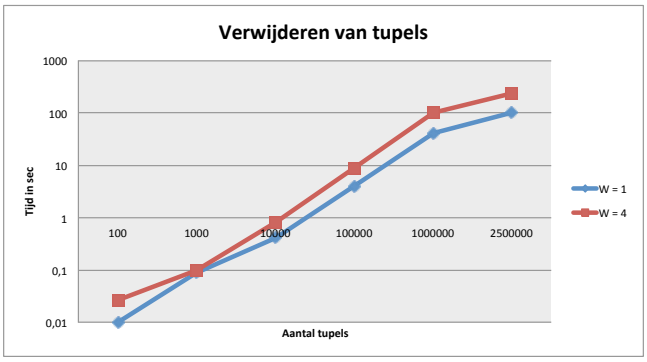
Aantal tupels	Toevoegen	Verwijderen
100	0,5 sec	0,027 sec
1000	4,673 sec	0,101 sec
10000	48,311 sec	0,819 sec
100000	502,368 sec	8,749 sec
1000000	4940,677 sec	102,793 sec
2500000	12552,687 sec	236,017 sec

Tabel 7.2: De resultaten van het experiment bij $W = 4$.

Om een duidelijk beeld te scheppen, maken we een grafiek van het toevoegen van documenten bij $W = 1$ en $W = 4$ in Figuur 7.10a. Merk op dat de Y-as opnieuw een logaritmische schaal heeft. In Figuur 7.10b maken we de grafiek van het verwijderen van documenten voor $W = 1$ en $W = 4$, ook met een logaritmische Y-as.



(a) Onderling verband tussen toevoegen van tupels bij $W = 1$ en $W = 4$.



(b) Onderling verband tussen toevoegen van tupels bij $W = 1$ en $W = 4$.

Figuur 7.10: Onderling verband tussen de uitvoeringstijden van $W = 1$ en $W = 4$.

We kunnen concluderen dat er een sterk verband is tussen de uitvoeringstijd en de variabele W. De oorzaak hiervan is dat bij $W = 1$ enkel één operatie moet slagen om door te kunnen gaan naar de volgende operatie, terwijl bij $W = 4$ de operatie bij vier nodes moet lukken om verder te mogen gaan naar de volgende operatie.

7.7 Conclusie

In dit Hoofdstuk hebben we het gehad over de NoSQL database MongoDB. We hebben gezien dat deze database partitie tolerant en consistent is. De consistentie wordt gewaarborgd door het feit dat enkel het primaire component schrijfoperaties mag uitvoeren en standaard ook enkel de leesoperaties. Door

instellingen te wijzigen, kunnen we er daarentegen wel voor zorgen dat ook de secundaire componenten leesoperaties mogen verwerken. Om toch een vrij sterke beschikbaarheid te verzorgen, maakt MongoDB gebruik van een verkiezingsprotocol dat ervoor zorgt dat het systeem niet lang zonder een primair component zal werken.

In de experimenten hebben we ons daarom vooral toegespitst op deze verkiezingen. We hebben allerlei situaties getest en gekeken welk component het primaire component zou worden bij het ontstaan van netwerkpartities.

Ten slotte hebben we opnieuw de invloed van de variabele W op de uitvoeringstijd bekeken.

Hoofdstuk 8

Conclusie

Deze conclusie bestaat uit twee delen. Het eerste deel geeft een samenvatting van alle geziene Hoofdstukken. In het tweede deel geven we een algemene conclusie over de thesis.

8.1 Samenvatting

In Hoofdstuk 1 hebben we een algemeen beeld gegeven over de relationele databases. Het belangrijkste wat we hier gezien hebben is het feit dat ze gebruik maken van het ACID principe. Relationele databases zijn met andere woorden beschikbaar (**a**vailable), consistent (**c**onsistent), geïsoleerd (**i**solated) en duurzaam (**d**urable). Daarnaast zijn we verder ingegaan op het schalen van de databases. Hier hebben we gezien dat er een verschil is tussen verticaal en horizontaal schalen. De eerste methode wordt meestal door relationele databases gebruikt, terwijl de tweede methode vooral door NoSQL databases wordt gebruikt. Als laatste hebben we in Hoofdstuk 1 gekeken naar gedistribueerde relationele databases. Hierbij is het belangrijk om te onthouden dat, hoewel ze gedistribueerd zijn, ze toch altijd het ACID principe moeten toepassen.

In Hoofdstuk 2 zijn we dieper ingegaan op de CAP theorie. De CAP theorie staat voor consistentie (**c**onsistency), beschikbaarheid (**a**vailability) en partitie tolerantie (**p**artition tolerance). We hebben gezien dat er altijd moet gekozen worden tussen twee van deze drie eigenschappen in een gedistribueerd systeem. Aangezien we het over NoSQL databases hebben, ligt de keuze van partitie tolerantie vast en moet er dus altijd een keuze worden gemaakt tussen ofwel consistentie ofwel beschikbaarheid. Daarnaast hebben we een belangrijke kritiek op de CAP theorie onder de loep genomen, namelijk dat we bij de CAP theorie geen rekening houden met de vertraging die op het netwerk kan zitten. Het begrip vertraging heeft dus ook een sterke verbondenheid met beschikbaarheid. Verder hebben we in Hoofdstuk 2 enkele belangrijke methodes bestudeerd die gebruikt worden om data te propageren over het netwerk. Hier hebben we twee varianten bekeken: enerzijds door geen gebruik te maken van een master node en anderzijds door wel gebruik te maken van een master node. Beide methodes hebben voor- en nadelen. Als laatste hebben we bekeken wat er allemaal kan en moet gebeuren als er een netwerkpartitie optreedt.

In Hoofdstuk 3 hebben we algemene principes in verband met consisten-

tie bestudeerd. Hierbij hebben we meteen onderscheid gemaakt tussen twee grote groepen, namelijk de sterke consistenties en de zwakke consistenties. We zijn vooral dieper ingegaan op de zwakke consistentie waarbij we enerzijds de uiteindelijke consistentie en anderzijds de causale consistentie hebben bekeken. De uiteindelijke consistentie zegt dat alle data uiteindelijk gesynchroniseerd zal worden. Deze eis wordt door veel NoSQL databases gebruikt. We hebben hier enkele technieken gezien die kunnen worden gebruikt om uiteindelijke consistentie op een automatisch manier te waarborgen, namelijk CRDTs en de CALM stelling. Naast ACID voor relationele databases, hebben we in Hoofdstuk 3 het design principe BASE van NoSQL databases gezien. Dit staat voor eenvoudige beschikbaarheid (**b**asic **a**vailability), zachte toestand (**s**oft-state) en uiteindelijke consistentie (**e**ventual consistency). Daarnaast hebben we ook de causale consistentie bekeken. Dit soort consistentie steunt op afhankelijkheden tussen verschillende schrijfoperaties. Schrijfoperaties waar de afhankelijkheden nog niet van zijn voldaan, worden nog niet toegepast. We hebben hier twee grote groepen gezien, namelijk de potentiële causale consistentie waarbij alle schrijfoperaties afhankelijkheden zijn en de expliciete consistentie waarbij we naar de semantiek gaan kijken wat betreft de afhankelijkheden.

In Hoofdstuk 4 zijn we dieper ingegaan op wat NoSQL is. We hebben gezien dat de meest gebruikte vertaling "Not only SQL" is. Dit wil zeggen dat NoSQL databases ook nog andere technieken gebruiken dan SQL. We hebben ons hier vooral beziggehouden met te onderzoeken wat de oorzaken zijn van de NoSQL opkomst. We mogen besluiten dat big data en de drang naar parallellisatie de grootste oorzaken zijn. Daarnaast hebben we nog enkele voor- en nadelen van NoSQL databases ten opzichte van relationele databases opgesomd.

In Hoofdstuk 5 hebben we enkele vormen van NoSQL databases besproken. We hebben eerst een voorbeeld in het relationeel model uitgewerkt en vervolgens hebben we dit voorbeeld toegepast op verschillende soorten NoSQL databases.

In Hoofstuk 6 hebben we kennis gemaakt met een bestaande key-value store, namelijk Voldemort. We hebben gezien dat er veel aanpasbaar is aan de instellingen van Voldemort waardoor we dus verschillende soorten systemen kunnen creëren. Voor de experimenten die we uitgevoerd hebben, hebben we ons echter vooral toegespitst op de CA en AP configuraties uit de CAP theorie. We hebben de achterliggende architectuur bekeken van Voldemort en we hebben gezien dat er gebruik wordt gemaakt van een hash ring en versies. In onze experimenten hebben we bestudeerd wat er gebeurt als netwerkpartities optreden. Als laatste hebben we gezien dat de variabele W een sterke invloed heeft op de uitvoeringstijd van de database. Herinner dat W het aantal schrijfoperaties voorstelt dat moet slagen alvorens de schrijfoperaties als succesvol worden geclassificeerd.

In Hoofdstuk 7 hebben we kennis gemaakt met een tweede bestaande NoSQL database. Dit keer hebben we de document store MongoDB besproken. In tegenstelling tot Voldemort hebben we niet zoveel vrijheid wat betreft de instellingen. MongoDB is een CP systeem volgens de CAP theorie. In dit systeem kan enkel het primaire component (master) schrijfoperaties uitvoeren. Leesoperaties kunnen standaard ook alleen door het primaire component worden uitgevoerd. We kunnen echter wel instellen dat de secundaire componenten ook leesoperaties mogen afhandelen. Dit kan echter wel ten koste gaan van de consistentie binnen het systeem. In de experimenten hebben we vooral gekeken wat er gebeurt bij een verkiezing tijdens een netwerkpartitie. Daarnaast hebben we net zoals bij Voldemort gekeken wat de invloed van de variabele W is op de uitvoeringstijd.

8.2 Algemene Conclusie

We kunnen besluiten dat NoSQL databases een mooi alternatief zijn voor de gewone relationele databases. We moeten echter wel voorzichtig zijn met te zeggen dat ze *de* oplossing zijn voor alle problemen waarmee relationele databases kampen. Als we willen overschakelen naar een NoSQL database moeten we zeker eerst een grondige analyse maken van het probleem. NoSQL databases zijn immers geen totaalpakket zoals de relationele databases. Bij NoSQL databases zijn er heel wat zaken waar we zelf rekening mee moeten houden. Zo moeten we soms zelf zorgen voor het samenvoegen van data na het herstellen van een netwerkpartitie en moeten we er zelf voor zorgen dat er geen foute informatie in de database wordt geplaatst (door gebrek aan een schema). Daarnaast zijn er ook heel wat instellingen die zonder grondige kennis, niet juist kunnen benut worden. Daarom mogen relationele databases zeker niet verdwijnen. Ze hebben immers een sterke theoretische onderbouwing en worden reeds lange tijd gebruikt. Overschakelen naar een NoSQL database zonder grondige redenen is dus niet aan te raden.

De CAP theorie is een onmisbare theorie als we het hebben over gedistribueerde databases. Er moet altijd gekozen worden tussen twee van de drie eigenschappen. In deze thesis hebben we meestal de eigenschap van partitie tolerantie vastgezet omdat dit een onmisbare eigenschap is van een NoSQL database. De keuze moet dus gemaakt worden tussen consistentie en beschikbaarheid. Welke eigenschap er effectief moet gekozen worden, hangt af van het soort systeem dat we willen creëren. Met andere woorden is er geen eenduidig antwoord op deze vraag. We hebben echter wel gezien dat bij het kiezen van beschikbaarheid er toch een vorm van consistentie blijft bestaan. We hebben dit uiteindelijke consistentie genoemd. We kunnen dus inconsistente data lezen, maar we zijn zeker dat we ergens in de toekomst consistente data zullen lezen. Het is belangrijk dat we bij het kiezen van enerzijds consistentie en anderzijds beschikbaarheid de voor- en nadelen goed in ons achterhoofd houden.

Bibliografie

[1] Daniel Abadi. Problems with cap, and yahoo's little known nosql system. `http://dbmsmusings.blogspot.be/2010/04/problems-with-cap-and-yahoos-little.html`, april 2010.

[2] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

[3] Haya Ajjan and Richard Hartshorne. Investigating faculty decisions to adopt web 2.0 technologies: Theory and empirical tests. *The Internet and Higher Education*, 11(2):71 – 80, 2008.

[4] Peter Alvaro, Peter Bailis, Neil Conway, Joe Hellerstein, Bill Marczak, and Sriram Srinivasan. bloom. `http://www.bloom-lang.net`.

[5] Peter Alvaro, Neil Conway, Joe Hellerstein, Bill Marczak, and Sriram Srinivasan. Calm: consistency as logical monotonicity. `http://www.bloom-lang.net/calm/`.

[6] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260. www.cidrdb.org, 2011.

[7] Amazon. Amazon dynamodb. `http://aws.amazon.com/dynamodb/`.

[8] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In Michael J. Carey and Steven Hand, editors, *SoCC*, page 22. ACM, 2012.

[9] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, 2013.

[10] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Pbs: Probabilistically bounded staleness. `http://pbs.cs.berkeley.edu`.

[11] Simone Benefico, Eva Gjeci, Ricardo Gonzalez Gomarasca, Eros Lever, Santo Lombardo, Danilo Ardagna, and Elisabetta Di Nitto. Evaluation of the cap properties on amazon simpledb and windows azure table storage. In *SYNASC*, pages 430–435. IEEE Computer Society, 2012.

[12] Eric Brewer. Cap twelve years later: How the "rules"have changed. http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed, mei 2012.

[13] Julian Browne. Brewer's cap theorem the kool aid amazon and ebay have been drinking. http://www.julianbrowne.com/article/viewer/brewers-cap-theorem, januari 2009.

[14] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system r. *Commun. ACM*, 24(10):632–646, 1981.

[15] Mike Chapple. Abandoning acid in favor of base: Changes to the long-held relational model. http://databases.about.com/od/otherdatabases/a/Abandoning-Acid-In-Favor-Of-Base.htm.

[16] Mike Chapple. The acid model. http://databases.about.com/od/specificproducts/a/acid.htm.

[17] Bobbie J. Cochrane and Kathy A. McKnight. Db2 json capabilities, part 1: Introduction to db2 json. http://www.ibm.com/developerworks/data/library/techarticle/dm-1306nosqlforjson1/, juni 2013.

[18] Jeff Cogswell. Nosql document storage benefits and drawbacks. http://slashdot.org/topic/bi/nosql-document-storage-benefits-drawbacks/, juni 2012.

[19] Jeff Cogswell. Sql vs. nosql: Which is better? http://slashdot.org/topic/bi/sql-vs-nosql-which-is-better/, juli 2012.

[20] Couchbase. Why nosql? http://www.couchbase.com/why-nosql/nosql-database.

[21] Olivier Curé, Myriam Lamolle, and Chan Le Duc. Ontology based data integration over document and column family oriented nosql. *CoRR*, abs/1307.2603, 2013.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 205–220. ACM, 2007.

[23] Larry Dyson. The four horsemen of nosql. http://www.modelmetrics.com/technology-viewpoint/the-four-horsemen-of-nosql/, juli 2012.

[24] Benjamin Erb. Scalibility. http://berb.github.io/diploma-thesis/original/024_scalability.html.

[25] Santhosh Kumar Gajendran. A survey on nosql databases. https://wiki.engr.illinois.edu/download/attachments/204768516/A+Survey+on+NoSQL+Databases.pdf?version=2&modificationDate=1354595521000.

[26] Hector Garcia-Molina, Nancy Lynch, Barbara Blaustein, Charles Kaufman, Sunil Sarin, and Oded Shmueli. Notes on a reliable broadcast protocol. 1985.

[27] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[28] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[29] Jeffrey A. Hoffer, Ramesh Venkataraman, and Heikki Topi. *Modern Database Management*. Prentice Hall Press, Upper Saddle River, NJ, USA, 10th edition, 2010.

[30] Judith Hurwitz, Alan Nugent, Fern Halper, and Marcia Kaufman. Key-value pair databases in a big data environment. http://www.dummies.com/how-to/content/keyvalue-pair-databases-in-a-big-data-environment.html.

[31] Judith Hurwitz, Alan Nugent, Fern Halper, and Marcia Kaufman. Why are multi valued databases so good? http://www.kallal.ca/Articles/fog0000000006.html, december 2011.

[32] Maria Indrawan-Santiago. Database research: Are we at a crossroad? reflection on nosql. In Leonard Barolli, David Taniar, Tomoya Enokido, J. Wenny Rahayu, and Makoto Takizawa, editors, *NBiS*, pages 45–51. IEEE, 2012.

[33] Amrith Kumar. The nosql vs. sql hoopla, general purpose vs. specialization. http://www.parelastic.com/blog/nosql-vs-sql-hoopla-general-purpose-vs-specialization, april 2012.

[34] Girish Kumar and Rahul Checker. Exploring the different types of nosql databases. http://blog.3pillarglobal.com/exploring-different-types-nosql-databases, oktober 2013.

[35] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, juli 1978.

[36] Sang Hun Lee. Visual guide to nosql systems. http://blog.beany.co.kr/archives/275, maart 2011.

[37] Sangyoon Lee, Ajay D. Kshemkalyani, and Min Shen. Performance evaluation of incremental vector clocks. In *ISPDC*, pages 117–124. IEEE, 2011.

[38] Adam Marcus. The nosql ecosystem. http://www.aosabook.org/en/nosql.html.

[39] Christopher Mims. Why cpus aren't getting any faster. http://www.technologyreview.com/view/421186/why-cpus-arent-getting-any-faster/, oktober 2010.

[40] MongoDB. Data modeling introduction. http://docs.mongodb.org/manual/core/data-modeling-introduction/.

[41] Inc. MongoDB. Production deployments. http://www.mongodb.org/about/production-deployments/, 2013.

[42] Inc. MongoDB. Mongodb documentation. http://docs.mongodb.org/manual/MongoDB-manual.pdf, mei 2014.

[43] Michael Nielsen. Consistent hashing. http://michaelnielsen.org/blog/consistent-hashing/, juni 2009.

[44] Oracle. Oracle nosql database. http://www.oracle.com/technetwork/database/nosqldb/learnmore/nosql-database-498041.pdf, September 2011.

[45] Dave Pinal. Big data what is big data 3 vs of big data volume, velocity and variety day 2 of 21. http://blog.sqlauthority.com/2013/10/02/big-data-what-is-big-data-3-vs-of-big-data-volume-velocity-and-variety-day-2-of-21/, oktober 2013.

[46] Nuno M. Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *ICDCS*, pages 395–403. IEEE Computer Society, 2009.

[47] Rocket Software. What is multivalue? https://u2devzone.rocketsoftware.com/ignite/test.html.

[48] Michael Stonebraker. The "nosql" discussion has nothing to do with sql. http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext, november 2009.

[49] Michael Stonebraker. Sql databases v. nosql databases. *Commun. ACM*, 53(4):10–11, 2010.

[50] Dave Valentine. Rules of engagement – nosql graph databases. http://www.ingenioussql.com/2013/03/27/rules-of-engagement-nosql-graphing-databases/, maart 2013.

[51] Werner Vogels. All things distributed. http://www.allthingsdistributed.com/2007/12/eventually_consistent.html, december 2007.

[52] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[53] Project Voldemort. Configuration. http://www.project-voldemort.com/voldemort/configuration.html.

[54] Project Voldemort. Design: Data partitioning and replication. http://www.project-voldemort.com/voldemort/design.html.

[55] Project Voldemort. Quickstart. `http://www.project-voldemort.com/voldemort/quickstart.html`.

[56] Project Voldemort. Voldemort is a distributed key-value storage system. `http://www.project-voldemort.com/voldemort/`.

[57] the free encyclopedia Wikipedia. Cap theorem. `http://en.wikipedia.org/wiki/CAP_theorem`.

[58] Cindy Wong. Overview of db2's xml capabilities: An introduction to sql/xml functions in db2 udb and the db2 xml extender. `http://www.ibm.com/developerworks/data/library/techarticle/dm-0311wong/`, november 2003.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
NoSQL: een vergelijkende studie

Richting: **master in de informatica-databases**
Jaar: **2014**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Cich, Glenn

Datum: **10/06/2014**